Zephan Sanghani, zrs29
Anthony Song, abs343

December 9, 2025

Lab 4 Report: Single-Core and Multi-Core Systems

1 Introduction

In Lab 4, we investigate and evaluate two designs: a single-core system and a multi-core system with private instruction caches and shared data caches connected via a ring network. We also investigate and evaluate software (algorithm) design for both hardware systems. Our baseline is a single 5-stage pipelined processor connected with instruction and data caches and a corresponding single-threaded sorting algorithm. The alternative design instead instantiates 4 pipelined processors, utilizing private instruction caches and a *banked data cache* system via a ring network with a corresponding multi-threaded sorting algorithm. This assignment connects deeply with a fundamental theme in computer architecture: the multi-core era. Per the end of *Dennard Scaling* around 2005 (fundamental limitations of single-core processors), computer architects looked to *exploiting parallelism* to improve performance. In this lab, we aim to quantify both the benefits and drawbacks of such multi-core designs.

2 Alternative Design

Our alternative design is an extension of our single-core system into a quad-core system. As such, we duplicate all four processors, instruction caches, and data caches. However, there are some challenges with such an implementation if the caches are *private* to each core.

Specifically, concerning the data caches. Private *instruction* caches are okay, as processors never *write* their instruction addresses — under the assumption that there is no self-modifying code. Also, this improves our *exploitation* of *spatial* and *temporal locality*.

However, with *private* data caches, we will have *coherency issues*. The data caches could hold different results for the same addresses. To solve this, we assign unique addresses to each data cache (via *banking*). As such, the processors may need to issue memory requests to *any of the four caches*, so we hook them up to the four processors via a *cache network*.

The resulting block diagram is seen in Figure 1. There are four pipelined processors (denoted by 'P'), and four instruction ('I$') and data ('D$') caches.

They are connected via 3 networks. Each of these are "bi-directional" (request and response networks). A memory request is routed to the proper *banked cache* via specific "bank bits" (in our case, 4 and 5). The response is routed to the requesting processor.

There are also two memory networks which are responsible for connecting the instruction and data caches with main memory (for refills and evictions).

For our three networks, we implement them with a 1-dimensional torus topology (a "ring network"). A block diagram for this network is seen in Figure 2. It consists of four input and output terminals, connected by 4 routers with nearest-neighbor connections.

The terminals and routers are connected via *latency insensitive val/rdy stream interfaces.* More

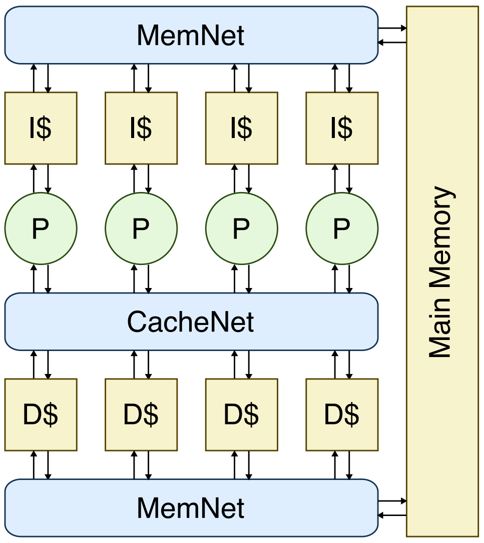

Figure 1: Alternative Design

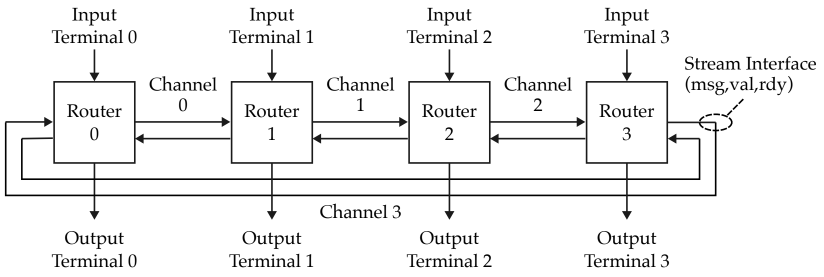

Figure 2: Ring Network Topology

specifically, each channel/terminal is gated with a 4-deep FIFO (first-in, first-out) queue to mitigate back-pressure and prevent deadlocks. The inputs to the routers are *network packets* — consisting of a source/destination (terminal) address, opaque field, and a payload.

On each cycle, each router takes some of its inputs (input terminal and neighbors) and directs them to either its output channel, or one its neighbors. To do this, we construct a router out of smaller submodules: three route units and three switch units. See Figure 3.

The three input streams correspond to the input terminal and the two neighbors. Similarly, the output streams correspond to the output terminal and the two neighbors.

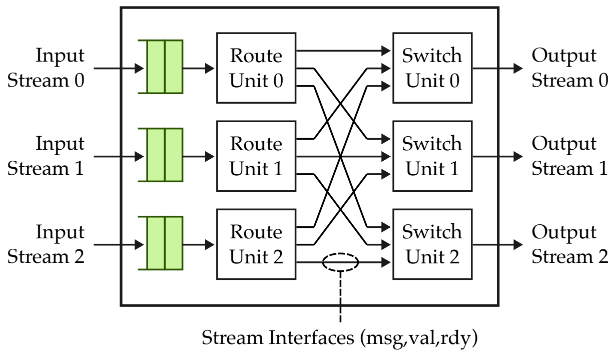

Figure 3: Router Block Diagram

The route unit determines the correct output to send packets to. Our implementation of the route unit routes each packet such that it achieves the shortest possible path.

In the case that the destination is equidistant in both directions (2 hops away), we route clockwise for *even-indexed* routers and counter-clockwise for *odd-indexed* routers. The goal of this is to prevent saturating the network in any direction.

It's possible that multiple inputs route to the same output; thus, we must arbitrate. For this, we instantiate *switch units* for each output — linked to route units via val/rdy streams. Each cycle, a switch unit selects one of the routed packets to send to the output stream.

Our implementation of the switch unit utilizes *round-robin arbitration*. The *valid signals* from the route units are used as the requests, such that *only* a valid packet will be selected. If a route unit was selected, it will be given last priority on the next cycle to ensure fairness.

Composing our route units, switch units, and 3 4-input FIFOs completes our router. Instantiating and linking up four of these routers (like Figure 2) creates our ring network. Instantiating two rings (one for request/response each) creates our memory/cache networks. Lastly, we arrange these with four of our pipelined processors and FSM caches like Figure 1.

We also needed to implement a multi-threaded benchmark for our multi-core system. First, it's helpful to discuss the single-threaded benchmark for our single-core system. For this, we implemented a quicksort algorithm (repeated recursive partitioning of an array).

We chose to use this algorithm as a subroutine for our multi-threaded algorithm. Our multi-threaded algorithm spawns an instances of this subroutine on each core. That is, each core sorts one subarray ($\approx n/4$ elements). Then, core 0 merges all subarrays with mergesort.

This quickly sorts the array (with some memory overhead). This is optimized further later.

We use *modularity* when instantiating *identical* routers, networks, processors, and caches. We use *hierarchy* when composing these (and their submodules) together. Then, we utilize *regularity* by using consistent (typically *latency insensitive*) interfaces across these modules.

3 Testing Strategy

In this lab we are required to test both our hardware (the ring network) as well as our software (single/multi-threaded sorting algorithms). To do this, we applied *several* testing strategies such as directed testing, random testing, delay testing, and test reusability. We utilize *directed* tests for *specific* edge cases, *random* tests (for broad coverage of "*unknown unknowns*"), and *delay* tests to test our modules performance *under backpressure*.

First, we test the core building block, the router, and its constituent modules: the route and switch units. We utilize *white-box testing* to test the *correctness* of our algorithms (the *specific* algorithm defines its behavior). We may only *black-box* test the *entire* network together. We test our shortest-path route unit with directed (e.g. all sources/destinations) and random test cases. We test each router id to ensure correctness across all four routers.

Next, we apply a similar testing strategy for our switch unit. We test our round-robin arbitration with various directed (i.e. all sources/destinations), random, and delay test cases.

Then, we test the composition of these submodules: the router. We implemented directed tests that test for common routing patterns (i.e. rotation, all-to-one, etc) as well as random tests for edge cases. We also test with random src/sink delays to validate backpressure.

We continue to raise the level of abstraction as we test the *entire* ring network. For this, we carefully implement a wide-range of directed tests for robustness. Some of these include rotation, all-to-one, and tornado (the worst traffic pattern for a torus topology). We supplement these tests with random tests, also including random source/sink delays.

Next, we test both the memory and cache networks (which utilize our ring network). For these we include a variety of directed tests which test common patterns for these networks. Some tests also used random source/sink delays to validate the *val/rdy stream interface*.

Next, we test the multi-core data cache — the caches linked up to the cache networks. This allows us to independently verify the functionality of composing these modules before integrating further components like the processors. We use directed tests to test common cache transactions (read/write hit/miss) on different banked caches, utilizing the network. We also include random testing with randomized source/sink delays to validate backpressure.

Once these comprehensive tests are verified, we can move on to testing our systems. For the single-core system, we select and *re-use* tests from Lab 2 to verify its functionality. Specifically, we test *add*, *mul*, *addi*, *lw*, *sw*, *bne*, and *jal*. This allows us to test register-register, multiply, register-immediate, branch, and jump instructions. More importantly, we can test our *memory instructions* (sw/lw) to verify our data cache integration. We utilize all tests for these instructions, including directed, random, and (random) delay tests.

Next, we approach testing the *multi-core* system in a similar manner. We *re-use* the same tests from Lab 2 as the single-core-system. That is, we again test *add*, *mul*, *addi*, *lw*, *sw*, *bne*, and *jal* with directed, random, and delay tests. Additionally, we implement *specialized* tests for the *multi-core* system. First, we develop directed *multi-core-specific csr* tests. Then, we develop *multi-core-specific memory* tests. These test behaviors specific to our quad-core system, such as read/writing all banks simultaneously, read/writing the same line from different processors, etc. These consist of directed, random, and delay tests.

Overall, we validate our single-core and multi-core systems with 489 tests. These supplement the 96 tests for our two-way *set associative* cache, the 265 tests for our *bypassed* pipelined processor, and the 53 tests for our iterative multiplier (903 total tests)!

It's also worth noting that we *re-used* the test cases for both our two-way *set-associative* cache and our *bypassed* pipelined processor when *we optimized the implementations.*

In addition to testing our hardware, we must also test our software for this lab. To do this, we take a modular testing approach reminiscent of the testing we did for our hardware. That is, we test our helper functions, and then test the functions (sorting algorithms) that compose them to verify correctness. For our single-threaded sorting algorithm (*quicksort*), we first validated the *partition* subroutine with several directed tests.

Once that was verified, we validated the entire sorting routine. We test various cases, such as: different array sizes, already-sorted arrays, arrays with duplicates, etc. We additionally sort both arrays allocated on the stack and on the heap (to ensure robustness to allocation). We first test the code running natively on processor. Then, we cross-compile it for Tiny-RV2 and test it on the FL model. Finally, we test it on our actual *single-core system.*

We follow a similar strategy for testing the multi-threaded benchmark. First, we test our *merge* subroutine. Then, we compose the *already tested* quicksort subroutine with this merge subroutine to construct our multi-threaded benchmark. Finally, we *re-use* the sorting test cases from the single-threaded benchmark testing to verify the multi-threaded benchmark.

4 Optimizations

Hardware Optimizations

We have three major components in our single-core and multi-core systems: the 5-stage bypassed pipelined processor, the two-way set-associative FSM cache, and the ring network. The ring network is already *quite optimal*, so we focus our efforts on the other components.

First, we begin with optimizing the cache. We start here since there is the most to gain. Our FSM cache implementation had a 3-cycle hit latency with 4-cycle sustained throughput. This gives us a *fundamental limit* of 4 cycles per instruction (bottlenecked by the instruction cache). We seek to improve this. First, we focus on achieving a 1-cycle *read hit* latency.

To do this, *we merge* the tag check, read data access, and done states *into one.* This translates to several modifications in our cache's datapath. First, we must divide the *data array* into two SRAMs in order to speculatively read both ways at once (in tag check).

Next, we must bypass the read data register to send the read data immediately. This is implemented by the "read around data mux" in our datapath (see Figure 4). Now, in tag check, we speculatively read both ways, multiplexing the correct data (if it read hits). Then, if we have a read hit, we can immediately send the data out on the cache response.

If the cache response is *not received* by the requester (`cacheresp_rdy`), we transition to the done state and wait. Otherwise, we can *skip the* done state, and go directly back to idle. To optimize *further*, if there is another request (`cachereq_val`), we can go *back to tag check.* Together, this gives us 1-cycle *sustained read hit latency* (we can just remain in tag check).

We implement similar optimizations for other states (read miss, write, init). That is, we send the cache response as soon as we are able to (and skip done if possible). Additionally,

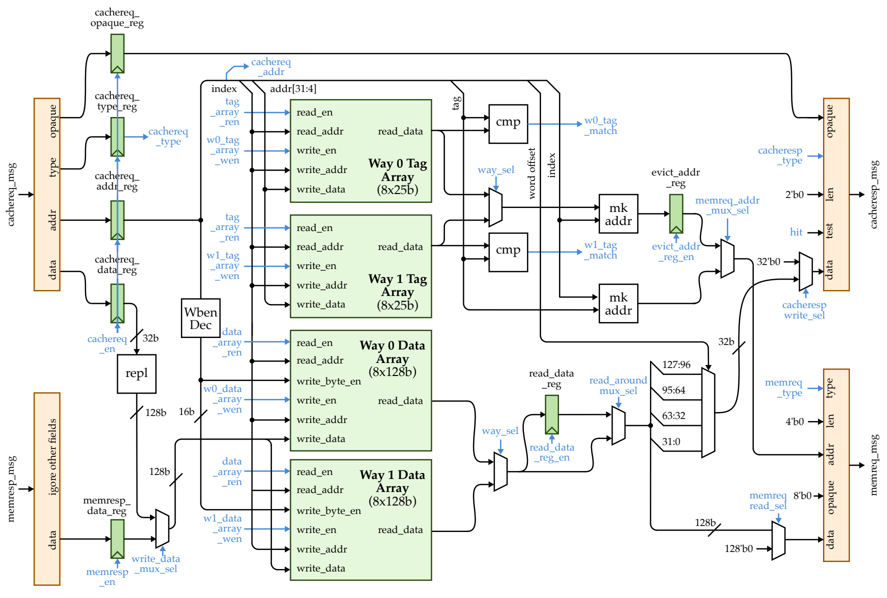

Figure 4: Optimized Cache Datapath

if we are done and there is another cache request, we transition directly back to tag check.

We also optimized the write response: implementing a “early write acknowledgment”. For any write transaction, we immediately send the acknowledgment response. Although the cache must still write the data, it allows the processor to continue on with execution. As such, we cannot maintain a high write throughput, but our write latency *is quite fast*.

To implement this, we add a multiplexer before the cache response, in which we can multiplex in all-zeros (the write response). This replaces the previous read data zero mux. We must also add a “zeroing” multiplexer before the memory request for read requests. We opt for separate multiplexers as it allows our cache to simultaneously send eviction requests (non-zero data to memory) while sending a write acknowledgment (zero data to processor).

In addition to these cache optimizations, we optimized our 5-stage pipelined processor. The primary causes for wasted cycles is (1) memory requests and (2) branch mispredictions. We seek to improve our branch prediction by implementing a Branch Target Buffer (BTB).

In our processor, a branch is not resolved until the X-stage. But as a result of compiling loops, many branches are taken *often*; so, we are constantly paying the two-cycle penalty. We could improve performance if we *predict the branch result* in F, and squash if we mispredicted.

The Branch Target Buffer (BTB) is a module with 4-entries to track branches: storing their PC, a valid bit, a taken bit, and their target address. It uses FIFO replacement. It’s updated in the X-stage. If a branch is taken and it’s *not in* the BTB, it’s added to the BTB.

We also consider the special case where a branch *isn’t usually taken*. We don’t want to predict taken *everytime* if it was only *taken once*. To resolve this, we add a “taken bit”.

In the F-stage, we search the BTB with the current PC. If we match a *valid* entry, and it was *taken on its last execution*, then we set the next address to its target. Otherwise, we continue to speculate PC+4. We also keep track of our prediction. Then, once we reach the X-stage, we determine if we mispredicted, and, if so, we squash and redirect control flow.

In other words, our Branch Target Buffer always predicts the previous branch result.

Software Optimizations

Additionally, we optimize our sorting algorithms to run faster on our multi-core system. First, we looked at the single-threaded sorting algorithm which is used as a subroutine. We looked at and analyzed various quicksort and insertion sort implementations. We chose these specific algorithms for their minimal memory overhead and cache performance.

Specifically, we looked at various partitioning schemes for quicksort: "Hoare" partitioning and "Lomuto" partitioning. For "Hoare", we also further investigated a median-of-three pivot. We found that quicksort had good performance for larger arrays, but it degraded significantly for smaller arrays due to recursion overhead. As such, we investigated with *hybrid quick sorts* which use *insertion sort* as a base case once the size reaches a specified threshold.

In the end, we chose a *hybrid quick sort* algorithm which uses "Lomuto" partitioning with a insertion sort threshold of 24 elements. See the next section for experimental details.

Next, we optimized our multi-threaded algorithm. Now that the subroutine was optimized, it suffices to optimize the merge step at the end of the algorithm. Originally, we had core 0 merge all 4 subarrays. However, this does not exploit parallelism between cores.

Instead, we implement a *parallel merge*. First, we observe the classic merge can be *reversed*. That is, we can merge two arrays *backwards* by starting at the end of each array.

Then, we notice that one core can do the "classical merge" while another core does the "backward merge". If they both stop at the halfway point in the output array, the array will be *fully sorted* upon termination (forward sorts first half, backward sorts second half).

This allows us to evenly divide the work of merging among two cores. In our optimized implementation, cores 0 and 1 merge the first two subarrays, cores 2 and 3 merge the next two subarrays. Then, cores 0 and 1 merge the two resulting subarrays into the final array.

It is possible to further parallelize the final merge. We implemented a quad-core merge: (1) cores 0 and 1 merge the *even indices*, (2) cores 2 and 3 merge the *odd indices*, and finally (3) compare indices i and $i+1$ for i odd, swapping if necessary (parallelizing across all cores). This is known as a odd/even merge; but, the benefits were negligible, so we dropped this.

5 Evaluation

Hardware Evaluation

First, we evaluate the results of our hardware optimizations on the five different microbenchmarks: binary search (*bsearch*) complex multiplication (*cmult*), masked filter (*mfilt*), vector-vector add (*vvadd*), and *our* sorting algorithm (*sort*). For these applications, we measure the speedup as a result from our optimizations on the *single-core system* (see Figure 5).

Looking at our performance results (Figure 5), we see an average performance increase of 3.09× as a result of our cache optimizations. This matches our expectations, as we've increased our instruction cache throughput by 4× with our new 1-cycle *sustained hit latency*. We also gain further performance benefits due to decreased *write* and *read miss latency*.

The cache optimizations resulted in an area increase. We had to add a second set of SRAM peripherals (to read both ways), as well as two new multiplexers to the datapath. This small increase in area leads to a corresponding increase in dynamic power usage. However, by significantly decreasing the execution time of programs, the energy/program decreases.

We also see a small increase in our cache critical path (likely critical path for the system). As upon reading the data array, there are now a few additional multiplexers before a register.

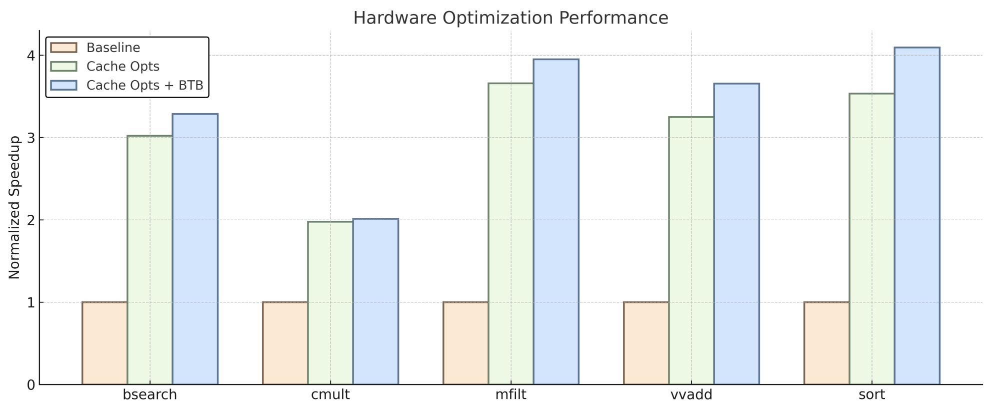

Figure 5: Performance of Hardware Optimizations on Microbenchmark Performance

However, the delay of these multiplexers are small compared to the SRAM read delay.

Overall, the performance improvements (3×) caused by these cache optimizations more than justify a potentially slightly lower clock frequency or any additional power consumption.

Again, looking at our performance results (Figure 5), we see an *additional* performance increase of 9.4% from adding the Branch Target Buffer. This makes sense as many of these programs are iterative (predictable branches). We see a large speedup (15%) in our optimized sorting algorithm, which is justified by the *nested for loops* in the insertion sort base case.

The added hardware was minimal — a few registers and accompanying control logic. Thus, the additional power is small. Also, it is unlikely that the BTB is on a critical path. Overall, the 9.4% performance increase is well worth the additional components for the BTB.

Software Evaluation

Next, we evaluate the results of our software optimizations. In particular, we perform a design space exploration to select the best sorting algorithms for our hardware systems.

First, we profiled various sorting implementations on the *single-core system*. The results of this can be seen on the left hand side of Figure 6. For our profiling, we simulated each algorithm on three randomly generated datasets for various array sizes ranging from 2 to 512, and then averaged the resulting execution time. First, we notice that insertion sort performs *best* for small arrays, but scales terribly due to its $O(n^2)$ asymptotic complexity.

Next, we notice that the (naive) quicksort implementations scale significantly better. Out of the three naive implementations, we notice that Lomuto partitioning performs the best. This is likely due to its inherent simplicity of the algorithm (fewer dynamic instructions).

However, quicksort is significantly worse with small array sizes due to its recursive calls. This motivates a *hybrid sorting algorithm*. We chose our best quicksort (Lomuto), and, when a recursive call is *smaller than threshold*, we utilize *insertion sort*. We considered thresholds of 16 and 24 since quicksort and insertion sort perform similarly on arrays of those sizes.

As seen in Figure 6, the "Lomuto" *hybrid quicksort* with threshold 24 performs best. It outperforms every other algorithm (in the limit), and is our chosen single-threaded algorithm.

Next we profiled various merge algorithms on the *multi-core system*. The results of this can be seen on the right hand side of Figure 6. We repeat a similar profiling methodology.

We utilize the *optimized* single-threaded implementation as a subroutine. Thus, the main

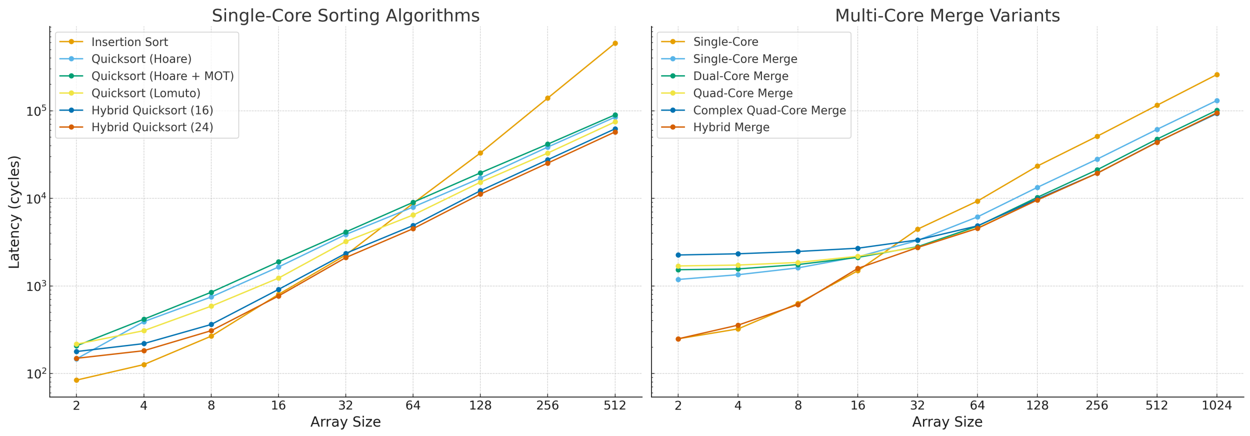

Figure 6: Software Profiling of Various Sorting Algorithms

concern of our multi-threaded implementation is how to *merge* the 4 sorted subarrays. As a baseline, we've plotted the performance of a single-threaded (only core 0) merge algorithm.

Our naive implementation is the "single-core merge". All four cores sort four subarrays (equally-sized), and then core 0 will merge each all four arrays into a singular array. A slightly more efficient algorithm utilizes two cores: each merges one pair of subarrays, and then they *both* (via a forward/backward merge) merge the final array ("dual-core merge").

Then, we implement a "quad-core merge". As previously discussed, this has pairs of cores sort the pairs of subarrays, finally having two cores perform the final merge. Additionally, we implemented a "complex quad-core merge" which uses all cores in a even/odd final merge.

First, we notice that for small arrays, the threading overhead is large. In other words, multi-threaded implementations that spawn *fewer* threads perform *significantly better*. However, for larger arrays ($n \geq 32$), more parallelized algorithms perform better as expected. Specifically, the "quad-core merge" performs the best for array sizes $32 \leq n \leq 512$. But, for array size 1024, the "complex quad-core merge" performs marginally better ($\approx 2\%$ faster).

We opt to create a *hybrid* multi-threaded sorting algorithm. To do this, we default to the single-threaded sorting algorithm for small arrays ($n \leq 16$) and utilize the "quad-core merge" for larger arrays. This achieves pareto dominance in our experiments (Figure 6).

Overall, for larger arrays, our *hybrid merge* algorithm performs 3× faster than the corresponding single-threaded algorithm run on the *multi-core system*. In the end, for the given reference data (100 elements), our single-threaded algorithm ran in 6791 cycles on the single-core system and our multi-threaded algorithm ran in 7182 cycles (6% slower).

Comparing Single-Core and Multi-Core Systems

Now, we compare our multi-core system with our single-core system, to determine if the multi-core system indeed *provides a speedup*. To do this, we will carefully quantify both the software (multi-thread) and hardware (multi-core) overhead of our implementations.

First, we evaluate the single-threaded (**SS** — single-thread software) algorithm on the single-core system (**SH** — single-core hardware). Then, we evaluate the multi-threaded (**MS**) implementation on the single-core system (**SH**). Following this, we also evaluate both software implementations (**SS, MS**) on the multi-core system (**MH**). We also evaluate the multi-threaded implementation (**MS**) on *a single core* in the multi-core system (**MH, 1C**).

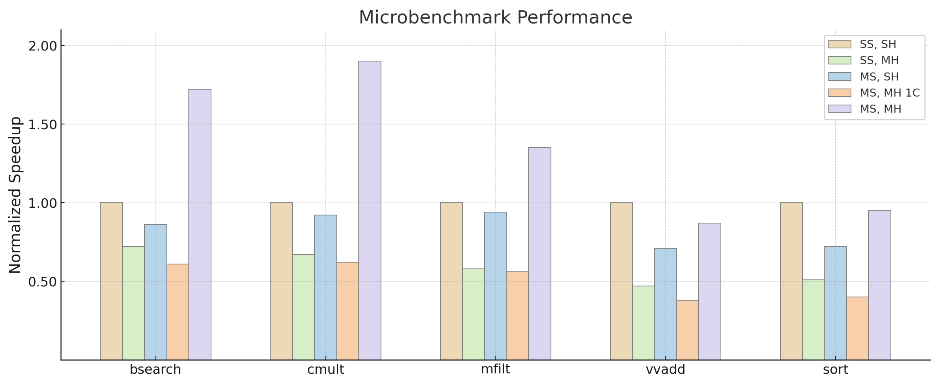

Figure 7: Performance with Single/Multi-threading on Single/Multi-core systems

The results and quantified and plotted as relative speedups in Figure 7.

Our single-threaded algorithms on the multi-core system run 74% slower on average than the single-core system. This demonstrates the inherent *hardware overhead* of our multi-core system. Additionally, our multi-threaded algorithm runs 22% slower than the single-threaded algorithm on the single-core system. This shows the inherent *software overhead* of threading.

These overheads make sense, as our memory transactions are significantly slower (≥ 3 cycles) in the multi-core implementation as each request must traverse multiple networks. Additionally, there is more work to do in order to spawn threads (for multi-threading).

Looking at specific benchmarks, we notice performance improvements moving to multi-threaded algorithms on our multi-core system for *bsearch*, *cmult*, and *mfilt*. Each of these algorithms can be easily parallelized to maximize performance. Additionally, these benchmarks have little software overhead — very small disparity between **SS, SH** and **MS, SH**.

The greatest performance improvement is seen in *cmult*. This corresponds with the fact that the main overhead in the algorithm is *multiplication*, which is split among the cores.

We notice performance degradation in *vvadd* and our sorting algorithm (*sort*). The former is very simple and as such the software overhead makes a larger impact. The latter has many stores/loads (for "swaps"), which takes time to traverse the various networks.

Putting the overheads together, we observe the multi-threaded algorithm on the multi-core system *with one worker* (active core) is 2.05× slower than the single-threaded algorithm

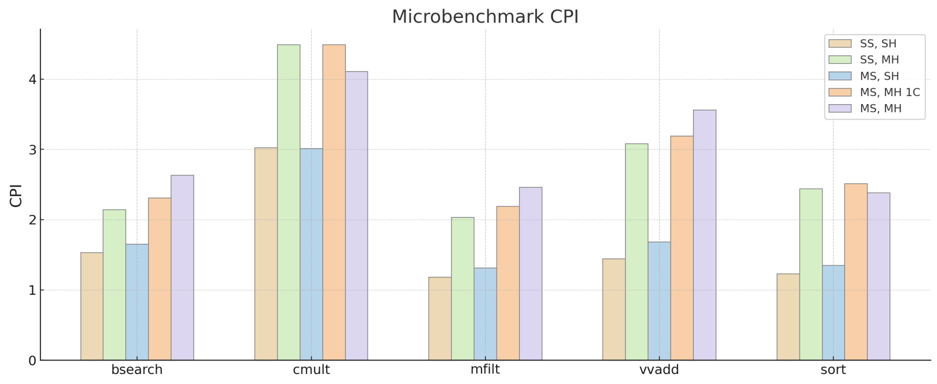

Figure 8: CPI with Single/Multi-threading on Single/Multi-core systems

on the single-core system. That is, there is a 2× overhead switching to a multi-core system. To see any performance benefits, we must achieve greater than 2× parallelism (performance).

However, we're limited to at most 4× parallelism, which limits any multi-threaded implementation to at most ≈ 2× improved performance. Also, by Amdahl's law, most algorithms are limited to much less parallelism, making true multi-core performance hard to achieve.

This highlights the *responsibilities* that multi-core systems assign to *software designers*. They must explicitly design algorithms for multi-core systems to maximize performance.

In Figure 8, we can see the *cycles per instruction* (CPI) of various microbenchmarks. We observe that (as expected), the multi-core *hardware* (MH) implementations have significantly *higher* CPI. This is due to each memory request having to *traverse the network*. As such, we don't see the benefits of our *optimized* one-cycle hit latency and write-ack.

Among our microbenchmarks, no benchmark exceeded a 2× performance increase. And, our multi-core system uses > 4× the amount of hardware — 4× the number of processors and caches *plus* multiple ring networks. This also results in > 4× dynamic power consumption.

Thus, we are paying significant area (> 4×), power (> 4×), and energy costs (> 2×) in order to gain ≤ 2× performance improvements. This makes it *hard to justify* our system. It's only worth it in the rare cases for *pure performance*, disregarding the area/energy costs.

6 Conclusion

In this lab, we constructed and compared *single-core* and *multi-core* systems. Then, with our cache and processor optimizations, we achieved an average 3.09× overall speedup on the *single-core* system with negligible area and energy overhead. Then, we explored a different approach to improve performance: *multi-core* systems, which have blown up in popularity since the end of Dennard Scaling. To support multiple cores running simultaneously, we had to implement a *ring network* to handle and arbitrate different requests. The *multi-core* system introduced *both hardware and software overheads*. In particular, across our microbenchmarks, there was a 74% hardware overhead and 22% software overhead. Our best *multi-core speedup* was 1.9× (*cmult*), which *doesn't justify* the > 4× increase in hardware. Thus, such systems are likely only useful in data-centers where performance is *critical*. In addition, we explored designing *software algorithms* for both systems. In this, we notice the importance of *hardware-software co-design* in maximizing performance of various programs. Overall, we've learned that there are still opportunities to *optimize* the *single-core system*. And, these optimizations are better than a *multi-core* system (3.09× vs 1.36× avg speedup). We believe the *multi-core* speedup (avg. 1.36×) is not worth the hardware overhead (> 4×).

7 Work Distribution

Anthony and Zephan individually wrote the baseline and alternative hardware/software implementations (and selected one implementation to use). Zephan wrote the test cases and did the hardware/software optimizations. The report was written together in its entirety.

We did not use AI in any form when creating the report or for our verilog implementations. We *did* enlist help from AI when we were creating test cases. We first brainstormed the test cases together *without* the use of any AI. We then explained the test cases to the AI and asked it to assist us in turning our designs into written *PyMtl* test case code.