Anne Potter (ap674) and Ashley Heckman (agh93)
Lab Report 4
ECE 4750
12/6/22

Lab Report 4

1. Introduction

In this lab, we implemented a single-core and multicore system. The single-core system is made up of an instruction cache, data cache, and processor. The multicore system is made up of four private instruction caches, a four-banked shared data cache, four processors, and four networks. We used the multiplier, processor and cache we created throughout the semester to implement both systems, and we implemented a simple ring network for the multicore system in this lab. Both systems can run very simple programs, including a sorting algorithm that we implemented. Our baseline design was a combination of the single-core hardware system and the single-core sorting algorithm that we implemented. The alternative design uses the multi-core hardware with four worker threads and the multi-core system software to create a multi-threaded sorting microbenchmark. This lab is intended to give us experience with creating simpler subsystems to create more complex systems, programming single and multi-threaded C programs, incremental and test-driven development, and architecture-level statistics. It is also the culmination of all the other projects we have completed this semester.

2. Alternative Design

We will begin this section with a discussion of our software implementations. We decided to use quicksort for our sorting algorithm. We did this because it is one of the faster sorting algorithms, as seen in *Figure 1* below.

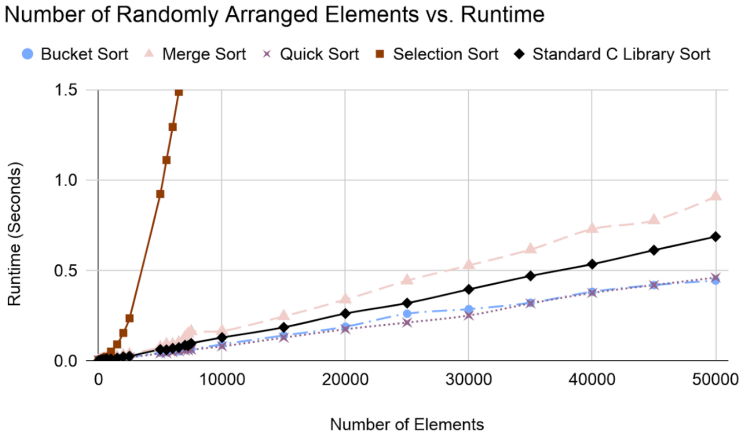

Figure 1: Speed of Different Sorting Algorithms

Quicksort is a recursive algorithm where a pivot point is chosen and the vector is reorganized, moving integers less than the selected pivot to the left, and integers greater than the selected pivot to the right. If the length of the new vector is greater than one, a new pivot is selected and the pattern repeats. The last element of the vector is used as the pivot every time.

Quick sort relies on a helper function called partition. Partition starts at the beginning of the array and compares every value in the array to the pivot. If the value is less than the pivot, it is placed in the new left array, and if it is greater than the pivot, it is placed in the new right array. The new, smaller arrays are once again partitioned, using the element in the final index of the new arrays as the pivot. The pattern is complete when the length of each array is one or zero. This is shown in *Figure 2* below. The original array is reassembled in sorted order by adding single elements at a time on the correct side of the pivot that was used to split the array. This recombination is shown in *Figure 3* below.

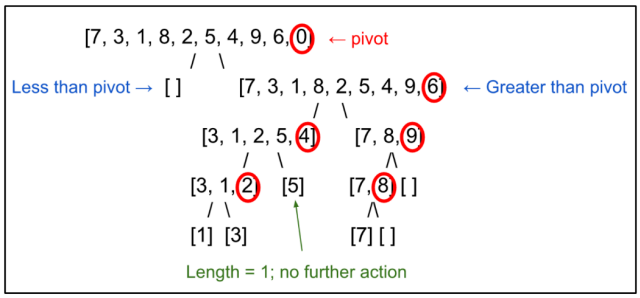

Figure 2: Partitioning of the Array

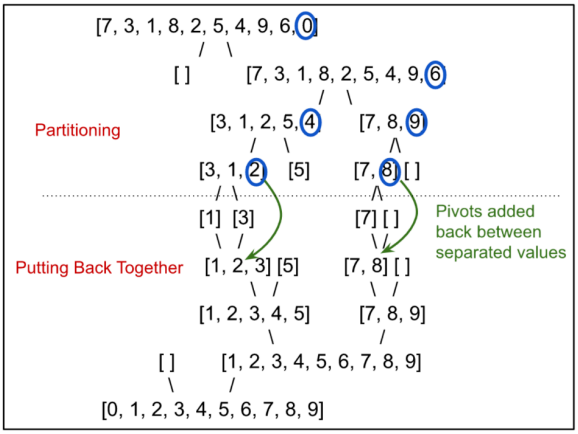

Figure 3: Recombination of the Array

Our alternative, multithreaded design also uses quicksort. However, to make use of all four threads, the data is split into four approximately equal sections, so that the load can be divided amongst the four cores. Each core uses quicksort to sort their section of the array, and the four small arrays are rejoined using three calls to merge. Merge starts with an empty array. It then iterates through two sorted vectors, and finds the array with the smallest value. This value is added to the end of the array. This process is shown in *Figure 4* below. The alternative sorting algorithm sends each (approximate) quarter of the vector to each core and then waits for the sorted segments to return. The data from cores 0 and 1 are combined using merge, then cores 2 and 3 are combined, and finally the two resulting combined vectors are combined one final time to create the original array in its sorted order.

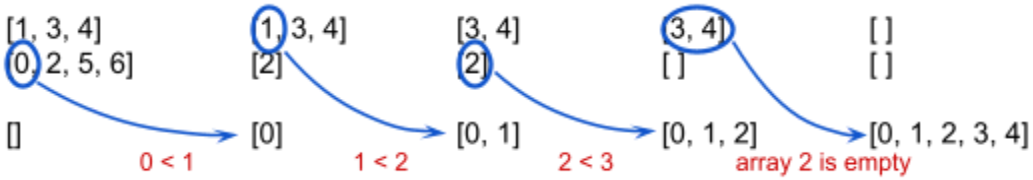

Figure 4: Merge Combining 2 Arrays

We now move onto the discussion of the hardware implementation. The baseline design is a simple composition of a processor, an instruction cache, and a data cache. The alternative design is composed of four processors, four independent instruction caches, a four-banked shared data cache, a memory network, and a cache network. The processors and caches are from the alternative design of labs two and three, respectively.

The networks used for the caches and memory are simple ring networks. The network has four input terminals, four output terminals, and four routers. Each router is made up of three route units and three switch units.

The route unit checks to see if the router id matches the destination of the request. If it does, this is where we want to output the response, so it keeps the data in the same spot. If it doesn't, it sends the data clockwise around the ring network to a different router. We decided to only route data clockwise to try to avoid backups at one router. If it wants to keep the data in the same spot, it sends the data to switch unit 0. Otherwise, it sends the data to switch unit 1. Note that this algorithm does not use switch unit 2.

The switch unit gives priority to messages that are already in the network to help avoid deadlock. If the router received a message from a different router, the switch unit will output that message. Otherwise, if there was no message received from a different router, the switch unit will output the new message coming into the network if there is one. Switch unit 0 is hooked up to the output terminal of the network, and switch unit 1 is hooked up to the clockwise input of the route unit to the right of it.

When implementing our designs, we incorporated the patterns and principles of modularity, hierarchy, and encapsulation. Our designs exhibit modularity by decomposing the design into different modules. Each main component is self-contained in its own module, highlighting what code belongs to what component.

Our designs show hierarchy through the decomposition into smaller modules. The MultiCoreSys.v file is at the highest level, and it ties everything together. Just below that are the processor, cache, and network files. Below those are files that these units use. Through tracing which module calls which other modules, it's very easy to see what the high level and low level files and blocks are in the hierarchy.

The designs exhibit encapsulation by hiding implementation details from the interface. One such detail is the system latency. This is hidden using the val/rdy signals. Another detail is which core each part of the program is running on. The user has no reason to know which cores are actively being used; all they care about is having their programs function properly. Thus, we do not reveal these details.

3. Testing Strategy

We were given or we had already made a large portion of the tests for this lab. Although we already had many of the tests, we still took an incremental approach to our implementation and testing strategy. We added directed tests to stress the function of the different components (and the lab as a whole) and random tests to increase code coverage. We also implemented delay tests to ensure that the components functioned properly in our latency-insensitive stream interface. We implemented one component at a time, then thoroughly tested that component to ensure correct functionality before moving on to the next.

We began testing our code by making sure that all components from the previous labs were still functioning properly, and that all revisions had been copied over to the main branch. This was a bit of a sanity check and ensured that the components that we were using from previous labs were working properly.

We then moved onto unit testing the new components that we built for this lab. Unit testing tests specific parts/modules of the design; it doesn't test the design as a whole. This allows us to check that the smaller parts of our design are working properly, which is critical for ensuring that the larger design works. In this lab, we implemented eight things that we were able to unit test: the single-threaded sorting algorithm, the multithreaded sorting algorithm, the router switch unit, the router route unit, the router as a whole, the network, the cache network, and the memory network. We tested each of these components with a mixture of directed and random test cases, and most of them with delay tests. For each component, we made sure all lines of code were covered with the different test cases. Once we were confident that these units were working, we moved onto testing the next unit, and eventually the system as a whole.

We used directed testing to test specific scenarios and ensure basic functionality. This allows us to both build really simple test cases and to test specific corner cases that we think might cause an issue in our smaller components and in our system as a whole. We used random testing to increase code coverage and ensure that we weren't only testing scenarios that we knew would pass. This combination of increasing code coverage and testing specific corner cases drastically increases the odds of us discovering a problem if one exists.

We also used delay tests. These tests make sure that the design can handle arbitrary source and sink delays. This ensures that if some other component needs to stall or if there's a component that isn't ready to send/receive data, the other components of the system don't break. For example, executing a multiply instruction takes multiple cycles. When this happens, the processor takes longer than usual to complete, which likely causes components ahead and behind it to need to stall. We add delay tests to test situations like these to ensure that the data is still handled properly in the event that data is not yet ready or something further along is not yet ready to accept the data that a component produced. A summary of what kind of situations we tested is shown below in *Table 1*.

Type	Category	Justification/Description
Sanity	Previous Labs	Ensure that all code from lab 1, lab 2, and lab 3 is passing all of the RTL tests. Since we use all of these components in this lab, they must function properly. This contains a mix of directed, random, and delay tests.
Unit	Single-Thread Sort	Ensure that the sorting algorithm that utilizes a single thread is functioning properly. We include a mixture of directed and random testing for this.

	Multi-Thread Sort	Ensure that the sorting algorithm that utilizes 4 threads is functioning properly. We include a mixture of directed and random testing for this.
	Switch Unit	Ensure that the switch unit (which is part of the router) is working properly. We test this with directed test cases and some source and sink delays. We make sure that every combination of source and destination is tested.
	Route Unit	Ensure that the route unit (which is part of the router) is working properly. We test this with directed test cases and some source and sink delays. We make sure that every combination of source and destination is tested.
	Router	Ensure that the network router is working properly. We test this with directed and random test cases, and with source and sink delays. We make sure that every combination of source and destination is tested.
	Network	Ensure that the network is functioning properly. We use a mix of directed, random, and delay tests. We make sure that every combination of source and destination is tested.
	Cachenet	Thoroughly test the request/response network pair for interconnecting four processors and four cache banks. Contains a mix of directed, random, and delay tests.
	Memnet	Thoroughly test the request/response network pair for interconnecting four processors and the main memory port. Contains a mix of directed, random, and delay tests.
Directed, Random, Delay	Single-Core System	Bring in assembly tests to ensure instructions are functioning properly. Ensure that this system passes all software tests. Includes a mix of directed, random, and delay tests.
Directed, Random, Delay	Multi-Core System	Bring in assembly tests to ensure instructions are functioning properly. Ensure that this system passes all software tests. Includes a mix of directed, random, and delay tests.

Table 1: Test Cases

4. Evaluation

The alternative design has more components and more logic than the baseline design. It has 3 extra processors, 3 extra instruction caches, 3 extra data caches, and 4 networks (2 for requests and 2 for responses). This obviously requires a lot more logic to connect everything together as well. All of these extra components and logic require more area. Since the alternative design has more stuff, it will take up more space than the baseline design does.

More components and more logic generally means a longer cycle time, as the data needs to travel a longer distance and through more components. The critical path of the alternative design goes through more logic and components, so the alternative design will have a longer cycle time than the baseline design.

Powering more components and logic will also take more energy per cycle. In the baseline design, only one processor, instruction cache, and data cache are used. In the alternative design, we use four times the amount of those components, and some extra additional components. These components all require energy to operate. However, even though the alternative design will take more energy per cycle, it generally takes fewer cycles to complete, so it will require less overall energy on average.

We can compare the performance between the alternative and baseline designs in terms of the speedup and the CPI (cycles per instruction) for different programs. We can also see the hardware and software overheads of moving to the multithreaded, multicore design through running the different evaluations on different microbenchmarks. Running the single-threaded program on the multicore system shows the hardware overhead, running the multi-threaded program on the single-core system shows the software overhead, and running the

multi-threaded program on the multicore system with only 1 worker thread shows the combined software and hardware overhead.

All functions perform better on the alternative design than the baseline design. There is a small software overhead associated with running the multi-threaded program, and a slightly bigger hardware overhead associated with running things on the multicore system. These overheads combine into a noticeable increase in the number of cycles required to run each program if we only use 1 worker thread, showcasing the combined hardware and software overheads. However, the full alternative design that uses a multi-threaded program on a multicore system with 4 worker threads is able to overcome these overhead costs and perform better than the baseline design. This makes sense, as the software overhead is due to spawning more threads, and this only happens once in the beginning of the program. The hardware overhead occurs every time we access the caches (due to things needing to travel through the networks), but the overall work for each processor is cut into a quarter of what it is in the baseline design, and this is enough to overcome that overhead.

For example, the sort program has a 0.83 times speedup due to the software overhead, a 0.74 times speedup due to the hardware overhead, and a 0.61 times speedup due to the combined hardware and software overheads. However, the alternative design is able to overcome these overheads, as it achieves a 1.16 times speedup compared to the baseline design. This is shown in *Figure 5* for all 5 experiments.

The CPI in *Figure 6* counts the cycles per instruction on core 0. Because the single-threaded programs only use one core, it computes the CPI of the entire program. However, the multithreaded programs use 4 cores total, so this only captures the CPI for roughly one quarter of the work for the multithreaded evaluations. Thus, the CPI is more a comparison of efficiency than anything else.

The multithreaded program running on the multicore system with 4 workers has a higher CPI than the baseline design. Much of this comes from the hardware overhead associated with the multicore system, as seen when looking at the CPI for the single-threaded program running on the multicore system. This makes sense, as the data has to travel through multiple networks to reach the same location, which will obviously add to the number of required cycles. The software overhead associated with using a multithreaded program is almost nonexistent, as seen when looking at the CPI for the multithreaded program on the single-core system. This makes sense, as spawning and combining threads only happens once and should take almost no cycles. The combined overhead of the hardware and software on CPI is really just the overhead due to the hardware.

For example, the sort program has a CPI that is 1.36 times greater than the baseline due to the hardware overhead, the same CPI as the baseline due to the software overhead, and a 1.36 times CPI due to the combined hardware and software overheads. The CPI of the alternative design is almost identical to this, at 1.34 times the CPI of the baseline design.

While core 0 is less efficient in the multicore system, the multicore system with the 4 worker threads still uses fewer total cycles than the baseline design. This is because even though each core is less efficient, there are more cores, so each core does less overall work. The difference in the amount of work is enough to overcome the inefficiencies of each core in the multicore system. Thus, we should use the multithreaded programs on the multicore system with 4 worker threads going forward.

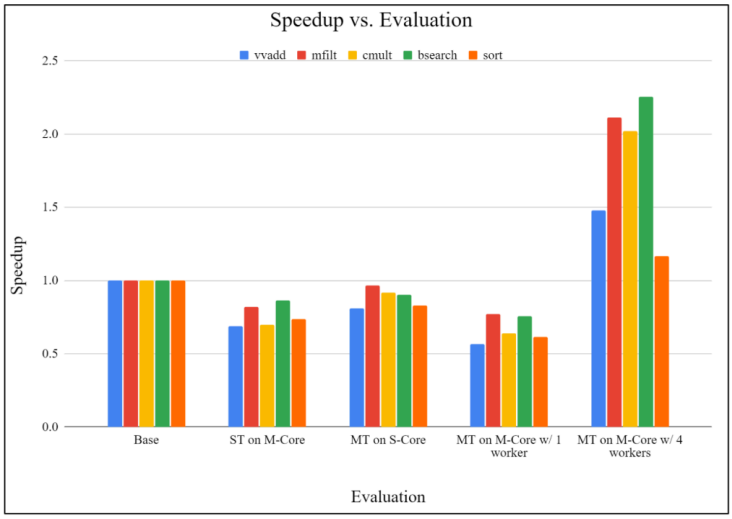

Figure 5: Speedup vs. Evaluation

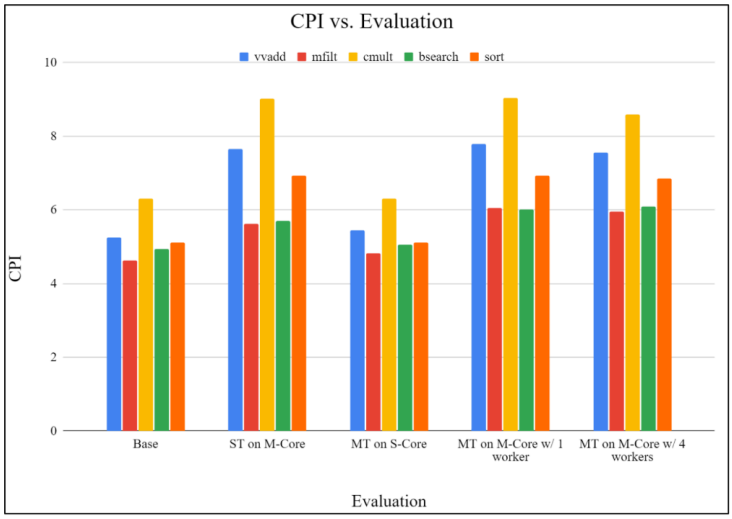

Figure 6: CPI vs. Evaluation

5. Conclusion

Our alternative design performed better than our baseline design for all five experiments. The alternative design cut the amount of work required by each processor into quarters. All of the extra components in the alternative design take up more area and cause a longer cycle time, but the total energy consumption is lower for the alternative design since it takes fewer cycles to complete on average. As seen in *Figure 5*, there is overhead associated with using the alternative design. Running the single-threaded program on the multicore system is slower than the baseline design due to the hardware overhead of every load and store instruction needing to travel across the network. Running the multi-threaded program on the single-core system is slower than the baseline design because of the software overhead of spawning the extra threads. Running the multi-threaded program on the multicore system with only 1 worker has the combined hardware and software overheads, and is the slowest out of all microbenchmarks. However, when everything is put together, running the multi-threaded program on the multicore system with 4 workers is faster than the baseline design. For example, for the vvadd evaluation, there is a 0.69 times speedup for the single-threaded program on the multicore system, there is a 0.81 times speedup for the multi-threaded program on the single-core system, and there is a 0.57 times speedup for the multi-threaded program on the multicore system with only 1 worker when compared to the base design. These are all slower due to the overheads associated with multithreading and multi-coring. When everything is put together, however, there is a 1.48 times speedup for the multi-threaded program on the multicore system with 4 workers when compared to the baseline design. This general trend was the case for all evaluation patterns. The CPI for the multicore system was higher than that of the baseline system, but since the total number of cycles was smaller for the multi-threaded program running on the multicore system with 4 worker threads, we should use that design going forward.

6. Work Distribution

Ashley implemented the hardware for the single-core and multi-core systems. Annie implemented the software for the single-core and multi-core systems. We both worked on the test cases and we wrote the report together.