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

ABSTRACT

Memory bottleneck still remains the major concern in today's CPUs. The speed at which memory is advancing has been unable to cope up with the increase in processing power. We therefore propose a solution to ease this problem. Nearby loads tend to have spatial locality which results in many accesses going to the same cache block. In this project we try and combine multiple load requests in order to reduce the requests going to the data cache. Our basic unit shows the potential to reduce memory accesses by around 80% on a couple of SPEC2006 benchmarks showing good locality. We therefore believe that this approach can lead to huge gains in memory bandwidth and as a result lead to a good performance speedup in applications that are memory intensive and show good spatial locality.

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 a small fast cache close to the processor. Caches exploit the spatial and temporal locality of memory references 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 load data can help in mitigating some of the unnecessary cache accesses. Such a mechanism can decrease the load latency and provide wider working set window to data prefetchers, increasing the throughput of the overall pipeline without increasing size of existing caches.

With a modest investment in hardware, 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 levels of the 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 [3] 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 [5] 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 [7] also tries speculative memory bypassing without the use of a store queue. Tyson and Austin's scheme [8] 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. [6] explores coalescing in GPUs using compilers and shows the potential for such a technique. [4] explores opportunities to carry out optimizations in decode time one of which includes combining multiple loads. Combining of loads is limited to a base register + offset type of loads and is dependent on a predictor. [9] proposes a way to skip tag checks to save energy.
We propose a novel approach which is not speculative in nature. Our approach triggers the fetching into the newly introduced coalescing buffer, post address generation of the load instruction that enters the pipeline. 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 data from multiple load instructions with the help of a hardware unit that fetches the entire cache block once and subsequently supplies data to incoming loads accessing the same cache block. This helps in servicing multiple load instructions provided there are no dependent store instructions in between them. According to our hypothesis, this will lead to an improvement in latency and energy savings due to the reduction in tag comparison when compared with the baseline out of order processor without an overhead of increasing the cache size.

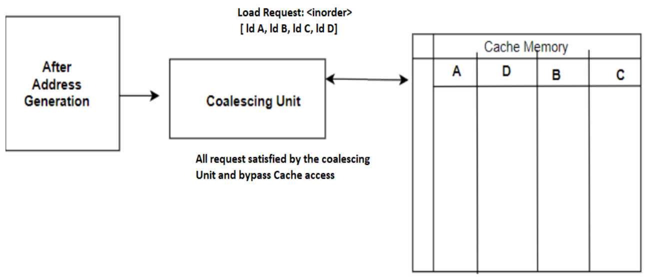
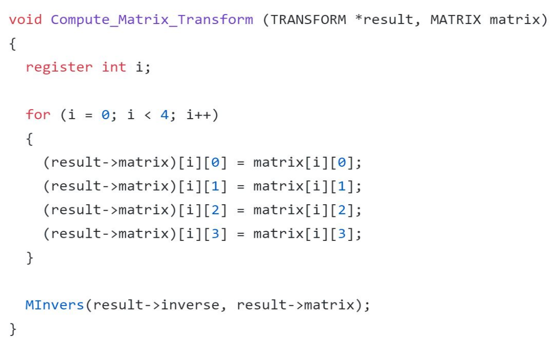

Load Request: <inorder>
[ld A, ld B, ld C, ld D]
After Address Generation
Coalescing Unit
All request satisfied by the coalescing Unit and bypass Cache access
Cache Memory
A
D
B
C

Figure 1: Load Coalescing Unit Overview

5 ANALYSIS

To support our hypothesis and motivation behind introducing a new 'Load Coalescing Unit', we analysed two applications/workloads from the SPEC_2006 benchmark suite namely: h246ref and povray.



```
void Compute_Matrix_Transform (TRANSFORM *result, MATRIX matrix)
{
  register int i;

  for (i = 0; i < 4; i++)
  {
    (result->matrix)[i][0] = matrix[i][0];
    (result->matrix)[i][1] = matrix[i][1];
    (result->matrix)[i][2] = matrix[i][2];
    (result->matrix)[i][3] = matrix[i][3];
  }

  MInvers(result->inverse, result->matrix);
}
```

Figure 2: SPEC_2006 Benchmark - povray Matrix operation code

The analysis carried out can be divided into two parts:

1. Observing if a load instruction (corresponding to the same PC) accesses same or different memory addresses.
2. Analysing memory accesses to the same cache blocks for different PC's (load instructions).

Each simulation was run for 10,000,000 instructions after fast forwarding 1 million instructions. The PC value was noted for each load instruction encountered during the run. Among the several load addressing modes of the x86 ISA, the load instruction of the following type was considered for the purpose of the analysis:

ld <reg>, «mem»

For the first strategy: The top 3 PC values, corresponding to the load instructions that had the maximum accesses to the memory during the simulation were picked and analyzed for their memory access pattern across time.

Further analysis of this access pattern was done to find out what kind of references were happening to a particular cache block and if there was a predictable stride in the access.

The below graphs (Figure2 and Figure3) show the reference pattern for the cache block size of 64. It was observed that majority of the references follow a kind of strided pattern and will benefit from our proposed scheme.

For the Second strategy, we carried out an analysis of the access pattern of load instructions to different cache blocks within a static instruction window of 160 instructions of the program. For an instruction window of 160 instructions, load instructions constituted about 10 percent of the instruction mix, which roughly comes up to around 10 load instructions on an average for the 160 instruction window considered.

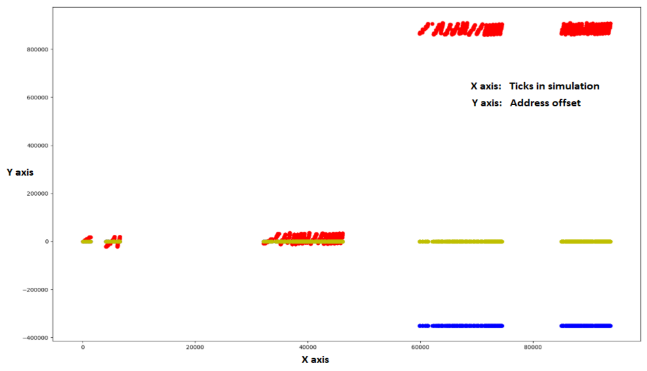


Figure 3: Memory access patterns for three PC values/Load Instructions (each shown with a different colour). Y-axis represents the offset in the address location being referenced and X-axis represents the time tick. The graph gives information of timeliness in the access pattern for a particular load instruction and also provides an insight into the addresses/cache blocks that are accessed most frequently.

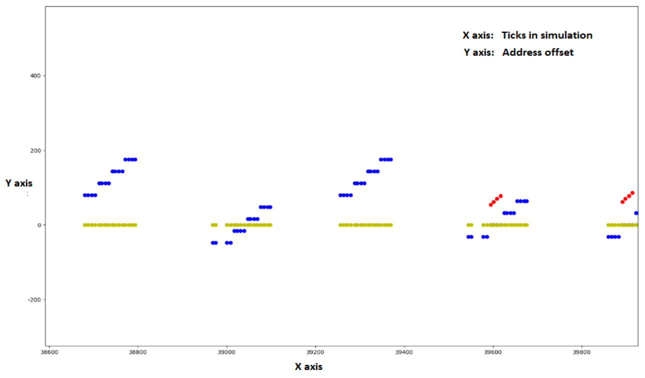


Figure 4: Zoomed in version of Figure 1. Y-axis represents the offset in the address location being referenced and X-axis represents the time tick. Here we can see the a better representation of the strided pattern of memory accesses for a particular PC (load instruction).

To determine if we could leverage load-coalescing among these loads within the instruction window, we calculated the number of load accesses across different cache blocks and also access to different words within the same cache block. For the L1 D-cache used in the DerivO3 CPU Model in Gem5, the cache block size is 64 bytes and hence masking out the last 6 bits of the generated load data-address gave us the number of the accesses to a particular cache block. The count for each word-access (within a cache block) was kept track of using the 6 offset bits of the load data-address.

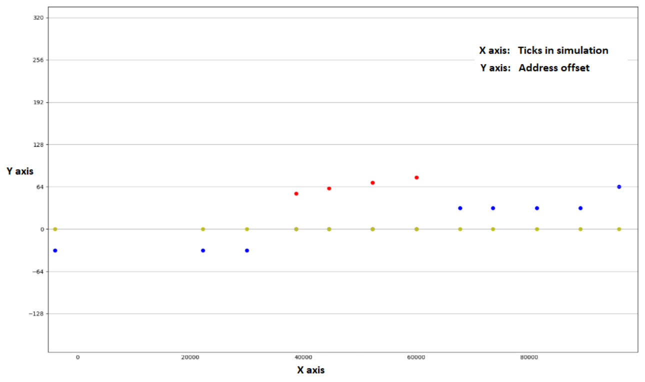


Figure 5: Zoomed in version of Figure 2 along with additional horizontal lines depicting a cache block size of 64. Y-axis represents the offset in the address location being referenced and X-axis represents the time tick. From this, we observe that a lot of accesses take place within the same cache block and also to the same words within the cache block.

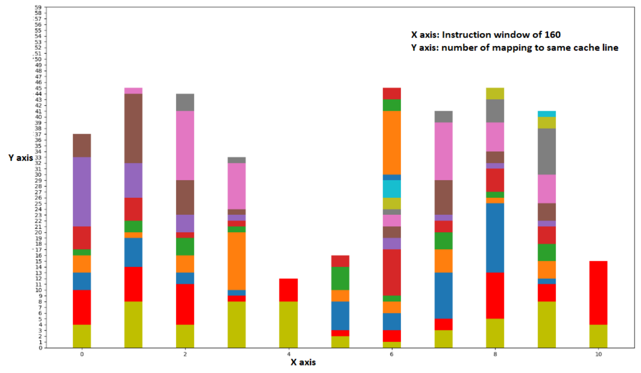
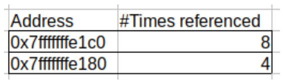


Figure 6: Plot of 11 instruction windows (each of 160 instructions) and the corresponding accesses to different cache blocks. Each bar in the plot is representative of number of load instructions within the 160-window size. A different color basically indicates access to a unique cache block and the height of the colored portion denotes the number of accesses within a particular cache block. X-axis refers to the each instruction window and Y-axis refers to the count of the access per cache block.

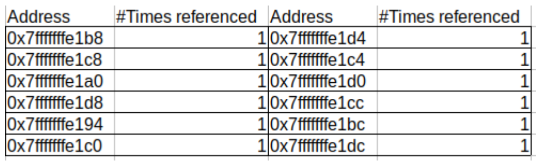
Taking the 5th bar graph (5th instruction window) as an example we show how the references to a cache block were calculated. The table below reflects the references to two cache block in the 160 instruction window cycle.

The references can be further divided within the cache block as shown in the below Figure 8.



Address	#Times referenced
0x7fffffffe1c0	8
0x7fffffffe180	4

Figure 7: Request in Cache Line



Address	#Times referenced	Address	#Times referenced
0x7fffffffe1b8	1	0x7fffffffe1d4	1
0x7fffffffe1c8	1	0x7fffffffe1c4	1
0x7fffffffe1a0	1	0x7fffffffe1d0	1
0x7fffffffe1d8	1	0x7fffffffe1cc	1
0x7fffffffe194	1	0x7fffffffe1bc	1
0x7fffffffe1c0	1	0x7fffffffe1dc	1

Figure 8: Request in different Cache block

6 PROPOSED SOLUTION

The preliminary analysis carried out for the two chosen benchmarks (h264ref and povray) highlight the potential performance benefit that can be achieved from load coalescing. From the typical load access pattern of programs in general, we also have sufficient reason to believe that our proposed solution would yield benefit for most of the applications, besides the two chosen ones in our analysis.

The main idea behind our solution is to try and reduce the number of accesses to the data cache; the load data requests would instead be serviced by a new hardware structure which we call the 'Load Coalescing Unit (LCU)'. This data structure holds entire cache blocks and their corresponding block addresses. The fetching into the LCU for the very first time is done when a load request accesses the same block (could be any word within the cache block) in the data cache.

To ensure that our implementation does not affect the current flow of the processor, we had to handle scenarios related to Load-Store Aliasing, Load Forwarding among several other things. We elaborate more on these in the Implementation section.

7 CURRENT IMPLEMENTATION

We have made the following changes to the Gem5 simulator in order to implement the proposed Load Coalescing solution[2]:

(1) A new coalescing buffer structure (named 'Load Coalescing Unit') that is indexed using the physical address of the block and stores the value of the entire cache block.
(2) A knob to change the size of the coalescing buffer or equivalently the number of cache blocks that the LCU holds at any given time.
(3) Different eviction policies like LRU, NMRU, Random and FIFO added in gem5 to evict the data in the buffer when it is full.

The Memory Data Flow is now slightly changed due to the added changes to the architecture:

(1) The address of any incoming load request coming to the O3 CPU (in the LSQ unit) or to the Atomic Simple CPU (during memory access) is checked for in the LCU to see whether the data to be fetched is already present in the buffer.
(2) Memory dependence detection and load store forwarding has also been handled for O3 CPU. In case, the data for the incoming load request is present in the store queue, we forward this latest data from the store queue.
(3) At the same time, in case of Load-Store Aliasing, where we have an intermediate store writing to the same cache block which is present in our buffer, we invalidate the corresponding block entry in the coalescing buffer to prevent incorrect/stale data from being present in the buffer.
(4) We also have a check for the time when the store commits; at this point we make sure that the load instructions requesting access from the block that was just written to is serviced with the correct data (by going to the memory to get the data and not through the LCU).

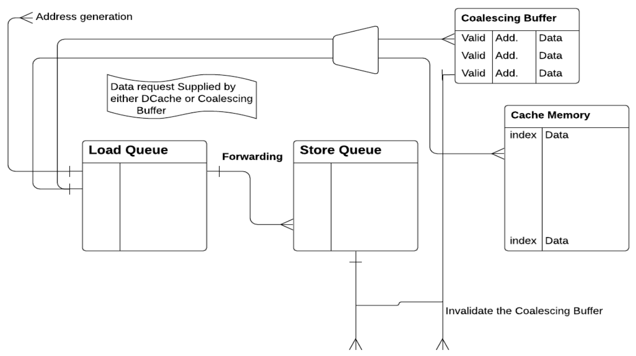
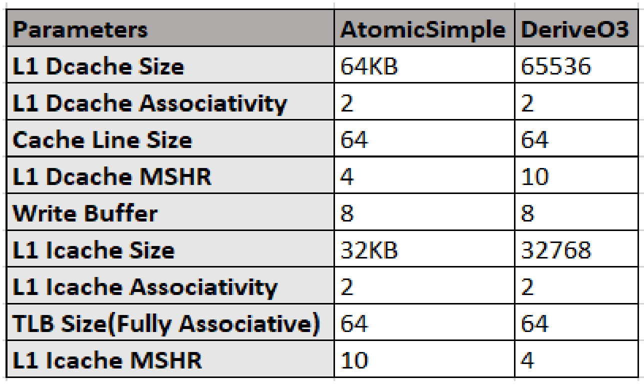


Figure 9: Load Coalescing Unit working

8 METHODOLOGY

Simulator: We use the gem5 simulator[1] to carry out all our simulations. We simulated a system with a single x86 CPU core with the configuration as shown in Figure 8.
Workload: We use 5 benchmarks from the SPEC2006 benchmark suite. We used the check-pointed benchmarks warmed up to 1,000,000 instructions and then running them for 10,000,000 instructions.



Parameters	AtomicSimple	DeriveO3
L1 Dcache Size	64KB	65536
L1 Dcache Associativity	2	2
Cache Line Size	64	64
L1 Dcache MSHR	4	10
Write Buffer	8	8
L1 Icache Size	32KB	32768
L1 Icache Associativity	2	2
TLB Size(Fully Associative)	64	64
L1 Icache MSHR	10	4

Figure 10: Parameters used for Simulation

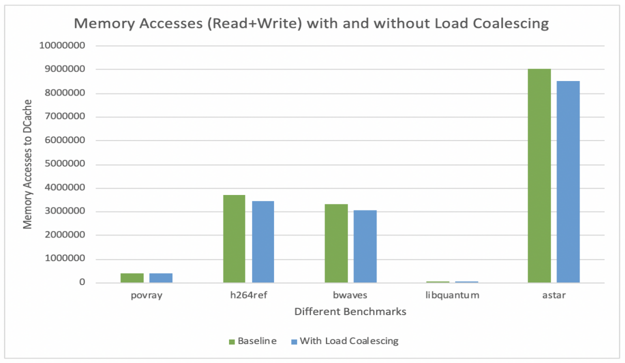


Figure 11: Memory Accesses (Read+Write) to DCache with and without Load Coalescing

9 RESULTS

To analyse the benefit that our proposed solution yields, we carried out a performance analysis of our architectural changes in gem5 by running several benchmarks for both the Atomic Simple and the Out of Order CPU. Here, we evaluated performance in terms of the reduction in the accesses to memory (L1 Dcache for our purpose). Each of the benchmarks mentioned below were run in gem5 by fast forwarding the first 1 million instructions and running the subsequent 10 million instructions.

POVRAY: 453.povray is an Image Ray-tracing Application. The test-case is a 1280x1024 anti-aliased image of a landscape with some abstract objects with textures using a Perlin noise function.

LIBQUANTUM: 462.libquantum is a Physics / Quantum Computing Application that simulates a quantum computer, running Shor's polynomial-time factorization algorithm.

H264REF: 464.h264ref is a Video Compression Application which includes a reference implementation of H.264/AVC, encodes a videostream using 2 parameter sets. The H.264/AVC standard is expected to replace MPEG2

BWAVES: 410.bwaves is a Fluid Dynamics Application that computes 3D transonic transient laminar viscous flow.

ASTAR: 473.astar includes Path finding library for 2D maps, including the well known A* algorithm. 473.astar includes Path finding library for 2D maps, including the well known A* algorithm.

9.1 Cache Accesses

Figure 11 shows the data cache memory accesses in each of the benchmark for the simulation period. As expected, benchmarks having regular cache line accesses have significantly reduced data cache accesses as the demand for the same is satisfied by the coalescing buffer.

This includes both the direct and indirect memory accesses as the analysis is done in the runtime compared to older work on load coalescing [6]. For the benchmarks 'astar', 'h264ref' and povray we see that there is close to 9-10% reduction in the total accesses to the cache while for the certain benchmarks like 'libquantum' there is negligible reduction. This is because libquantum is a streaming application with very low cache hit rate. Consequently, we have very few loads being serviced from our LCU since there are very loads that actually hit in the cache for them to be picked up by our buffer.

The formula used to compute the percentage decrease in memory accesses is:

$$\frac{(MemoryAccessesinbaseline)-(MemoryAccesseswithCoalescing)}{(MemoryAccessesinbaseline)}$$

9.2 Read Requests

Figure 12 shows the decrease in the Read Requests to Data Cache across multiple benchmarks. The percentage decrease can be directly correlated to the reduction in Dcache accesses because the data requested is supplied by the coalescing unit.

We see that majority of the benchmark is benefited because of important fundamental pattern of memory accesses [spatial and temporal locality].

9.3 Coalescing Buffer Size

An experimental study was done to determine the optimal size of the load coalescing buffer as the size of the buffer will act as a important parameter affecting performance related to data forwarding from the coalescing buffer. The primary focus was to reduce the Dcache access and ensure that the demand is satisfied by the coalescing buffer. The experiment was performed by changing the buffer size incrementally - [2,4,8,16,64,128] - on different benchmarks. We evaluated the trade-off between the size increment and the percentage reduction in the Dcache accesses that was happening. We

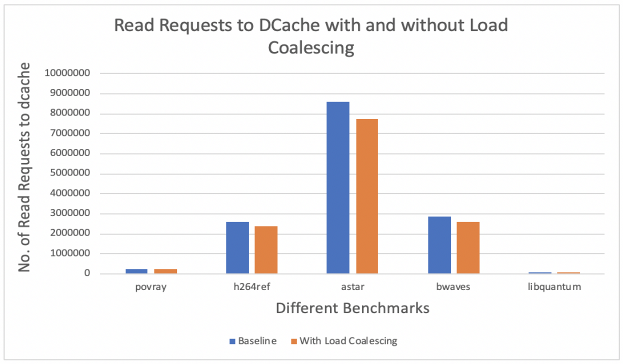


Figure 12: Read Requests to DCache with and without Load Coalescing

saw diminishing returns on increasing the size of the buffer slowly. Figure 13 and 14 shows the trend for two benchmarks with Y axis representing the number of memory accesses and x axis representing the buffer size. It is evident based on different benchmarks that the buffer size of 8/16 will be a good coalescing buffer size.

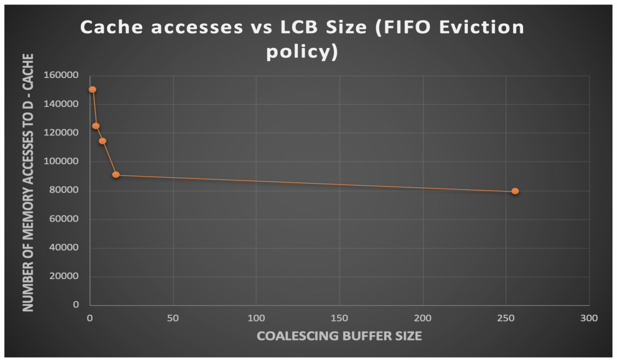


Figure 13: Analysis for 'povray' for different Eviction Policies and LCU Size

9.4 Eviction Policy

The experiment was performed with different eviction policies: LRU/NMRU/Random/FIFO. It was seen that LRU was the best eviction policy for the coalescing buffer. This can be reasoned since most of the memory accesses follow a sequential pattern and hence with LRU, we are able to service most of the loads from our buffer rather that from the memory.

9.5 Performance evaluation - AtomicSimpleCPU

In the AtomicSimpleCPU, we see a decrease in memory read accesses by approximately 70-80 percentage. We believe that the results from the Simple Atomic CPU indicate the maximum performance gain (in terms of the reduction in memory accesses) that we can get out of Load Coalescing.

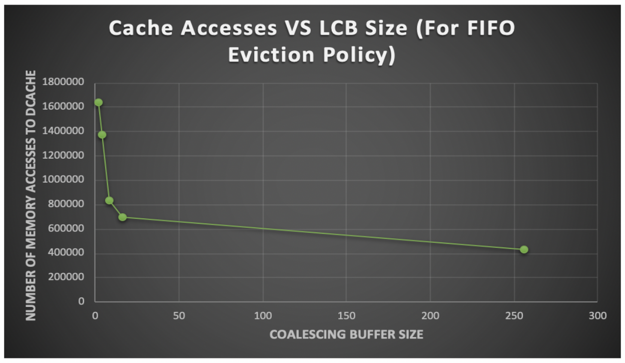
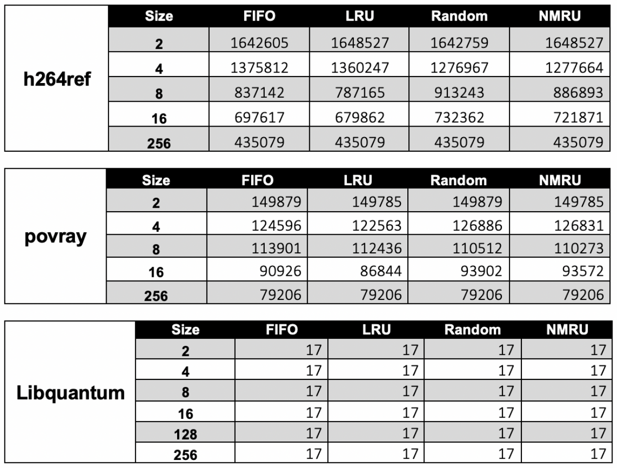


Figure 14: Analysis for 'h264ref' for different Eviction Policies and LCU Size

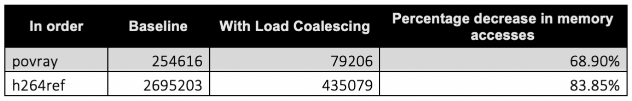


	Size	FIFO	LRU	Random	NMRU
h264ref	2	1642605	1648527	1642759	1648527
	4	1375812	1360247	1276967	1277664
	8	837142	787165	913243	886893
	16	697617	679862	732362	721871
	256	435079	435079	435079	435079

	Size	FIFO	LRU	Random	NMRU
povray	2	149879	149785	149879	149785
	4	124596	122563	126886	126831
	8	113901	112436	110512	110273
	16	90926	86844	93902	93572
	256	79206	79206	79206	79206

	Size	FIFO	LRU	Random	NMRU
Libquantum	2	17	17	17	17
	4	17	17	17	17
	8	17	17	17	17
	16	17	17	17	17
	128	17	17	17	17
	256	17	17	17	17

Figure 15: Variation of D-Cache memory accesses with Load Coalescing Buffer size and Eviction Policies

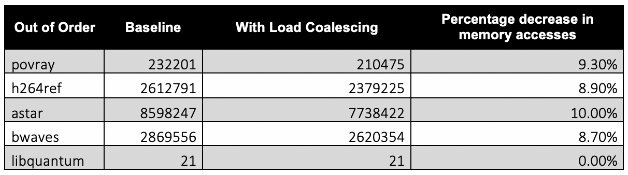


In order	Baseline	With Load Coalescing	Percentage decrease in memory accesses
povray	254616	79206	68.90%
h264ref	2695203	435079	83.85%

Figure 16: Performance Improvement for the Atomic Simple CPU

9.6 Performance evaluation - DerivO3CPU

In the Out of Order CPU, we see an improvement of memory read accesses by approximately 9-20 percentage. The results are conservative and we can expect with proper load replay policy the number of memory access can be further reduced. Hence, the 10% reduction is actually a lower bound on the maximum achievable performance gain from load coalescing in Out of Order CPU.



Out of Order	Baseline	With Load Coalescing	Percentage decrease in memory accesses
povray	232201	210475	9.30%
h264ref	2612791	2379225	8.90%
astar	8598247	7738422	10.00%
bwaves	2869556	2620354	8.70%
libquantum	21	21	0.00%

Figure 17: Performance Improvement for an Out of Order CPU

10 CHALLENGES

Gem5 simulator has a huge code-base and it was tough to start implementing and adding changes to it. Tracing the path of a load request and tracking all the dependencies in the O3 CPU model was challenging. Since we used cache block as the granularity for dependence, there were a lot of unforeseen issues that arose and were leading to segmentation faults in gem5.
Another challenge was to move the buffer to the register file. Since load requests vary in size from 1 to 8 bytes we were not able to effectively tackle the sub-word and multiple word dependency problem where the data that the load requires is either some part of one or a part of multiple registers which led us to skip this approach for the project.
Loads mapping to multiple cache blocks cannot be handled by our implementation and are skipped and follow the normal path.

11 FUTURE WORK

There are a lot of improvements that can be made the initial design in order to extract more performance. We can experiment with good techniques or a predictor to better be able to insert entries into the coalescing buffer which will help to avoid thrashing in the buffer. A good predictor can also help us perform optimizations at decode stage before waiting for the address to get generated and successfully take data from the new coalescing buffer. This will lead to even greater performance improvement given that the predictor is able to achieve good enough accuracy.
In our current implementation we track dependencies at a cache block granularity hence a store to the same cache block but a different offset will lead to invalidating the entire cache block. This dependence tracking is conservative and will lead to some lost opportunities. We can track dependencies at a finer granularity so a store at a different address but in the same cache block will not get invalidated due to this.
Another performance and area optimization to explore would be to use the register file instead of a new structure. We can reserve some registers for this optimization which will not be used during renaming. If the predictor is able to predict that the instruction will end up loading from one of the registers then we can have the rename logic to point at the particular register without actually issuing the instruction to the Load Store Queue.
Since we are targeting long latency instructions and they end up getting completed quickly and freeing up the cycle critical resources such as the Issue Queue we can also experiment with a smaller IQ and ROB to see if we can get the same performance with smaller structures which could lead to potential area and power savings.
Area and power exploration of this optimization can be studied further to find an optimal configuration of the load coalescing buffer.

12 CONCLUSION

Many past proposals have tackled the memory bandwidth problem at a coarser level by using data prefetching and efficient caching schemes. However, our research have attempted to optimize data fetching at a finer granularity by reusing the cache blocks. In this paper, we proposed coalescing of multiple load memory requests that can reduce the L1 Data Cache accesses. This proposed technique reduces the Load Memory accesses for SPEC2006 Benchmarks by around 80% in the simpler model. For an out-of-order CPU model, it has shown a reduction of around 10% in the memory reads, thus reducing the overall latency of a critical stage.

13 ACKNOWLEDGEMENTS

We would like to express our gratitude to Prof. Mikko Lipasti (University of Wisconsin-Madison) for sharing his pearls of wisdom with us during the course of this research. We are also immensely grateful to Prof. Gurindar Sohi (University of Wisconsin-Madison) for providing his invaluable insights on this research. We thank our colleague Shyam Murthy who greatly assisted the research during the project work. We would also like to extend our gratitude to the technicians of the laboratory of the Department of Computer Sciences and the Department of Electrical and Computer Engineering (University of Wisconsin-Madison) for their help in offering us the resources to carry out the research work.

REFERENCES

[1] Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., Hestness, J., Hower, D. R., Krishna, T., Sardashti, S., et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News 39*, 2 (2011), 1–7.
[2] https://github.com/prathameshpatel07/gem5_cs_752_project.

[3] 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.

[4] Kim, I., and Lipasti, M. H. Implementing optimizations at decode time. In *Proceedings 29th Annual International Symposium on Computer Architecture* (2002), IEEE, pp. 221–232.

[5] 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.

[6] Puthoor, S., and Lipasti, M. H. Compiler assisted coalescing. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques* (2018), ACM, p. 11.

[7] 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.

[8] 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.

[9] Zheng, Z., Wang, Z., and Lipasti, M. Tag check elision. In *2014 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)* (2014), IEEE, pp. 351–356.