RS: Second Homework

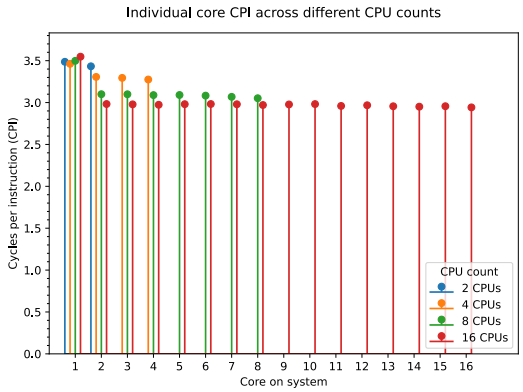
Mark Loboda in Simon Goričar

19. 4. 2025

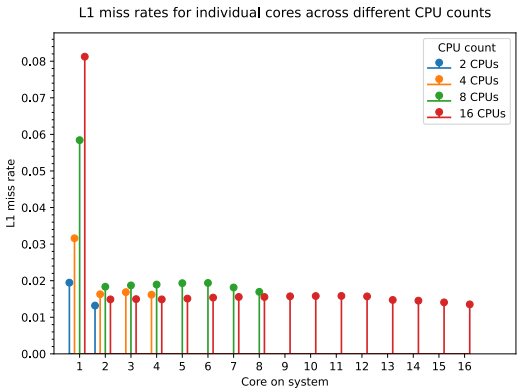
1 Benchmarks

1.1 First task: performance of a snooping-based cache coherence protocol

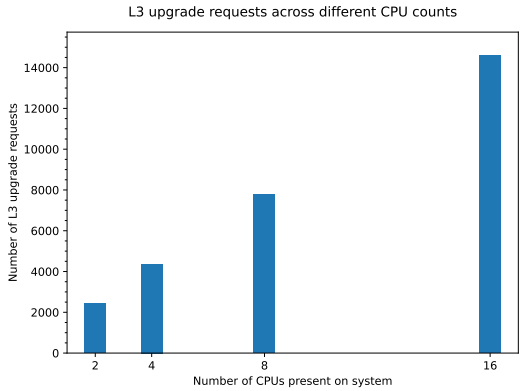
In this task, we ran the Cholesky decomposition workload using `GEM5` and observed several metrics related to the snooping-based cache coherence protocol in a multiprocessor system. We benchmarked systems with 2, 4, 8, and 16 processors, each one with 32 KiB of 8-way associative L1 cache and 256 KiB of 8-way associative L2 cache; we used 2 MiB of 16-way associative global L3 cache.



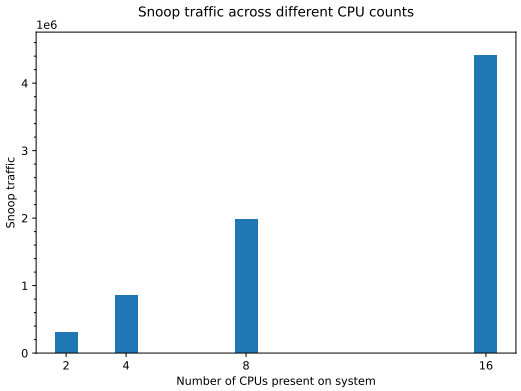
(a) Per-core CPI against number of CPUs.



(b) Per-core L1 miss rate against number of CPUs.



(c) L3 upgrade requests against number of CPUs.



(d) Snoop traffic against number of CPUs.

Figure 1: Results of our snooping-based cache coherence protocol benchmark.

First off, let's have a look at how the average and per-core **cycles per instruction** (CPI) values are impacted by different numbers of processors (figure 1a). As we can see on the plot, the CPI of the primary processor stays rather fixed, but each subsequent core observes a lower CPI, with a 16-CPU system having the smallest of all tested configurations. It's hard to say precisely why, but our guess is that, since cores other than the first one are usually part of OpenMP's worker thread pool, the work those threads execute is commonly numeric, focused, and uses heavily optimized operations or instructions (or a `for` loop with rather simple computations, which the compiler may be able to optimize well). This could also mean that the branch predictions for those focused short operations, if any, are more often correct, leading to a lower average cycles per instruction value.

Secondly, let's look at the **L1 miss rate**, both on average and per core (figure 1b). As we can see in the figure, the higher the number of CPUs in the system, the larger the L1 miss rate of the primary core; the likely reason is that the primary core belongs to the main thread, whose goal is to execute the entire program, not just the focused computations (which the rest of the cores mostly do). If we assume that, we can conclude that the work

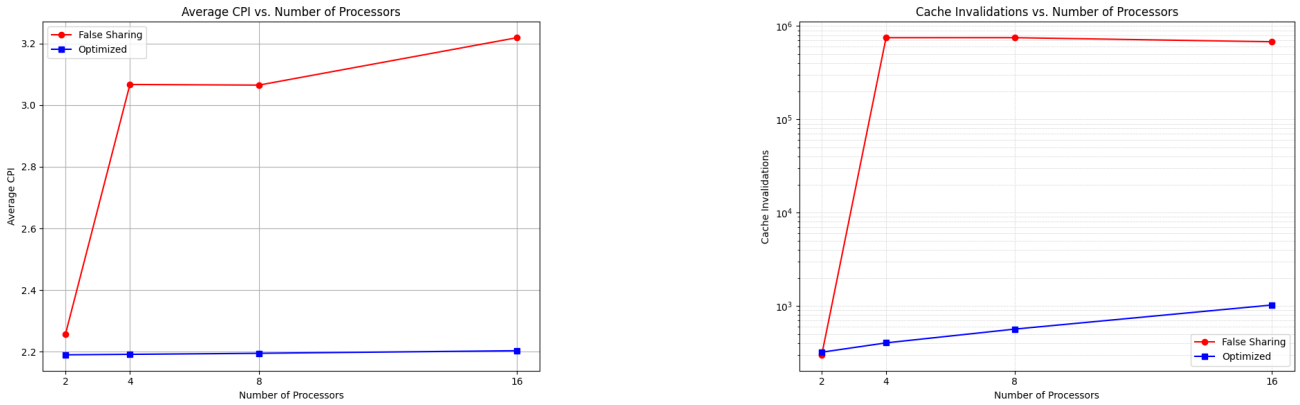
the rest of the cores do is limited in the number of instructions and often repeated (e.g., in a `for` loop), leading to a rather small instruction cache requirement compared to the first core, which has to go through the entire program.

Thirdly, **L3 upgrade requests** and **snoop traffic** (figures 1c and 1d). Here we simply observe that the number of L3 cache upgrade requests and the presence of snoop traffic grow as the number of cores on the system increases, seemingly linearly. The increasing number of L3 cache upgrades could come from the higher number of cores having to compete for an upgraded cache line on their core more often (for cases where two or more cores need to write to the same cache line, even if not to the same value). The increase in snoop traffic is also very sensible, as the cache coherence mechanisms have much more work to do as the number of coherent cores to maintain gets higher.

1.2 Second task: Impact of false sharing on performance in directory-based cache coherence protocols

In this task, we examined the impact of false sharing on a parallelized program for computing digits of π. Two program versions were compared: the first with false sharing present, where multiple threads share a cache line, and the second, where the program is optimized to avoid this issue. False sharing can lead to unnecessary cache line invalidations and increase memory traffic.

We used `GEM5` configured with the SMP Ruby coherence protocol and two versions of the workload. Additionally, we used 32 **KiB** of 8-way associative L1 cache and 256 KiB of 8-way associative L2 cache, both with 64-byte cache lines. We measured the performance using 2, 4, 8, and 16 processors.



(a) (b)

Figure 2: Results of our false sharing benchmark.

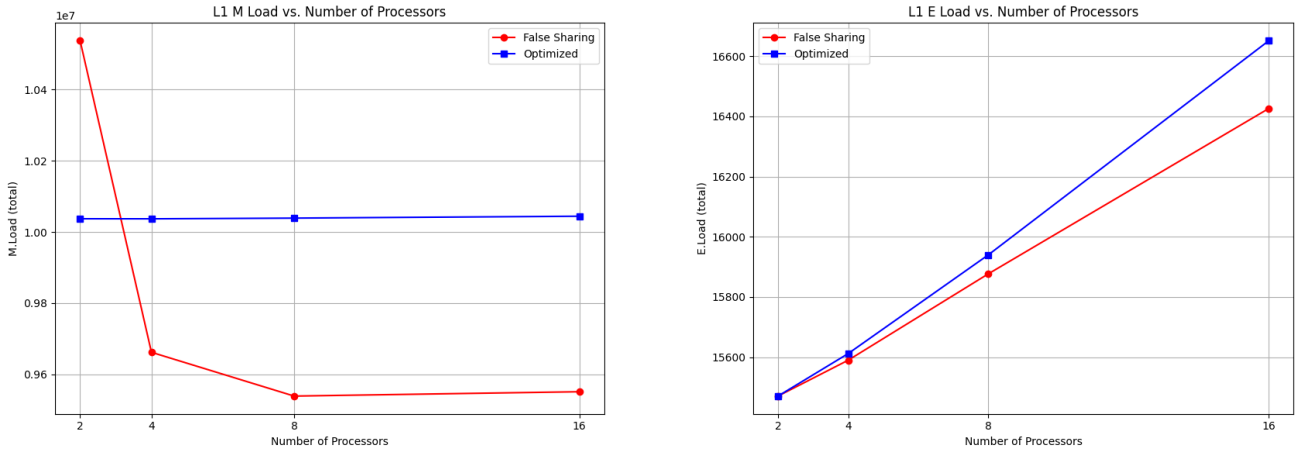
We noticed that the CPI of the false sharing implementation is substantially higher as the number of processors increases, while the optimized version's CPI stays relatively the same (figure 2a). This is likely to be in large part due to the number of cache invalidations occurring in the workload with false sharing, meaning a larger number of cycles is spent reloading cache lines (figure 2b). The problem is compounded as the number of processors increases because more cores continuously fight over shared cache lines.

We also analyzed the memory coherence behavior of the two workloads by looking at the number of L1 load operations across the four MESI states: **Modified** (M), **Exclusive** (E), **Shared** (S) and **Invalid** (I).

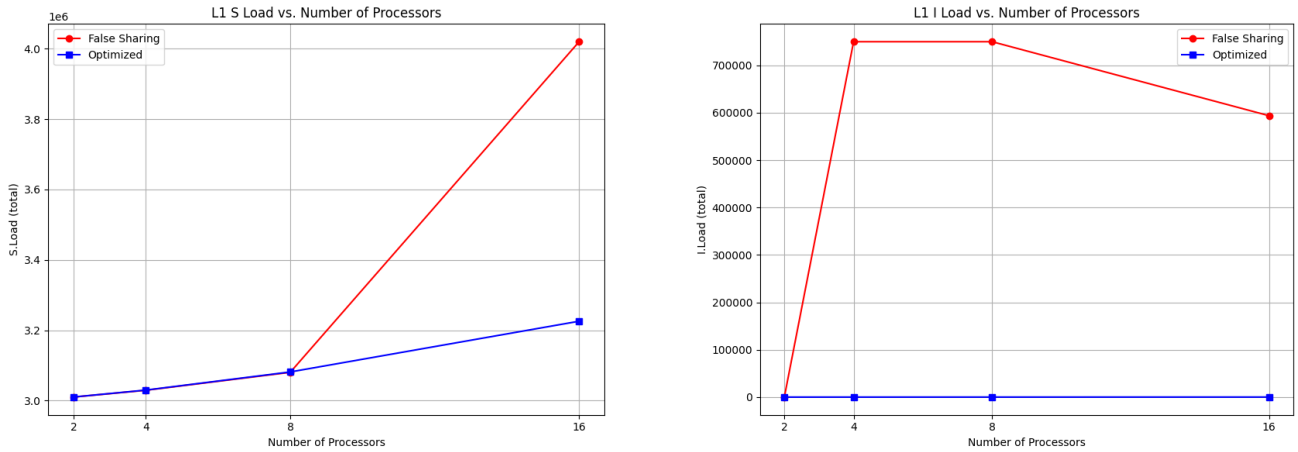
The measured values clearly differ between implementations of the two integration workloads. **I-state** loads are significantly higher in the false sharing algorithm, as the false sharing causes frequent cache line invalidations, forcing cores to reload frequently. The optimized version shows consistently lower loads in the I-state. Operations in the **S-state** only deviate with sixteen processors, where the optimized algorithm shows a more stable and efficient scaling of the number of processors. The number of **E-states** scales consistenly with the number of processors but is more frequent in the optimized algorithm, suggesting an improved handling of non-shared data. When looking at the **M-states**, the optimized algorithm displays a consistent number across all numbers of processors, while the false sharing only shows a high number with two cores but quickly falls off when scaling to more cores. We conclude that the optimized version of the algorithm improves utilization of the cache hierarchy and the MESI protocol, improving efficiency.

The data for the L2 cache requests 4 in the false sharing algorithm shows significantly more `GETS` and `GETX` requests compared to the optimized version. The implementation with false sharing causes increased memory traffic due to frequent cache invalidations and data reloads.

To conclude, we can see that the optimized version improves on cache line invalidations that otherwise occur with the false sharing implementation. It also increases memory traffic with L2 cache `GETS` and `GETX` requests. This affects the CPI and overall execution time (figure 5) when running both algorithms.



(a) (b)



(c) (d)

Figure 3: Results of the pi integration benchmark of the number of loads under different states.

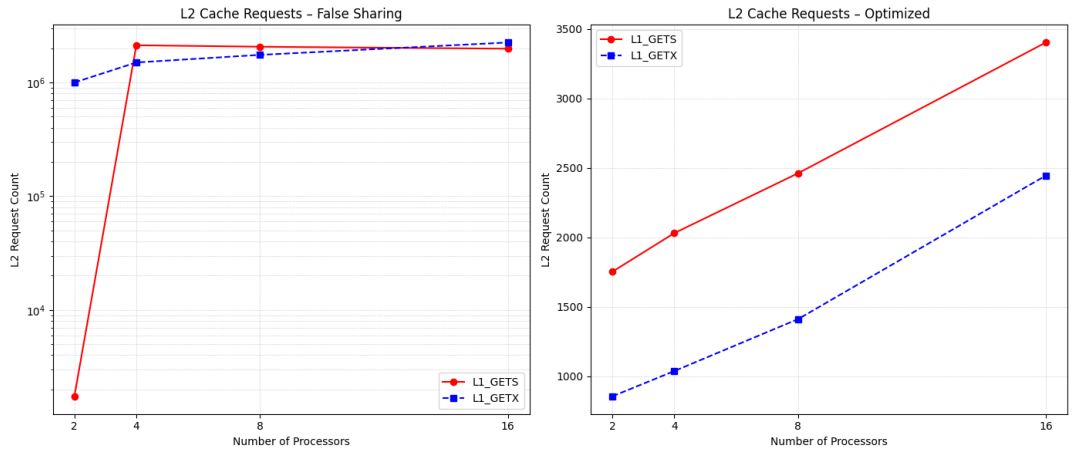


Figure 4: L2 requests with two implementations of the workload.

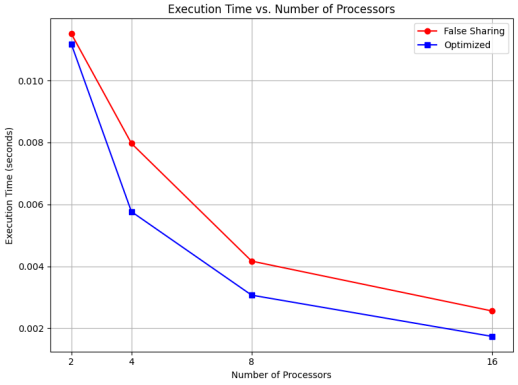
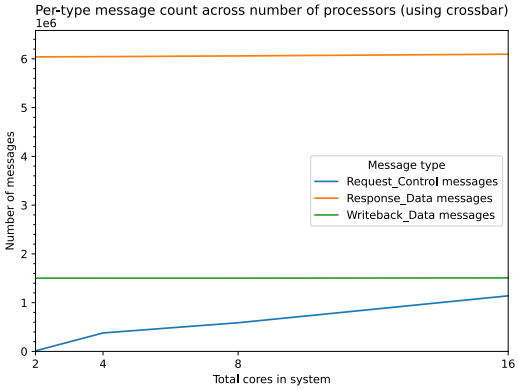


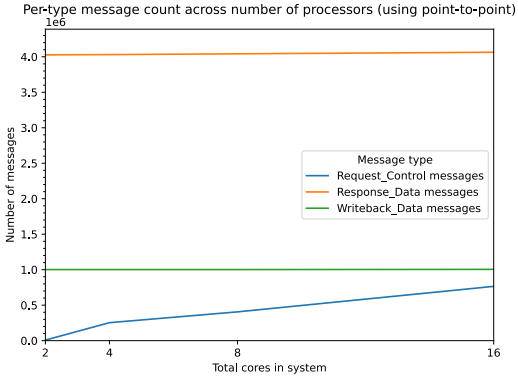
Figure 5: Execution times of the two implementations across different processor counts.

1.3 Third task: performance of different interconnection networks

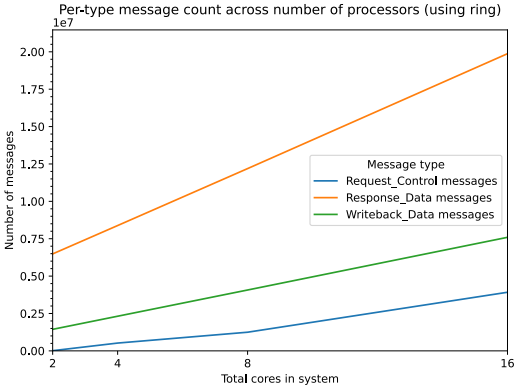
In this task, we ran the STREAM benchmark and observed several metrics related to the traffic on the interconnection network. We benchmarked systems with 2, 4, 8, and 16 processors, 32 KiB of 8-way associative L1 cache, and 256 KiB of 8-way associative L2 cache.



(a) Network messages against number of CPUs in system for *crossbar* interconnection network.



(b) Network messages against number of CPUs in system for *point-to-point* interconnection network.



(c) Network messages against number of CPUs in system for *ring* interconnection network.

Figure 6: Results of our interconnection network benchmark.

As we can see in figures 6a, 6b, and 6c, the number of messages exchanged on the interconnection network monotonically grows as the number of cores in the system increases. The increase is least visible on the *crossbar* and *point-to-point* network topology, likely because in those two network types, the messages that need to be exchanged perform the minimal number of hops needed to reach their destination (1 hop in the case of point-to-point and about two hops in the case of crossbar). The somewhat linear increase is visible on the *ring* network because, as the number of cores increases, so does the number of average hops required to send a message to the core you want (it's rather unlikely the target core will be next in the chain).