ECE 752 Progress Report

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 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.

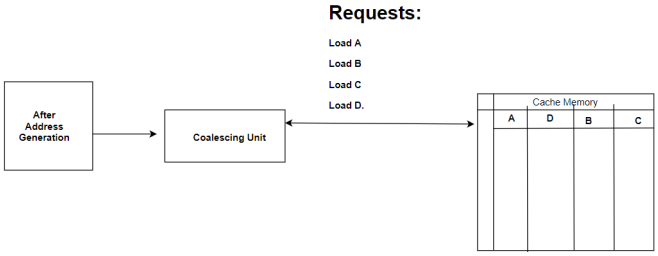
A data prefetcher can improve the utility of caches, by predicting what data will be used in the near future, fetching it from DRAM into the cache. But even with an optimal data prefetcher, sequential loads have to repeatedly access the caches block. These repeated cache accesses affect the performance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



4 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 libquantum.

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 1,00,000 instructions after fast forwarding 1 billion 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.

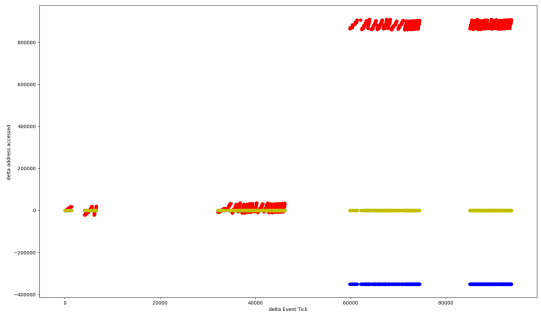


Figure 1: Memory access patterns for three PC values/Load Instructions (each shown with a different colour). Y-axis represents the offset in the address location being referenced and X-axis represents the time tick. The graph gives information of timeliness in the access pattern for a particular load instruction and also provides an insight into the addresses/cache blocks that are accessed most frequently.

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

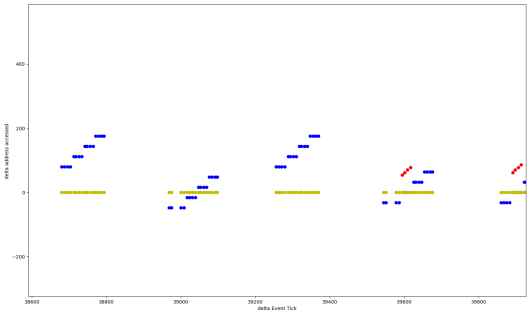


Figure 2: Zoomed in version of Figure 1. Y-axis represents the offset in the address location being referenced and X-axis represents the time tick. Here we can see the a better representation of the strided pattern of memory accesses for a particular PC (load instruction).

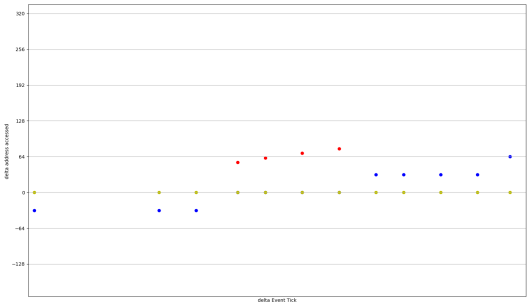


Figure 3: Zoomed in version of Figure 2 along with additional horizontal lines depicting a cache block size of 64. Y-axis represents the offset in the address location being referenced and X-axis represents the time tick. From this, we observe that a lot of accesses take place within the same cache block and also to the same words within the cache block.

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.

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 DerivedO3 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.

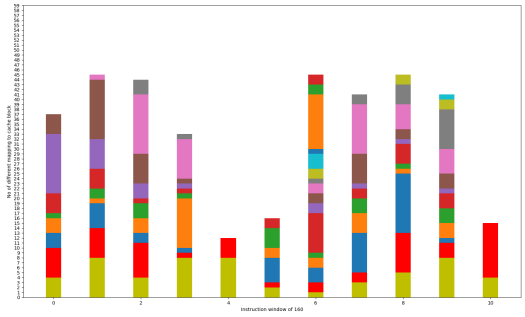


Figure 4: Plot of 11 instruction windows (each of 160 instructions) and the corresponding accesses to different cache blocks. Each bar in the plot is representative of number of load instructions within the 160-window size. A different color basically indicates access to a unique cache block and the 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.

Address	#Times referenced
0x7fffffffe1c0	8
0x7fffffffe180	4

The above references can be further divided within the cache block as shown in the below image.

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

5 GOALS AND CURRENT IMPLEMENTATION

We first plan to get a non-speculative approach working before moving directly to the speculative approach. In the non-speculative approach, after address generation, we check to see whether the requested block is present in the register file. If it is, then we just update the rename logic to point to the new register and skip the memory request. We have currently completed the following implementation in Gem5:

(1) Added a new buffer that is indexed using the physical address and stores the value of the cache block.
(2) Checking of incoming loads to see whether the data is present in the buffer.

We plan to incrementally make updates to our model in the following order:

(1) Determining a good trigger to insert a particular cache block in our new coalescing buffer.
(2) Handling memory dependencies and invalidating the entry in the buffer following a corresponding store at the same address.
(3) Coming up with a good eviction policy to evict the data in the buffer after some condition depending on time since last access.
(4) Try to allocate fixed registers in our register file instead of using a separate buffer so that the renaming logic becomes simple and we just need to modify the destination mapping to point to the new registers.
(5) Try to move to a speculative approach where at the decode stage a predictor determines whether load should proceed normally or take the data from the register file directly. Try and come up with a good predictor that has high success rate.
(6) Handle recovery in case of a misprediction by the predictor.

6 TOOLS

1. Literature Review:
In order to understand the opportunities of Load Coalescing in the context of CPU micro-architecture, we studied the techniques mentioned in the available research papers [1], [2], [3] and [4]. The ideas mentioned in the papers provided us relevant information to understand the scope of hardware mechanism to support Load coalescing. Further, we reviewed the traces generated by running standard benchmarks applications: a,b,c on the Out of Order CPU Model of gem5 simulator. The Load Instructions presenting the opportunities for coalescing were isolated after parsing the execution traces by using Python programming language. This exercise helped us to understand the process of research exploration in the domain of architecture design by analyzing the traces generated from the benchmarks.

2. Gem5 Simulator:
To move forward with our implementation plan to support Load Coalescing functionality, we studied the CPU O3 Model of the Gem5 simulator on x86 ISA. This activity involved learning the implementation of various pipelines stages of Gem5 simulator which is heavily based on the interaction of

various C++ objects internally referred as SimObjects. Understanding the functionality of each of the units provided us the relevant information to write the code of our implementation into the existing SimObjects. Unlike other standard blocks like Branch Predictor where a standard framework is provided to insert the implementation, the Load Coalescing functionality needs to be embedded within the CPU model files. Thus, we had to understand the functioning of various blocks in detail to incorporate our changes in the existing architecture. Primarily, we are concerned with modifying the files in the Load Store Queue (LSQ) and the existing Re-order Buffer (ROB) units in the CPU model to achieve our initial goal. Later, for our future goals we plan to overwrite few methods in the Renaming and Decode stage to optimize the performance further.

3. Git Version Control:
In order to achieve our goal of efficient project development, we are utilizing the Git Version control tool for our implementation. Using a popular and convenient distributed Version Control System (VCS) tool like Git is helping the involved group members to familiarize ourselves to work in an environment with multiple contributor. The tool also provides us additional service of providing backup and remote server access to our code.

4. Gdb usage and learning:
The study involved understanding the program structure and using gdb as the method of debugging the problem. Several gem5 functions are designed to be called from gdb. Functions like dumpDebugStatus(), SimObject:find() were used for the debugging purpose. Stand alone gdb analysis for different program was also done to see the reference pattern and dump the assembly level implementation of the program. It helped in getting more insight into as to how the PC value for different load are referenced and was helpful in the early stage of debugging the small workload that was used to observe the memory pattern references.

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] 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.

[3] 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.

[4] 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.