Lab 4: Single-Core & Multi-Core Systems

Asma Ansari (ara89) & Shencheng Xu (sx286)

I. Introduction

Despite the significant improvements in CPU performance in the past century, limitations in the physical design of CPUs have forced computer engineers to develop clever methods to improve performance. Namely, multi-core processors are a key design that has propelled CPU development forward. Multi-core processors improve performance by allowing programs to operate in parallel by creating threads that run on each core. Single-core processors can only process one thread at a time, so providing the processor the bandwidth for processing several threads is significant for efficiency. This lab explores the differences between each design, especially the tradeoffs with regards to energy consumption, area, and cycles per instruction (CPI).

II. Alternative Design

A. Hardware

In a single-core system, processor memory requests can be sent directly to the cache. However, in a multi-core system, we cannot simply connect the four pipelined processors, four instruction caches, four data caches, and memory together. This is particularly critical for data caches, as in reality, a memory request from processor 1 might need to be routed to cache 3. Ignoring this would result in significant loss of parallelism for the processors. Thus, the structure of our multi-core system is illustrated in Figure 1. The multi-core data cache handles data requests from the processor, with its structure shown in Figure 2, which includes the cache network (Figure 3(a)) and the memory network (Figure 3(b)) along with the four caches. In addition, there is another memory network connecting the instruction cache to memory.

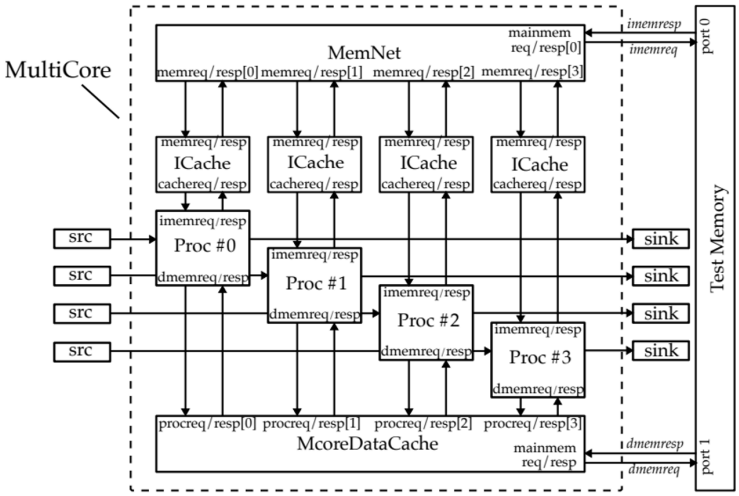


Figure 1: Multi-core system

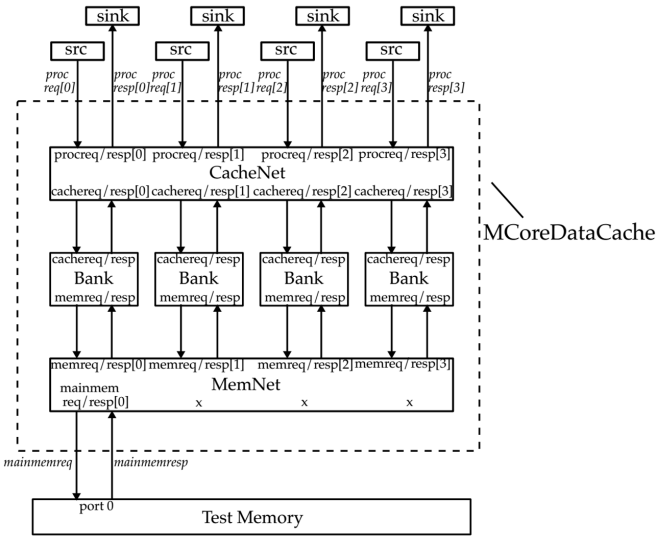


Figure 2: Multi-core data cache

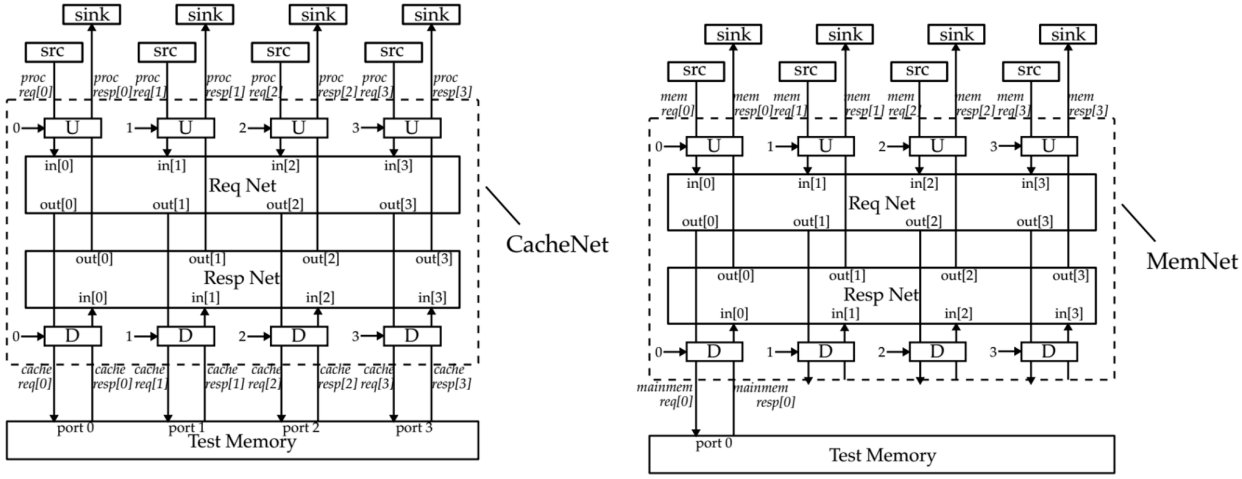


Figure 3: (a) Cache net (b) Memory Net

For the cache network and the memory network, we use a ring network structure as shown in Figure 4. The network has four input terminals and four output terminals, and it is composed of four routers, which has four input stream interfaces and four output stream interfaces. The function of the ring network is to interconnect the processors to the data caches and to interconnect the caches to the main memory interfaces. Each network consists of a request network and a response network, as well as four upstream adapters, and four downstream adapters. The function of those adapters is that we need to convert the requests to the cache and the responses from the memory into a format that can be processed by the ring network, as shown in Figure 5. The upstream adapters can convert cache requests into network messages, and downstream adapters can convert network messages into corresponding cache requests. The source field and destination field in a network message are used to indicate which port the message is from and to which port it should be delivered. More specifically, when a processor requests the cache, the upstream adapter sets the network message's source field according to the port number of the input. Once the

downstream adapter receives the message, it stores the source and destination fields in the cache request's opaque field before sending it to the cache. When the cache response is returned from the cache, the downstream adapter sets the target output port of the response message by examining the opaque field in the cache response and then appropriately sets the source field.

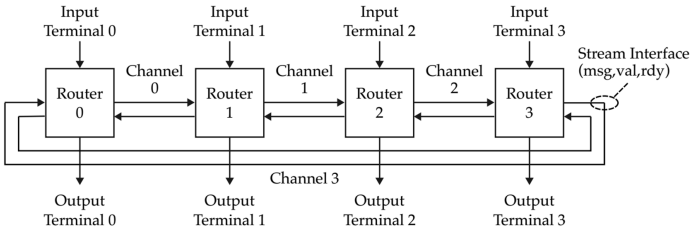


Figure 4: Ring Network

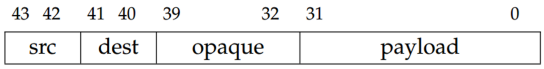


Figure 5: Network Message Format

The structure of our routers is shown in Figure 6.Each router consists of three input queues, three route units, and three switch units.

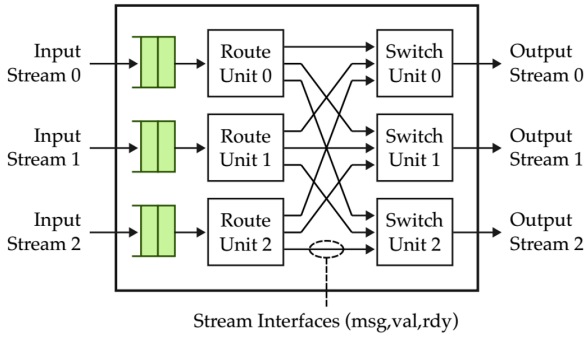


Figure 6: Ring Network Router

The route unit determines the shortest path for message delivery based on the current router ID and the destination field in the network message. It calculates the number of steps needed for both clockwise and counterclockwise directions and selects the shorter path. If the path lengths are equal, the clockwise direction is given priority.

Our switch unit employs a round-robin arbitration algorithm for more fair scheduling, avoiding the starvation issues that can arise in simple priority-based scheduling algorithms, where lower-priority tasks might go unserved for long periods. In our switch unit, the last processed input port is recorded, and the next arbitration starts from the following port in a fixed cyclic order. The arbitration sequence we use is clockwise, counterclockwise, and port input, in a circular order. When the inputs in all 3 directions are valid, we assign the priority of these 3 ports in a circular manner. This sequence is designed because the clockwise and counterclockwise messages already exist in the ring network, while the port input represents new messages entering the system during the current cycle. As shown in Figure 7, where # indicates that all three ports of the switch unit have valid messages, the output port

correctly outputs messages according to our arbitration algorithm. In contrast, with a fixed-priority algorithm, as illustrated in Figure 8, the output port must wait for the higher-priority clockwise message to be processed before handling messages with lower priority.

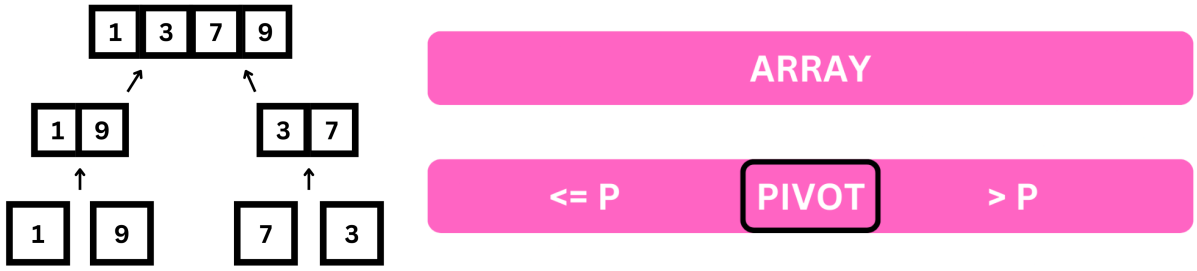
Input Terminal	Clockwise	Counterclockwise	Output Terminal
#	1>0:00	#	1>0:00
#	#	2>0:00	2>0:00
0>0:00	#	#	0>0:00
#	1>0:01	#	1>0:01
#	#	2>0:01	2>0:01
0>0:01	#	#	0>0:01
#	1>0:02	#	1>0:02
…	…	…	…

Figure 7: Part of the line trace of test case "stream_from_all1" under round-robin algorithm

Input Terminal	Clockwise	Counterclockwise	Output Terminal
#	1>0:00	#	1>0:00
#	1>0:01	#	1>0:01
#	1>0:02	#	1>0:02
#	#	2>0:00	2>0:00
#	#	2>0:01	2>0:01
#	#	2>0:02	2>0:02
0>0:00	#	#	0>0:00
0>0:01	#	#	0>0:01
0>0:02	#	#	0>0:02

Figure 8: Part of the line trace of test case "stream_from_all1" under fixed priority algorithm

B. Software



(a) (b)

Figure 9: Illustrations of (a) mergeSort() (b) quickSort()

Initially, we chose to use mergeSort() for our baseline sorting algorithm, but this utilized significantly more cycles and instructions than other microbenchmarks. As per the lab handout, we needed to prioritize speed for the sorting algorithm, especially for the single-core processor. Therefore, the baseline sorting algorithm implements quickSort(), a lightweight sorting algorithm that splits an array at a pivot point and recursively sorts the algorithm.

The sorting algorithm we designed for the alternative design utilized the sorting algorithm made for the baseline design and combined it with mergeSort(). For the multithreaded algorithm, we chose to split the array into four "blocks" which each thread sorted using the ubmark_sort() function. To merge all of these blocks back into the original array, we implemented merge() from mergeSort() and created another helper function called mergeBlocks() to take care of this. Essentially, our design splits up the sorting work between each thread and merges every block back together, similar to the illustration above.

Our multithreaded algorithm uses hierarchical and modular design due to the helper functions we implemented. The helper functions, merge(), mergeBlocks(), and sortBlocks(), allowed each thread to function independently until they finished sorting their respective blocks. Furthermore, we had to be careful about allocating and deallocating memory such that the only calls to ece4750_malloc() and ece4750_free() occurred in mtbmark_sort(), essentially the main() function for this algorithm. We utilized the provided threading library to initialize all threads correctly and handle their behavior as the program progressed.

III. Testing Strategy

A. Hardware

As with the design process, we use a bottom-up testing model, starting with the lowest level units.

1) Route Unit

Since our router unit calculates the shortest path based on the destination and the current router ID, the routing unit determines the shortest path and forwards the message to the corresponding direction: sink0 (directly to the output terminal), sink1 (clockwise), or sink2 (counterclockwise). Therefore, when testing the routing unit, we need to set different destination positions to observe whether network messages are delivered to the expected ports for routers with different IDs.

Table 1 shows the test cases we used when evaluating the routing unit. All four routing units, each with a different ID, use the same test cases but can produce different results. Thus, we need to use the same logic as in the RTL code to determine if the results are correct. Our testing strategy is categorized into directed and random tests. In the directed tests, we began with basic tests by inputting a few messages(1 to 4) and setting different destination positions to verify the fundamental functionality. Then, in the "Stream to one port" test cases, we gradually increased the number of messages injected into the routing unit, directing multiple messages to a single port. Finally, we combined test cases in the "Stream to one port", sending multiple messages to different ports. Additionally, although the routing unit does not specifically consider the source of messages, we included tests to evaluate this aspect. After passing all the directed tests, we designed random tests to ensure there were no overlooked scenarios in the directed tests. We used three types of random tests for simulation: "Random Source", "Random Destination", and "Fully Random". Lastly, we used the previous test cases with varying source delays, sink delays, and delay modes to ensure the routing unit could handle scenarios involving delays correctly. Once the routing unit passes all the above tests and its waveform matches our expectations, it is ready to be integrated into the router.

Testing Strategy	**Test Cases**	**Introduction**
Directed	Simple	A small number of net messages sent to different ports
	Stream to one port	A large number of messages sent to 1 specific port
	Stream to all ports	A large number of messages sent to different ports
Random	Random source	Random messages with random source and fixed destination
	Random destination	Random messages with fixed source and random destination
	Fully random	Random messages with random source and destination

Table 1: Routing unit test cases

2) Switch Unit

In the switch unit, we need to verify whether the output ports can deliver messages according to the Round-robin arbitration algorithm. Our switch unit receives messages from three different directions: clockwise, counterclockwise, and port input. The arbitration algorithm tracks the last input port and rotates to the next port in a fixed circular order.

As can be seen in Table 2, our testing strategy includes directed tests and random tests. In directed tests, we start with cases where only one input port has valid messages and then progress to cases where all three ports have valid messages, verifying the basic functionality of the switch unit in transferring messages. Next, by enabling 2–3 input ports with valid data, we examine whether the message transfer order conforms to the Round-robin arbitration algorithm. Once the correctness of the arbitration algorithm is preliminarily confirmed, we further test complex

cases by injecting large numbers of messages from one port, two ports, or all three ports. After all directed tests pass, we use random tests for a more comprehensive test. In the "Random Source" test, all network messages originate from the same input direction, while in the "Fully Random" test, the source, destination, opaque, and payload of network messages are completely random. Finally, we simulate scenarios with delays in the source and sink to verify whether the Round-robin arbitration continues to function correctly under uncertain delays.

Testing Strategy	Test Cases	Introduction
Directed	Simple	A small number of net messages sent from different input ports
	Round-robin	A small number of messages to test the arbitration algorithm
	Stream from one port	A large number of messages sent from 1 specific port
	Stream from two ports	A large number of messages sent from 2 specific ports
	Stream from all ports	A large number of messages sent from 3 specific ports
Random	Random source	Random messages from specific source
	Fully random	Random messages from all sources

Table 2: Switch unit test cases

3) Net Router

After assembling the input queues, routing unit, and switching unit into a complete router, we verify its functionality through successful test outcomes and by observing the line trace. Specifically, we validate the following aspects: (1) The congestion of messages in the queues and whether a new message can be enqueued after a previous message is successfully output. (2) Whether the inputs from the three input ports can be routed to the correct output terminal according to the routing algorithm. (3) Whether the output order aligns with the arbitration algorithm. We use the same test cases for routers with different router IDs, but they produce different results. Table 3 outlines our testing strategy. Similarly, we use both directed tests and random tests, with additional randomized delays applied to these test cases.

It is worth noting that since it is difficult to determine the order in which messages arrive at the stream sink when testing a single router, we chose to ignore the order of arrival in most of our test cases, focusing only on the ability of our network messages to be delivered to the desired output port.

Testing Strategy	Test Cases	Introduction
Directed	Simple	A small number of net messages sent from different input ports
	Same source to same destination	Stream from the same source to the same destination
	different source to same destination	Stream from different sources to the same destination
	different source to different destinations	Stream from different sources to different destinations
Random	Random source	Random messages with random source and fixed sequence destination
	Random destination	Random messages with fixed sequence source and random destination
	Fully random	Random messages with random source and destination

Table 3: Net router unit test cases

4) Ring Net

After assembling the four routers into a ring network, we can reuse the directed and random tests from the net router for validation. At this point, we can clearly observe the movement of network messages within the ring network through the line trace.

Figure 9 demonstrates that the ring network can deliver a message, which would normally require passing through three routers clockwise, to the output terminal in just one cycle via the counterclockwise direction. This confirms the correct implementation of our routing algorithm.

As shown in Figure 10, when all incoming messages from the four input terminals are destined for output terminal 1. In Cycle 2, only the message 1>1:00 can be directly output. In Cycle 3, even though router 1 receives a new message 1>1:01 at its input terminal, the Round-robin algorithm prioritizes the message 0>1:00 arriving from the clockwise direction as the router's output. In the next cycle, the message 2>1:00 from the counterclockwise direction is selected as the output based on the same arbitration algorithm, even though 3>1:00 and 1>1:01 both also

want to output. In Cycle 5, after one complete arbitration round, the priority order resets to: current input, clockwise input, and counterclockwise input. Thus, the message 1>1:01 is output first, and in Cycle 6, the message 3>1:00 from the clockwise direction is selected. This demonstrates that the ring network correctly implements the desired arbitration algorithm.

Cycle	Router 0	Router 1	Router 2	Router 3
1	0>1:11	1>2:12	2>3:13	3>0:10
2	3>0:10	2>3:13	1>2:12	0>1:11

Figure 10: Line trace of test case "Rotate 2"

Cycle	Router 0	Router 1	Router 2	Router 3	Clockwise		Counter-clockwise	Output Terminal 1
					Router 1	Router 4	Router 2	
1	0>1:00	1>1:00	2>1:00	3>1:00				
2	0>1:01	1>1:01	2>1:01	3>1:01	0>1	3>1	2>1	1>1:00
3	0>1:02	1>1:02	2>1:02	3>1:02	3>1	3>1	2>1	0>1:00
4	0>1:03	1>1:03	2>1:03	3>1:03	0>1	3>1	2>1	2>1:00
5	0>1:04	1>1:04	2>1:04	3>1:04	3>1	3>1	2>1	1>1:01
6	0>1:05	1>1:05	2>1:05	3>1:05	0>1	3>1	2>1	3>1:00
7	0>1:06	#	2>1:06	3>1:06	3>1	3>1	#	2>1:01
8	#	#	2>1:07	3>1:07	#	3>1	2>1	1>1:02

Figure 11: Line trace of test case "Stream all to dest1"

In conclusion, after successfully passing all directed, random, and additional delay tests for the route unit, switch unit, router, and ring network, we can confirm that our ring network is functionally correct.

5) Cache Net, Memory Net and Multi-Core Data Cache

We then need to test the cache net, memory net and multi-core data cache. All test cases provided for us have passed.

6) Multi-core System

We first used the tests shown in Table 4 on a single-core system. After completing the basic instructions tests which combine direct and random tests, we performed more complex tests using assembly flows which we designed in Lab 2. These assembly flows not only simulate real-world situations represented by the execution of array operations in C, but also combine all instruction types.

Type	**Instructions**
CSR	csrr, csrw
Reg-Reg	add, mul
Reg-Imm	addi
Memory	lw, sw
Branch	bne
Jump	jal
Assembly	array[i] = array[i]+1
	array[i] = array[i] * constant
	result[i] = array_1[i] * 5+array[2]
	branch and jump together

Table 4: Test cases for single-core system

However, the lab2 test will only let all 4 cores execute the same test, but in a multi-core system, the program executed by the 4 cores may not be the same, so we need to design the test for multi-core system so that the program executed by each core in the same cycle is not exactly the same. So, after we use the single-core tests in a multi-core system, we need to init each core's mngr2proc with different values, so that all 4 cores are running the same instruction but not exactly the same values in the registers. As in the case of single-core systems, we first tested the basic single-instruction types on a multicore system, with both directed and randomized tests. For Reg-Reg and Reg-Imm instruction types, we considered operations with numbers of different sizes, positive and negative. In terms of Branch, we also consider the branches executed after comparison between different values. In terms of the instruction of jump, since it is difficult to let different cores execute the same instruction but jump to different destinations in a multi-core system, we can only let 4 cores execute the same jump and determine whether their positions after the jump are as expected. We have also designed special test cases that stress the memory system, as can be seen in Table 5. Each category focuses on different aspects of the memory system, "Basic Functionality Test" is to verify fundamental operations and correctness. "Cache Feature Test" is to evaluate cache behavior and performance. "Multi-Core Access Mode Test" is to examine interactions between multiple cores. "Memory Access Mode Test" is to analyze various memory access patterns. "Stress Test" is to push the system to its limits under complex scenarios. Finally, we use random tests for a more comprehensive examination, with additional latency tests for the tests described above. All of the tests passed, showing that our multi-core system is fully functional.

Type	Introduction
Basic Functionality Test	Basic load/store operations to verify fundamental memory access
	Tests memory access at boundary conditions
	Verifies each core correctly accesses its dedicated memory region
Cache Feature Test	Tests different cores accessing different words in the same cache line
	Fills up all cache lines in a 2-way set associative cache
	Tests cache capacity miss
	Tests cache conflict miss
Multi-Core Access Mode Test	Tests memory access patterns with contention between cores
	Tests interleaved memory access patterns between cores
	Tests interleaved read/write patterns for memory coherency
	Tests parallel memory operations across all cores
	Tests alternating read/write between cores

	Tests butterfly communication pattern between cores
Memory Access Mode Test	Tests rapid consecutive memory accesses
	Tests memory access with different strides
	Tests sequential access across entire memory space
	Tests memory access with varying stride patterns
	Tests complex patterns of reads and writes
Stress Test	Tests using multiple registers for concurrent memory operations
Random Tests	Tests random memory access patterns to stress test the system

Table 5: Test cases for multi-core system

In our multi-core system, when we are testing assembly flows by reusing the tests in single-core system, all of them passed except for the assembly test "array[i] = array[i] * constant". This specific test failed because, in a multi-core system, when two cores load, modify, and store different values to the same address, the final result becomes unpredictable. Such a "race condition" can occur in the array self-multiplication test in a multi-core system, which makes this type of test unsuitable for multi-core systems.

B. Software

ubmark_sort()	mtbmark_sort()
1. White-box testing 2. Black-box testing 3. partition() testing 4. swap() testing	1. White-box testing 2. Black-box testing 3. merge() testing 4. mergeBlocks() testing

Table 6: Test Cases for Algorithms

Testing the algorithms for the single-core and multi-core processors were relatively similar. We used directed testing, along with black-box and white-box testing, for both. We implemented quickSort as the basic algorithm for our baseline design while we integrated merge() (from mergeSort()) to create a parallel sorting algorithm for our alternative design.

Black-box testing was crucial for ensuring our entire algorithm, including helper functions, functioned properly and delivered the expected results whether run on our baseline or alternative design (single-threaded quickSort algorithm only). White-box testing verified the functionality of helper functions, such as swap(), partition(), and merge(), especially in the context of our multi-core processor.

Furthermore, we utilized the provided ece4750 library to create randomized testing. These functions allowed us to stress test our algorithm by passing values without any patterning. We also created various combinations of values, such as arrays with only negative numbers, mixed numbers, odd number size, etc. These conditions ensured that quickSort actually identified the correct pivot and partitioned the array appropriately. Furthermore, for the multithreaded algorithm, we tested merge() and mergeBlocks() to ensure that it was able to combine blocks to produce the correct result.

IV. Evaluation

Area Analysis

In terms of area, our multi-core system requires more hardware area than the single-core system. First, there is an increase in the number of basic components: we use 4 processors and 8 two-way set-associative caches (including 4 instruction caches and 4 data caches), which directly quadruples the hardware usage compared to the single-core system. Meanwhile, to enable these processors to work properly, we need three ring networks to coordinate communication between them: two memory networks and one cache network. Each network is equipped with 4 upstream adapters and 4 downstream adapters for data format conversion, and has separate request and response networks.

In these networks, each has 4 routers working, with each router's hardware overhead mainly coming from three parts: 3 4-entry FIFO input queues implemented with SRAM for data storage; 3 routing units requiring combinational circuits for address comparison and direction selection; and 3 switching units needing multiplexers and state registers for data selection. Based on the complexity estimation of each component, a single router takes up about 3-4% of a processor's area, so the 4 routers in each network occupy about 12-16% of the processor area. Adding the small area overhead of adapters, each network takes up about 13-17% of the processor area. Considering that we have three such networks, the network portion totals about 40-50% of a single processor's area, representing an increase of about 0.4-0.5 times the area. Therefore, our multi-core system uses approximately 4.4-4.5 times the area of a single-core system.

Energy Analysis

The power consumption analysis of the multi-core system is similar to the area analysis, as our multi-core system requires more power than the single-core system. For basic components, the power consumption has increased by 4 times due to the processors and caches, which is obvious. Additionally, to support inter-core communication, the three ring networks generate extra power overhead in each cycle. Specifically, each of the 4 routers in each network requires power for: SRAM read/write operations in input queues; optimal path calculation in routing units each cycle; and continuous data selection and forwarding in switching units.

Although the power consumption of a single router is relatively small, considering that there are 12 routers working simultaneously in the system, plus the power consumed during data transmission through the network, the network portion adds approximately 0.2 times the power consumption of a single-core system. It's worth noting that while the network portion takes up about 0.4-0.5 times extra area, its power increase ratio is smaller than its area increase ratio. This is because the circuits in the network portion are not active in every cycle, with many components only consuming dynamic power when data transmission is needed. Therefore, our multi-core system consumes approximately 4.2 times the power of a single-core system.

As demonstrated by the microbenchmark results, the multi-core processor excels at improving performance for compute tasks which can make up for the tradeoff in area and energy. Synchronization overhead is a key reason why energy consumption may be higher, but due to the performance increase, programs may finish quicker, resulting in less overall energy consumption in the long run.

Performance Analysis

The CPI for the multi-core processor is overall significantly lower than that of the single-core processor. However, an interesting trend is that the multi-core processor that only initialized one worker to run the multithreaded microbenchmark performed better than the multi-worker setup. This can be explained by the fact that initializing multiple threads requires more overhead and experiences more memory contention.

Our sorting microbenchmark did not demonstrate a significant improvement in total cycles between designs, but the cache miss rate improved regardless. Since we chose to implement quickSort() in our single-threaded algorithm and utilize it in our multithreaded algorithm design, the actual computation time depends on how complex the workload is. The multithreaded algorithm's evaluation utilizes significantly larger datasets that are sorted quicker when the array is split. The same algorithm designed for a single-core system takes more cycles on a multi-core system than on a single-core system due to the fact that a multi-core system has more memory networks than a single-core system and it takes a certain number of cycles for information to turn over in the network.

Additionally, the multi-core system sometimes experiences higher cache miss rates, such as in mfilt, whereas other microbenchmarks demonstrate a marked improvement. Occasionally, the system experiences a tradeoff with parallelism and efficient memory accesses. The multi-core processor seems to provide the greatest performance benefit with computing-heavy tasks, such as vvadd and cmult. Regardless, the overall execution time of each microbenchmark is significantly enhanced when using the multi-core processor, further proving the performance benefit from parallelism.

Microbenchmark	ST-SC	ST-MC	MT-SC	MT-MC(SW)	MT-MC
Sort	39923	48219	54141	38525	38582
Vector-Vector Add	4267	5799	5284	7054	2710
Masked Convolution	29093	34697	28552	34347	12853
Complex Multiplication	13673	17647	14670	18996	5728
Binary Search	10705	12068	11858	13571	4408

Table 7: Evaluation results of execution time in cycles for five microbenchmarks

microbenchmark	ST-SC	ST-MC	MT-SC	MT-MC(SW)	MT-MC
Sort	4.93	1.29	5.03	1.40	1.40
Vector-Vector Add	5.25	1.37	5.43	1.37	1.65

Masked Convolution	4.66	1.31	4.75	1.31	1.45
Complex Multiplication	7.55	1.42	7.44	1.42	2.19
Binary Search	5.04	1.32	5.14	1.29	1.44

Table 8: Evaluation results of CPI for five microbenchmarks

microbenchmark	ST-SC	ST-MC	MT-SC	MT-MC(SW)	MT-MC
Sort	0.05	0.02	0.11	0.05	0.05
Vector-Vector Add	0.25	0.25	0.28	0.24	0.21
Masked Convolution	0.17	0.18	0.18	0.19	0.23
Complex Multiplication	0.19	0.19	0.20	0.19	0.18
Binary Search	0.57	0.41	0.53	0.40	0.31

Table 9: Evaluation results of data cache miss rate for five microbenchmarks

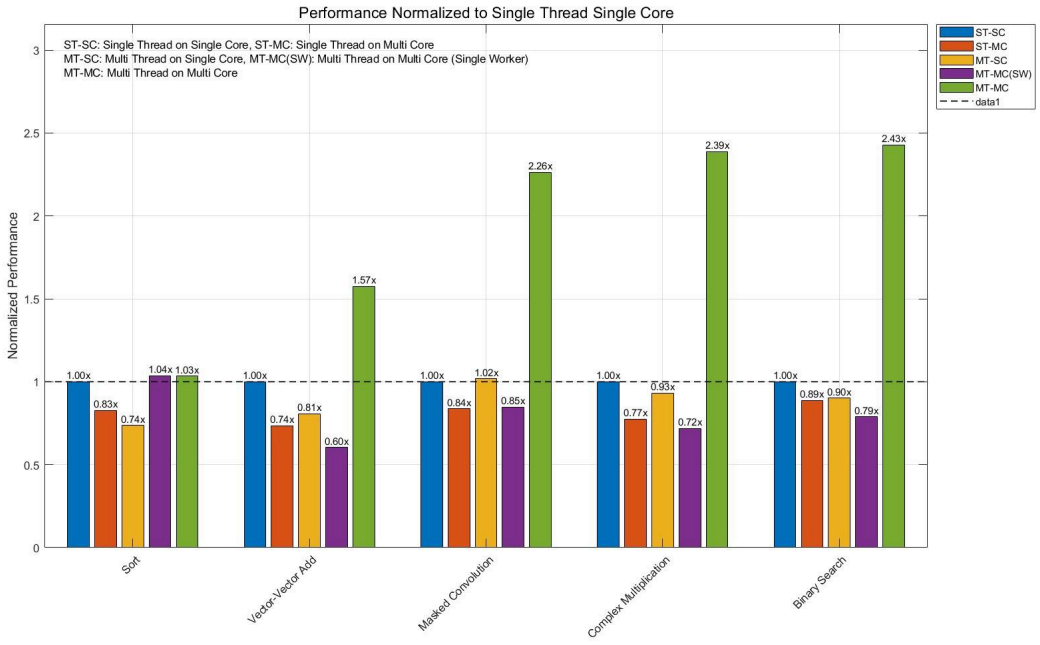


Figure 12: Performance of five benchmarks normalized to single thread single core

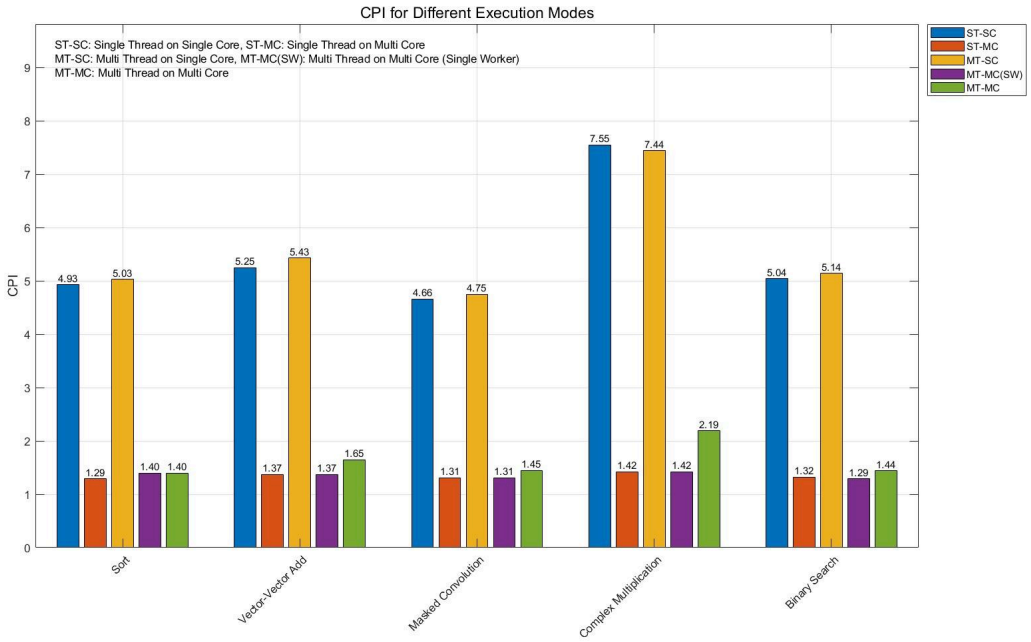


Figure 13: CPI for different execution modes

V. Conclusion

Based on our evaluation, we conclude that the multi-core processor design significantly enhances performance through the inclusion of parallelism. While a multicore system requires about 4.5 times the area and 4.2 times the energy, the multicore system is about 2 times faster than a single core for the same program. And although the increase in speed isn't perfectly proportional, the trade-off is justified in scenarios where faster processing and reduced latency are critical. In the current era, the demand for processor performance is increasing. In high-performance computing or real-time response scenarios, the speed gains from multi-core systems translate into higher task parallelism, lower latency, and significant efficiency benefits. This explains why, while there is still a place for single-core processors, the increasing demand for performance has led to a move towards parallel processing. At the same time, however, we need to pay attention to the design algorithms. Optimizing algorithms, such as increasing parallelism, can close the gap between area, energy consumption, and performance, improving the overall cost-effectiveness of the system, while algorithms that lack parallelism are likely to show slightly less performance than a single core in a multicore system. To conclude, in an era of increasingly demanding computing power, we should use multi-core systems and optimize our algorithms for multi-core systems.

VI. Work Distribution

Shencheng Xu focused primarily on implementing the hardware while Asma Ansari focused on developing the sorting algorithms. Both worked on test cases for each respective area and worked on the lab report.