ZORO – Zack's Open Row Oriented Memory Request Scheduler

Jack Davidson Zackery Painter
Electrical and Computer Engineering
Rose-Hulman Institute of Technology
davidsjt@rose-hulman.edu painteza@rose-hulman.edu

ABSTRACT

Conventional memory scheduling takes advantage of the row currently open in memory by prioritizing memory requests in that row. Only if there are no more memory requests for the open row does memory switch to another row, based on the oldest memory request in the queue. This First-Ready-First-Come-First-Serve (FR-FCFS) scheduling is standard in modern processors. This paper aims to take advantage of FR-FCFS by implementing a memory scheduler that prioritizes same-row memory requests sent from the memory controller – ZORO.

We found a slight improvement in activation energy and read latency for larger cache sizes but found minimal changes in row hits for all cache sizes. For smaller cache sizes, our results were worse for all metrics we recorded.

INTRODUCTION

As processor performance continues to improve at a faster pace than memory performance, memory becomes an increasing barrier to maximal performance (commonly known as the memory wall problem). Our proposal is to use a buffer to complement an FR-FCFS scheduling policy in an attempt to mitigate this issue.

IMPLEMENTATION

In a FR-FCFS scheduler the memory requests that are for the DRAM row that is currently charged are handled prior to ones that are not, hence the "first ready" part of the name. This situation is the best-case scenario as row pre-charge time can be ignored, leading to lower latencies and less time with a processor or thread waiting for data to continue operations.

We can improve this by re-ordering requests in the CPU before submitting a transaction to the memory controller. ZORO achieves this by speculating the current open row in the memory controller and only scheduling transactions that are known to be in that row. ZORO speculates the current open row by recording the last known row that was opened and assuming that the memory controller still has this row open. When we begin issuing requests for a different row, the stored row updates and ZORO continues its search. This takes advantage of the "first ready" portion of the FR-FCFS because the memory controller should immediately issue transactions in the currently opened row.

The main problem of this approach comes with memory requests that are outside of the current row. If we don't provide the data the processor needs then it must wait, wasting precious power and compute cycles. Our solution is to track the age of each request in our buffer and to issue these requests once they reach a certain age, even if the transaction isn't in the currently opened row. We will discuss this further in the analysis of our results.

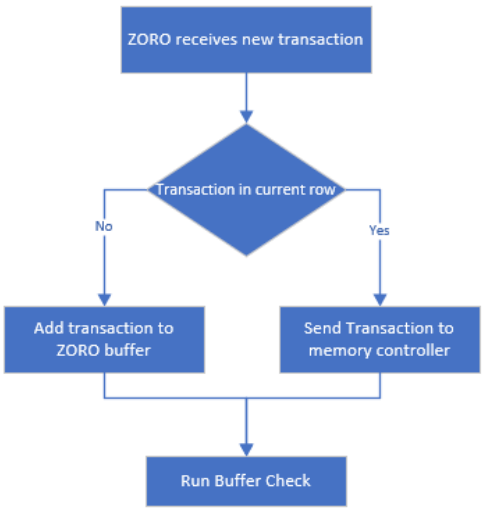


Figure **1- ZORO's new Transaction handler. ZORO always checks for additional transactions that it can send on receiving a new transaction.**

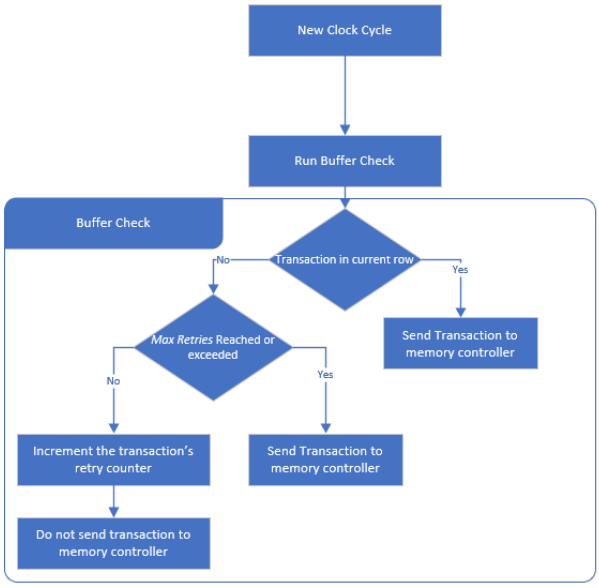
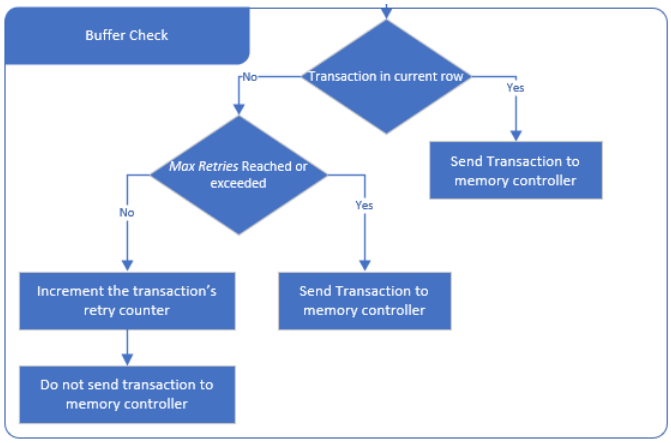


Figure 2 - On each clock cycle we run a buffer check to send any transactions that are in the speculated row. We also send any transactions that have exceeded the max retries allowed. If the transaction isn't in the row, the max retries counter increments.



Figu**re 3 - A zoomed in view of the ZORO buffer check**

SIMULATIONS

Our primary goal was to increase the number of same-row memory accesses. To measure this, we used DRAMsim3, which provides many statistics such as the number of read and write commands it received, and how many of them had various ranges of latencies. We used gem5 for our CPU simulator primarily due to its ability to easily integrate DRAMsim3[2] [3]. Our benchmark of choice was EEMBC Coremark as it provides a variety of realistic workloads (list processing, matrix manipulation, state machine validation, and CRC) as well as a variety of configuration options.

Gem5 allows for a variety of CPU models to be implemented. We ran our simulations primarily with the TimingSimpleCPU and O3CPU options [2]. Our initial testing was done with TimingSimple model as the O3CPU has a tendency to fail with errors when left running for long periods of time. To mitigate this error, we ran Coremark with a set number of iterations smaller than the default option of 400000. Our initial O3CPU testing showed almost no difference between 100 iterations and 2500 iterations of Coremark. This is because the data that Coremark is operating on was already in the cache, and DRAMSim3 handles main memory, which is what we

wanted to look at [3]. This was not an issue with TimingSimple as it does not require the use of caches. We decided to use 100 Coremark iterations for the remainder of our testing because of this. 100 iterations provided a balance between simulation time and number of DRAM commands performed.

After we noticed no difference in the number of main memory transactions with the increase in iterations in Coremark, we decided to change the cache size to force the number of memory transactions to increase. This was important because it would allow us to better see the differences in the results of ZORO. We do understand that this isn't ideal, and not a very realistic way to get a result, but for our purposes it was necessary.

In order to simulate a more memory intensive benchmark, we changed the size of the cache available to the O3CPU model. While this is not a perfect solution, it increases the number of memory transactions that take place to provide variation to our data.

RESULTS

While analyzing the data, we found interesting results in energy consumption, latency, and row hit rate. We noticed that for the default cache configuration, we did slightly better than the baseline in most of our data. However, as the cache size decreased, we had a much worse performance than the baseline.

An example of this is visible in Figure 4, which shows the activation energy across multiple different max retry values. The figure shows that for the lower max retry values we actually have a lower activation energy than the baseline parameters. For larger values of max retries, we use more energy than the baseline. We argue that this isn't an issue because the we can adjust max retries to give us best results. However, as mentioned above, as we decrease the cache size, we use more activation energy than the baseline regardless of the max retry value. This is shown in Figure 5, which was captured with $1/256^{th}$ the baseline cache size across all max retry values.

The read latency with ZORO is lower than the baseline for each value of max retries. The lowest latency was around 8 max retries. This is different than the activation energy, which was lowest around 3 max retries. This is represented in figure 9 below, which shows the read latency for a smaller cache. As shown, we have a higher latency than the baseline. This also shows the decrease in performance as the cache size decreases. We anticipate that this is due to an oversight in ZORO that withholds non-same-row requests until the max retries value has been met. Given more time, this would be the first concern to address.

Read and write row hit rates both tended to drop as max retries increased, shown in figures 6 and 7. The highest improvement ZORO achieved over the baseline in the default cache size was 0.22% at 2 max retries for read hit rate, and 0.91% at 6 max retries for write hit rate.

We noticed that as cache size decreases, the hit rate begins to drop below that of the baseline, similarly to latency. We anticipate the root cause is the same.

Another interesting observation from the hit rates as cache size decreases comes from the unmodified gem5 behavior. Read and write hit rate cross over twice, then converge towards 100% as show in figure 10. This convergence is due to operands being constantly overwritten, to the point where the victim cache cannot keep up. This results in many operations happenings within the same row for a single set of calculations.

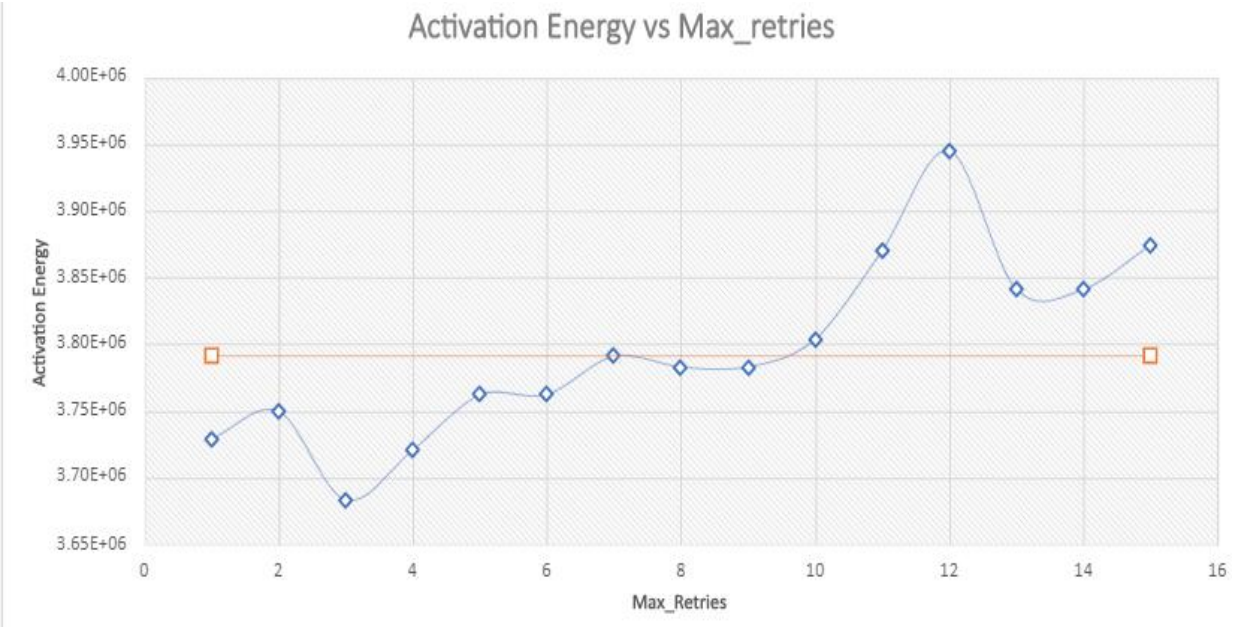


Figure 4- Activation Energy vs Max Retries. As shown in this figure, our lower values of max retries have a lower activation energy than the baseline. After 10 max retries we actually use more energy than the baseline. This is for the default cache configuration.

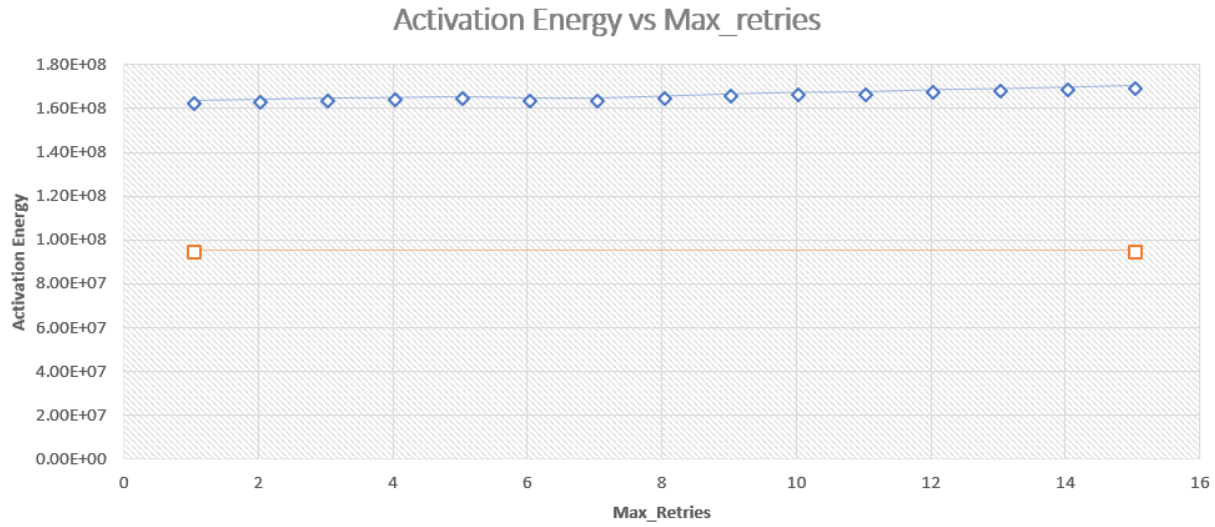


Figure 5 - Activation Energy vs max retries for 1/256th the default cache size. As shown, for every max-retry value we use more energy than the baseline value.

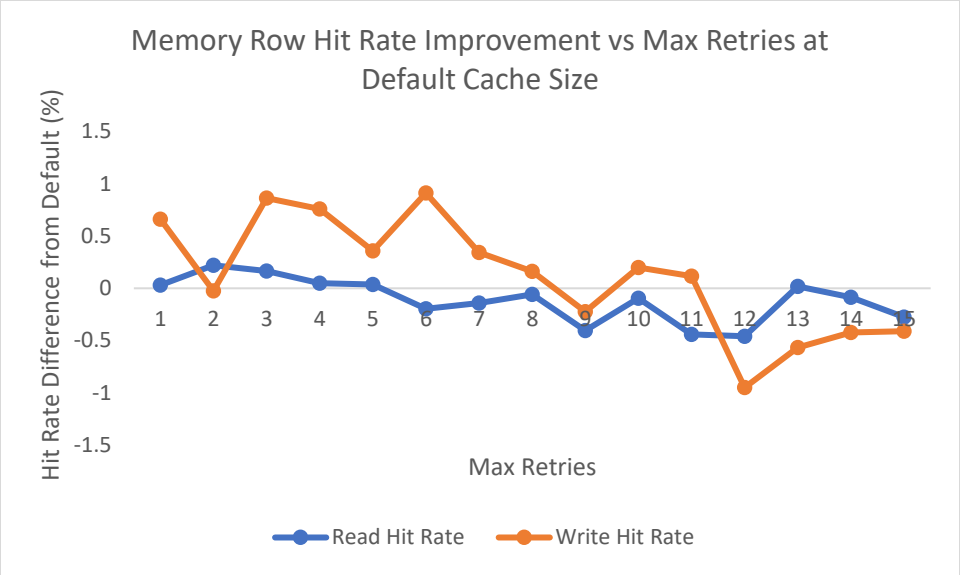


Figure 6 – Row hit rate for memory reads and writes vs max retries for the default cache size, normalized to the baseline row hit rate. As max retries increases row hit rate tends to decrease.

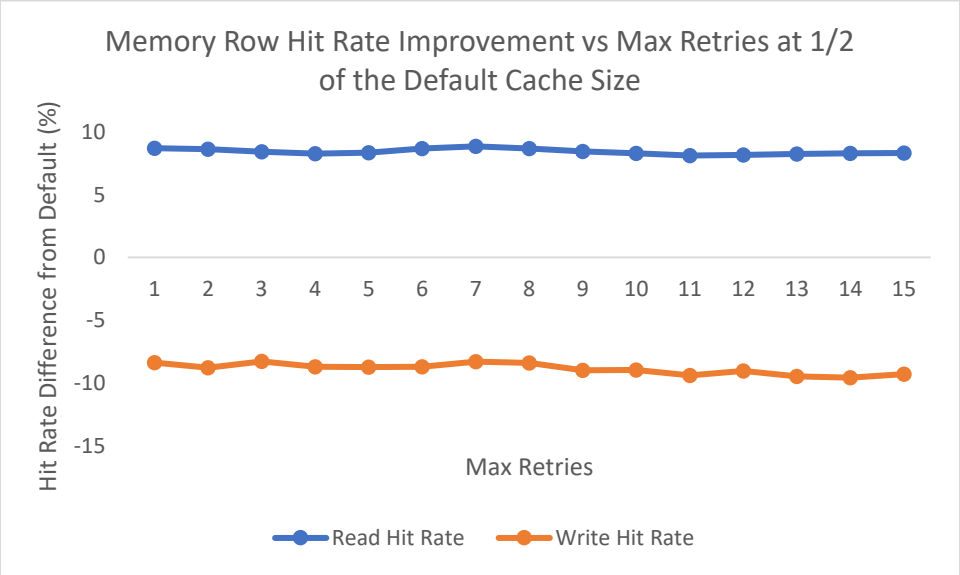


Figure 7 – Row hit rate for memory reads and writes vs max retries for 1/2 of the default cache size, normalized to the baseline row hit rate. The read hit rate and write hit rate almost cancel each other out.

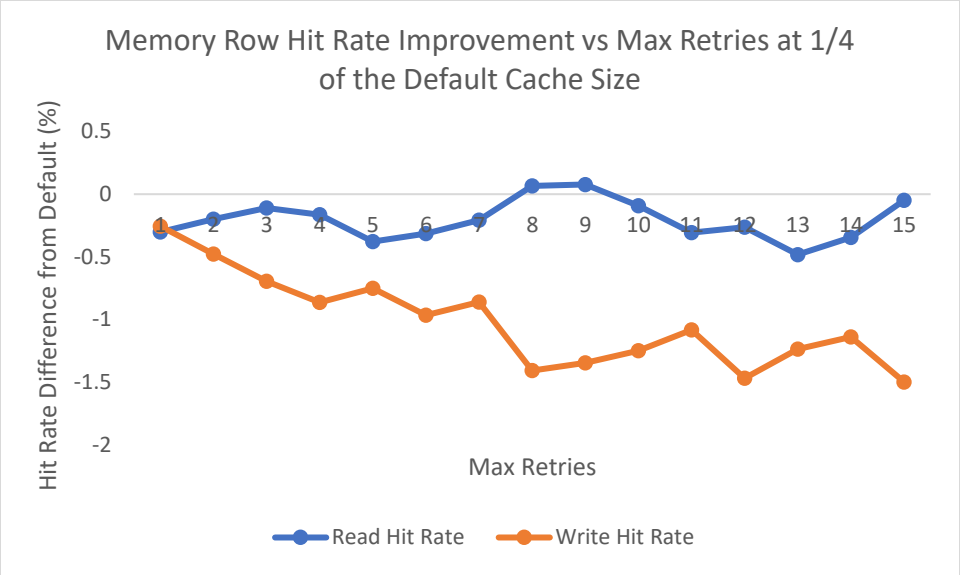


Figure 8 – Row hit rate for memory reads and writes vs max retries for the 1/4th of the default cache size, normalized to the baseline row hit rate. As max retries increases write hit rate tends to decrease, while read hit rate stays more consistent. Both read and write tend to be below the baseline.

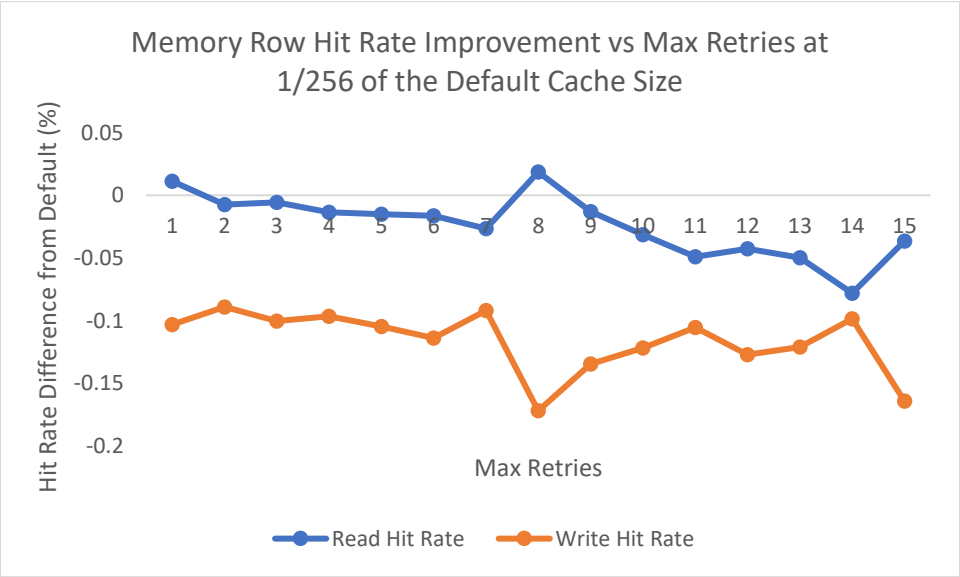


Figure 9 – Row hit rate for memory reads and writes vs max retries for the 1/256th of the default cache size, normalized to the baseline row hit rate. As max retries increases row hit rate tends to decrease.

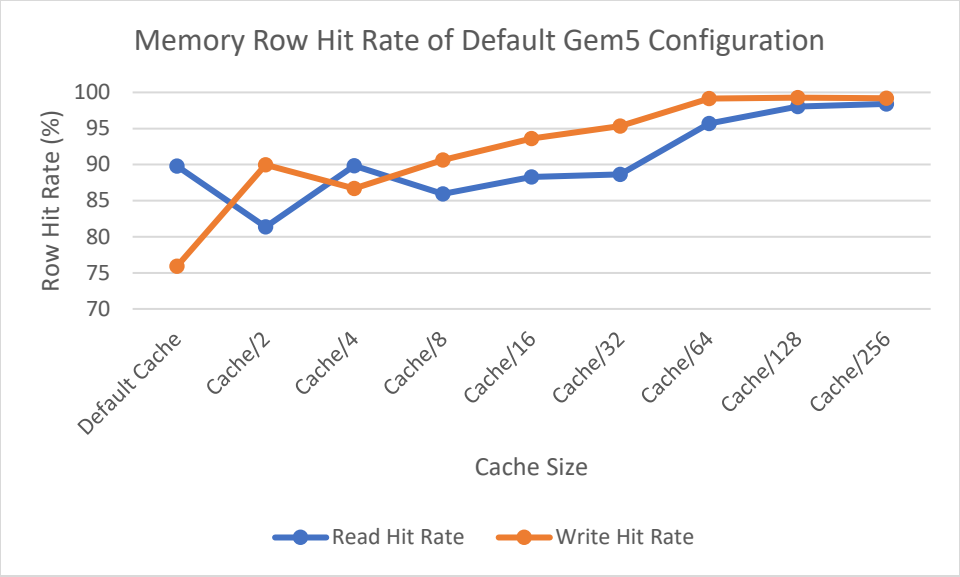


Figure 10 – Row hit rate for memory reads and writes vs cache size for the default gem5 configuration.

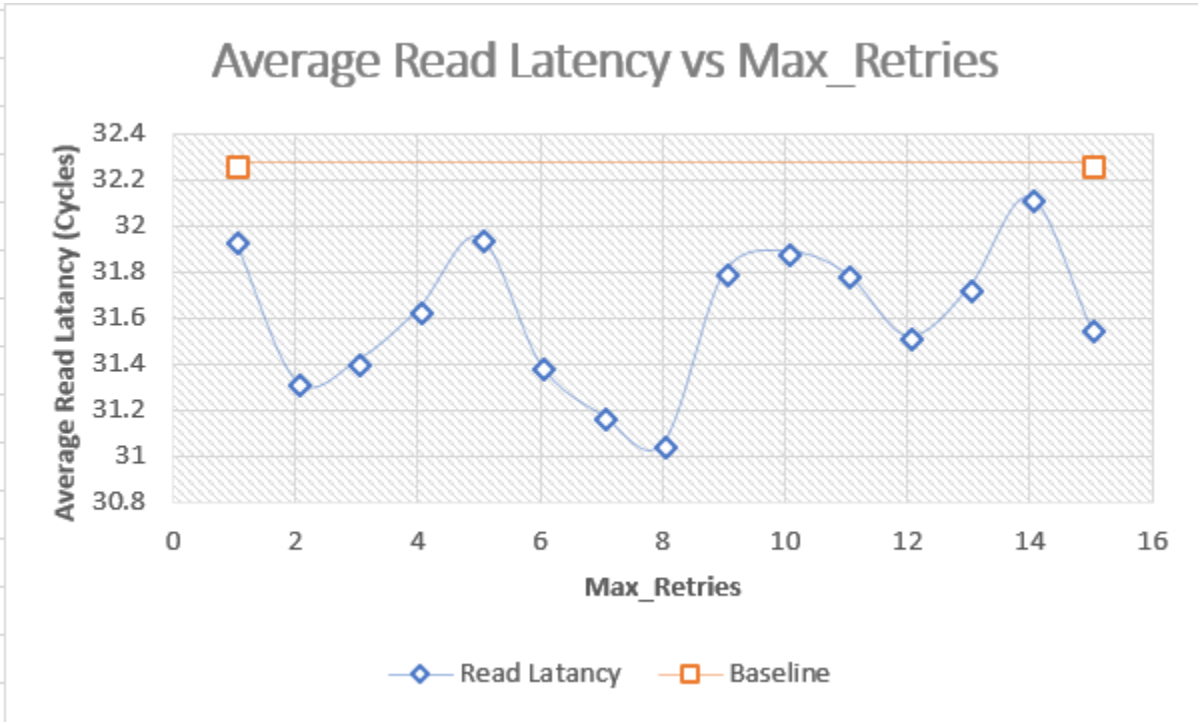


Figure 11 – Average read latency vs max retries plotted for the default cache size. Also plotted is the baseline read latency without ZORO for the default cache size.

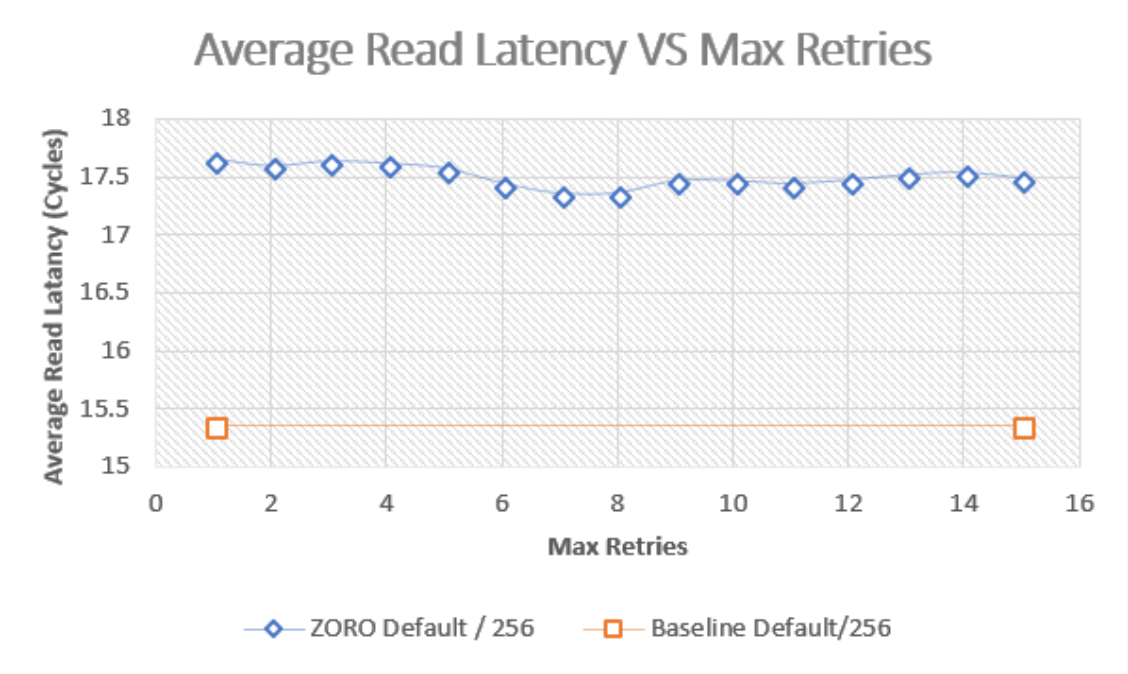


Figure 12 - Average read latency vs max retries for 1/256th the default cache size. As shown, our latencies are much higher than the baseline 1/256th cache size simulation without ZORO.

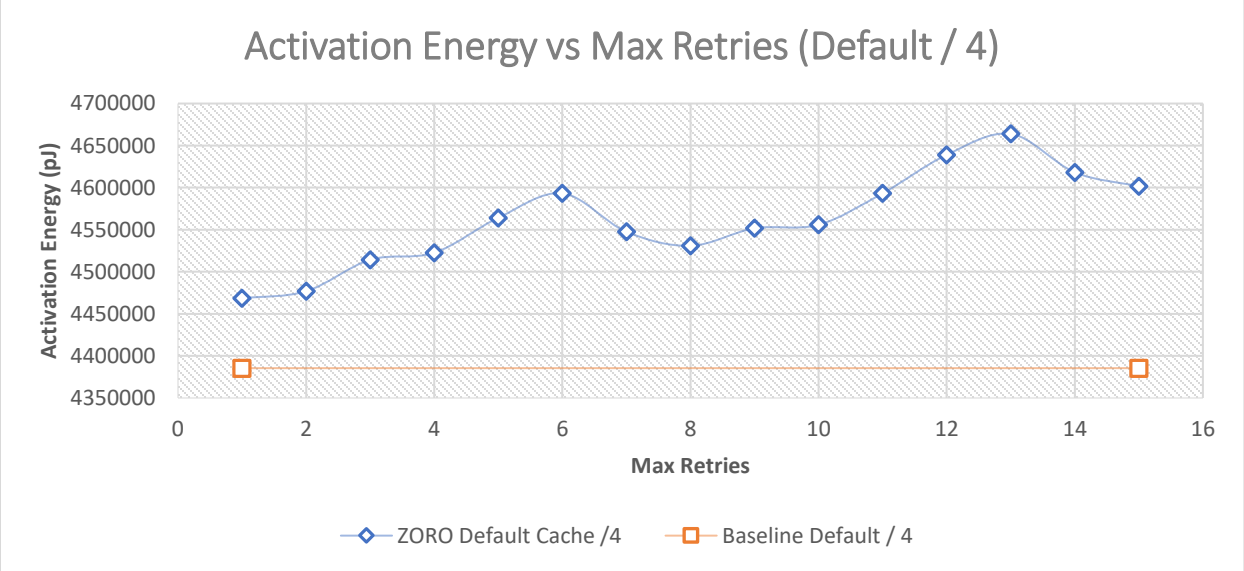


Figure 13 - Activation Energy VS Max Retries for a cache size of Default/4. As shown, even with a significantly larger cache size than the 1/256th cache, we still have a higher total max energy than the unmodified baseline system.

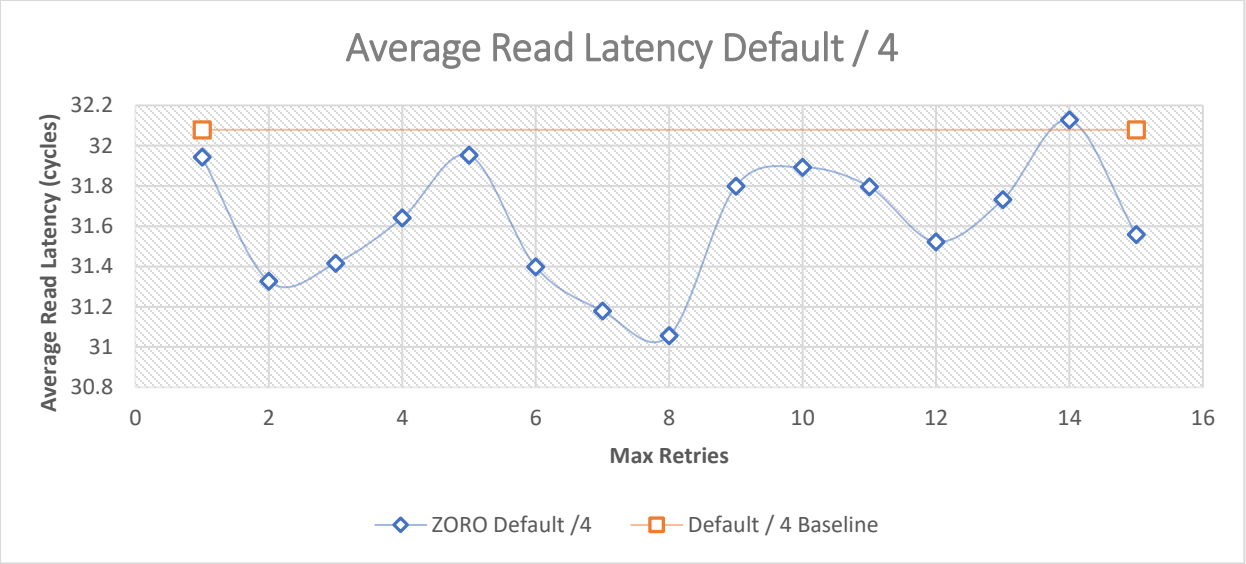


Figure 14 – Average Read Latency for 1/4th the default cache. As shown from this figure, with 1/4th the cache size we still have a lower read latency for most of the max retries values

CONCLUSION

ZORO is a CPU memory scheduling algorithm that seeks to take advantage of memory controllers that use FR-FCFS. This is done by grouping together transactions that are in the same row and sending the groups to the memory controller when we speculate that they are in the currently opened row.

ZORO was able to provide some benefits for GEM5's default cache sizes[2]. We were able to achieve a slightly better row hit rate and activation energy for lower values of our max retries parameter. Activation energy being about 3% better and row hit rate only being about 1% better for reads and 0.25% better for writes. However, as we changed the default cache size these numbers changed so that we had more energy and a lower hit rate.

We conclude that our ZORO algorithm, while not making much of a difference in the number of row hits, does save a considerable amount of energy in DRAM. Though, we would like to note that we did not take into account total energy of the system. We also conclude that we can improve our latency slightly with this algorithm as well as shown above.

FUTURE WORK

Running a larger variety of simulations, benchmarks, and memory configurations would give better perspective on the potential this system has to offer. We had trouble running simulations aside from Coremark with gem5, but this would be the first major step. More benchmarks would give a better perspective on how ZORO affects performance in various workloads. We anticipate that memory intensive workloads would have the highest benefit from this system. Our results are only simulated for a single DRAM configuration, 8 GB of DDR4 (DDR4_8Gb_x8_3200.ini from DRAMsim3). Different memory configurations could indicate applications where ZORO could thrive, as well as where it would be a waste of area for the difference in performance.

If there is promise to this system, the next step would be to improve the algorithm. The first step is to implement more similar logic to FR-FCFS. In its current state, ZORO will wait until the Max Retries value has been met before issuing a memory request, even if a full sweep of the buffer has shown there are no same-row requests. We anticipate that this would mitigate the decrease in performance compared to the baseline as the cache size shrinks. The next step is to track how many same-row memory requests have been issued in order to predict the row that DRAM has charged even after we have begun sending requests from a different row. Say ZORO send 5 requests to row A, then send 2 requests to row B. If it sees another request in row A, DRAM will likely still be handling the row A requests and can still push this request via the FR policy.

Our results are solely based on simulations as we did not have the time to attempt implementation of a physical system. A physical implementation of ZORO would allow for analysis of area as well as power and performance, providing better indication of the potential for this system in a commercial processor.

Additionally, we did not do any thermal analysis from DRAM or the CPU. In the future, we would attempt to do this to see if ZORO has any effect on the heat generated both in simulated models and in the physical implementation.

RELATED WORK

We found some related work on the subject of predictive memory schedulers by Hurr and C. Lin [1]. Their work was a bit different and sought to change the memory scheduler on the memory controller instead of editing the transactions coming from the CPU. Additionally, they did a more standard

prediction algorithm. This is in contrast to our ZORO algorithm that uses grouping and a very simple prediction to determine what transactions get sent from the CPU.

REFERENCES

[1.] Hur and C. Lin, "Adaptive History-Based Memory Schedulers for Modern Processors," in IEEE Micro, vol. 26, no. 1, pp. 22-29, Jan.-Feb. 2006, doi: 10.1109/MM.2006.1.

[2] J. Lowe-Power *et al.*, “The gem5 Simulator: Version 20.0+”, *arXiv [cs.AR]*. 2020.

[3] S. Li, Z. Yang, D. Reddy, A. Srivastava and B. Jacob, "DRAMsim3: a Cycle-accurate, Thermal-Capable DRAM Simulator," in IEEE Computer Architecture Letter

STATEMENT OF WORK

	Initial Simulations	Progress Reports	System Modifications	Simulations	Final report
Jack Davidson	55	40	20	70	60
Zackery Painter	45	60	80	30	40

Signed, Zackery A. Painter & Jack Davidson