# Coalescing Multiple Loads to Reduce Memory Traffic

Adarsh Mittal
amittal26@wisc.edu

Ishan Yelurwar
yelurwar@wisc.edu

Prathamesh Patel
prathamesh.patel@wisc.edu

Vanshika Baoni
vbaoni@wisc.edu

## 1 INTRODUCTION

Even with several different forms of Parallelism leveraged in superscalar processors these days, the greatest challenge faced by computer architects is the problem of memory bandwidth. Simply stated, the memory is not able to supply data at a rate fast enough to keep up with the number of instructions supplied by the pipeline front-end. Consequently, most often memory becomes the performance bottleneck of a system. Instructions that need to interact with the memory subsystem are usually some form of load and store instructions. In this project, we propose a way to handle multiple load requests efficiently. Quite often, we have load requests which fetch data from close-by or consecutive addresses. We look at a way to combine multiple loads that map to the same cache block in the memory. The idea is to identify the load addresses and determine if they map to the same cache block; if they do, we can fetch the corresponding data blocks from the cache, coalesce or combine these blocks and send it back to the processor. This not only helps in saving the overhead of multiple tag matching for the same cache block, but also helps to service multiple loads with just one access to memory.

## 2 MOTIVATION

The Von Neumann performance bottleneck is a well-known and persistent problem within computer architecture. The latency of an access to DRAM can cause the processor to stall for many cycles, a significant inefficiency.

Many techniques have been implemented to address this problem, with the most important being the use of small, fast caches close to the processor. Caches exploit the spatial and temporal memory access locality exhibited by most programs to reduce the latency of an access.

A data prefetcher can improve the utility of caches, by predicting what data will be used in the near future, fetching it from DRAM into the cache. But even with an optimal data prefetcher, sequential loads have to repeatedly access the caches block. These repeated cache accesses affect the performance of the pipeline even with the availability of the relevant data in the cache block.

A dynamic hardware supported coalescing of such loads fetches can prevent the unnecessary cache access. Such a support can decrease the load latency and provide wider working set window to the data prefetchers, increasing the throughput of the overall pipeline without increasing existing caches size.

With a modest investment in hardware support in the form of a Load Coalescing Unit (LCU), the load latency can be drastically reduced at run-time due to less cache accesses at various memory hierarchy without the need to alter instruction set architecture.

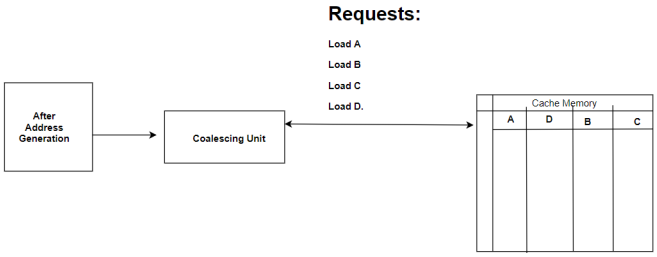
## 3 RELATED WORK

While there has been some work done to reduce number of instructions executed, all of them have had different approaches. John et al [1] proposed adding a code coalescing unit which stores the value written out by instructions earlier. According to their analysis, around 25% of the loads can be eliminated using this unit. Similarly, Moshovos and Sohi's scheme [2] tries to predict dependencies between load and stores and try and eliminate dependencies. This is a speculative scheme which does require a recovery mechanism. NoSQ proposed by Sha and others [3] also tries speculative memory bypassing without the use of a store queue. Tyson and Austin's scheme [4] uses a special load/store cache to perform memory renaming and improve the communication of memory values between instructions. Again, the approach is speculative in nature and needs to handle mis-speculations. We propose a novel approach which is not speculative in nature. Our approach also does not rely on prior store instructions in order to coalesce instructions. We take advantage of the locality of the load requests in order to coalesce multiple load instructions into one request.

## 4 HYPOTHESIS

Our approach tries to reduce the memory traffic by coalescing multiple load instructions with the help of a hardware unit which ensures that if there is a contiguous load request from the memory, it will fetch the data and do the tag comparison only once. Data can then be reordered and distributed to the destination registers of load instructions provided there are no dependent store instructions in between them. According to our hypothesis, this will lead to an improvement

in latency due to the reduction in tag comparison when compared with the baseline out of order processor without an overhead of increasing the cache size.




## 5 EXPERIMENTAL METHOD

We plan to carry out all our experiments using the gem5 simulator. We will be making changes to an out of order CPU model in Gem5 and then testing our changes on a couple of benchmarks which do actually have multiple load requests that have good spatial and temporal locality. We expect image manipulation benchmarks like 538.*imagick_r*, 510.*parest_r* and 511.*povray_r* from SPEC2017 to show this kind of behaviour and benefit from our approach.

## 6 TIMELINE

1. Literature review (Week 1): The first 3-4 days will be invested in reviewing the available research paper and talking with the professor regarding the feasibility of the proposed idea. It will require us to work on finding and understanding the existing implementation in Out of Order processor.
2. Tool Learning (Week 1): Getting familiar with the gem5 simulator and understanding the different files which are interacting to realize the architecture. We will be investing 2-3 days in understanding the same.
3. Identifying the Benchmarks (Week 2): Identifying the benchmarks that may have the expected behavior of multiple contiguous load requests. We will then try to run the benchmarks we find on gem5.
4. Work on building the Coalescing Unit: (Week 2-3): Understand the complexity in the implementation and try to implement the unit in the simulator
5. Running Benchmarks (Week 4): We will try to run the benchmarks on our updated CPU model and get the performance numbers. We will then try to correlate our results to our hypothesis to reason about the numbers that we collect.

## 7 SUMMARY

Our project basically consists of two phases - the first one being the design and implementation of the 'Coalescing Unit' and the second one being the evaluation of performance improvement that our proposed model can offer over the existing implementation of the load data fetch procedure from memory. We expect an increase in IPC with our approach as the load instructions and dependent instructions are able to get their values quickly.

## REFERENCES

[1] John, J., Teh, Y., Matus, F., and Chase, C. Code coalescing unit: a mechanism to facilitate load store data communication. In *Proceedings International Conference on Computer Design. VLSI in Computers and Processors (Cat. No. 98CB36273)* (1998), IEEE, pp. 550–557.

[2] Moshovos, A., and Sohi, G. S. Streamlining inter-operation memory communication via data dependence prediction. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture* (1997), IEEE Computer Society, pp. 235–245.

[3] Sha, T., Martin, M. M., and Roth, A. Nosq: Store-load communication without a store queue. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture* (2006), IEEE Computer Society, pp. 285–296.

[4] Tyson, G. S., and Austin, T. M. Improving the accuracy and performance of memory communication through renaming. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture* (1997), IEEE Computer Society, pp. 218–227.