Stream Buffers for Effective Prefetching in the RISCV Processor

By Jonathan Terry

Project Objective

Whenever we talk about the performance of computers, we are concerned with both the efficiency as well as the raw speed. In this project, I sought to improve processor performance by increasing the efficiency of computation. As we are well aware, a memory hierarchy exists in computers in a trade off of size and speed. Despite this inherently necessary design, there exist bottlenecks between the different memories which cause a reduction in computer efficiency. In lab 6, we added a Level 1 Cache to reduce the latency discrepancy between DRAM and registers. That said, everything that exists in the cache must be first acquired from DRAM, and this can still cause a significant decrease in processor efficiency. My take on how to improve the performance of the existing RISCV processor of lab 6 was to implement a device to prefetch words from memory in an intelligent and predictive manner to aid in the filling of the cache. The efficacy of prefetching stems from the likelihood that other non-memory operations can be done in the interim, at least partially masking the memory access operations.

The way by which I chose to explore the realm of prefetching was through the implementation of a device called a Stream Buffer, first proposed and studied by Norman Jouppi in 1990. This device acts like an aid to the cache, capable of prefetching a certain number of data words in a certain pattern to present to the cache before the cache even requests it. The hope is that while the processor isn't making memory accesses, the Stream Buffer is busy interacting with DRAM so that the cache seldom has to. In developing and testing this system, the FPGA is an obvious choice because we are testing a device that one would hope to realize in computer hardware. In terms of performance improvements, we hope to see a mitigation in cache interactions with DRAM and as such a reflection in the processor IPC.

Background

As previously stated, the Stream Buffer lives in limbo, alongside a cache and interacting with lower memory. In the event that there are multiple caches, Stream Buffers may fetch from either lower caches or DRAM depending on the architecture. Furthermore, since both data memory and instruction memory may benefit from prefetching, you may find the Stream Buffer in either type of cache. In either case, their goal is the same: to fetch data words from memory before they would be requested normally, thus improving processor performance.

At its core, the Stream Buffer is a relatively simple FIFO data structure, with each slot in the FIFO containing a memory address identifier, an available bit, and the cache line associated with the memory address. Upon a cache miss, the head of the Stream Buffer (i.e. the first element) has its address compared to that of the address that missed in the cache. If the address matches and the cache line is available, the cache line is replaced as normal. If the address is to be there, but is not yet available, you wait until it is available, since no matter what this will in the worst case be neutral. Should a cache line be popped from a FIFO, the vacancy at the end of the FIFO will be filled by another cache line as determined by the Stream Buffer. Finally, in the unfortunate case that the Stream Buffer does not have the cache line we want, the cache then has to interact with DRAM directly.

Now that the basic operation of how one interacts with a Stream Buffer are out of the way, we should ask ourselves how the Stream Buffer determines what to fetch. The most naive Stream Buffer will, upon there being a single cache miss, begin allocating sequential cache lines until the FIFO is filled. While this is very appropriate for the instruction cache, it may not be so appropriate for the data cache. If the goal of the Stream Buffer is to prefetch data, and say you are doing something like a matrix multiplication, then the elements you are accessing may lie several cache lines apart from each other. Therefore, in order to make the stream buffer more efficient, one can create predictors to determine what the Stream Buffer fetches and when.

Finally, something to consider is when the Stream Buffer is invoked. For instance, certain memory access patterns may not warrant an entire Stream Buffer. Tangentially related to this question may be how many Stream Buffers exist in a given cache. When you start adding additional Stream Buffers, then a question of an allocation policy comes into play. When does a Stream Buffer get acquired, and if all are taken, which one gets the axe. All of these questions were explored in my design, and naturally I made the design of my modules quasi-parametric (used typedefs) in order to make the exploration easier.

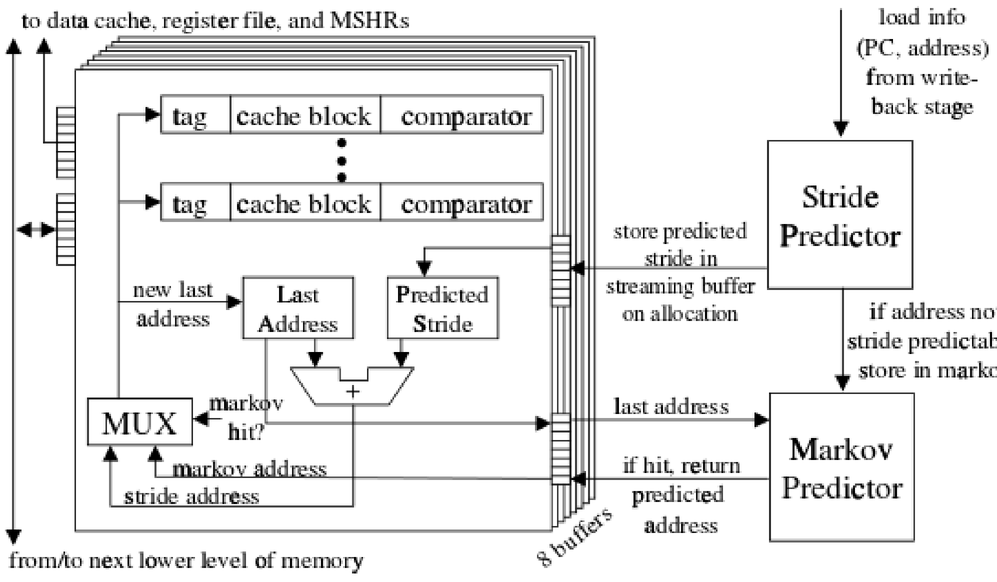
to data cache, register file, and MSHRs
tag
cache block
comparator
tag
cache block
comparator
new last address
Last Address
Predicted Stride
+
MUX
markov hit?
markov address
stride address
from/to next lower level of memory
8 buffers
load info (PC, address) from write-back stage
Stride Predictor
store predicted stride in streaming buffer on allocation
if address not stride predictable, store in markov
last address
Markov Predictor
if hit, return predicted address

Figure 1: Diagram showing the general architecture of a set of Stream Buffers, highlighting the structure of the FIFOs as well as how they interacts with lower memory. Further, it shows the addition of predictors to aid in the prefetching process.

High-Level Design

The original high-level design of the stream buffer was to create a Stream-Buffer module, a Predictor module, and update the Cache module to support a Stream Buffer. Ultimately, the line between the Cache and the Predictor were blurred, and moreover, it turned out that I needed to implement a Memory Controller in order to delegate requests and responses to the DDR3 Memory from both the Stream Buffer and possibly the Cache simultaneously. The final design that I converged on was, at a system level, as seen in the figure below:

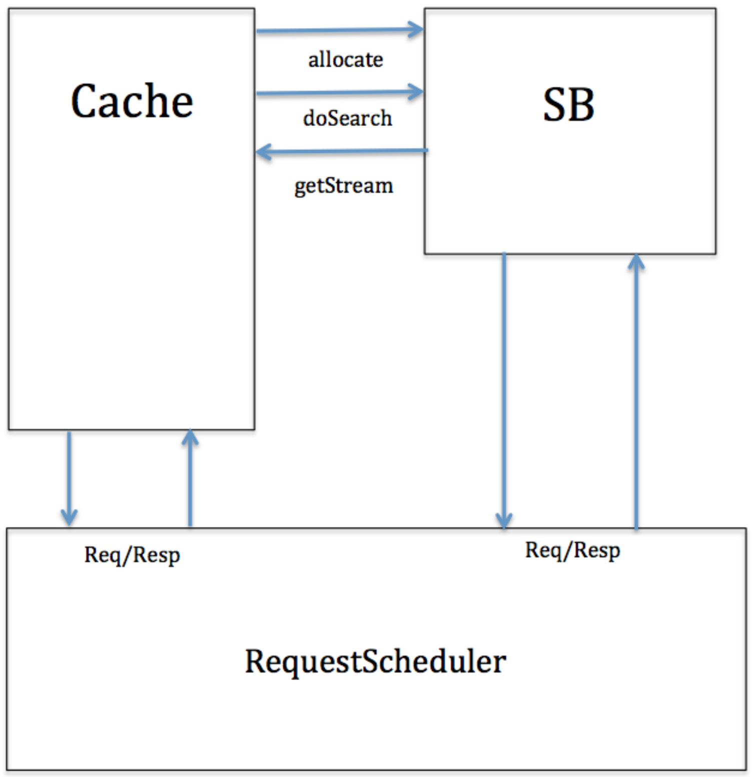

Figure 2: Diagram showing the high-level design of the Stream Buffer System that I designed.

In the high-level design, an important decision was how to design the API's of the respective units. After giving it thought, since I was going to have to redesign the Cache anyways, I decided to integrate the predictor into the Cache. As a consequence, this allowed the Cache to simply make a request to allocate a Stream Buffer starting with a certain address and a particular access pattern (also known as a stride). The Stream Buffer also exposes a search function to check for a stream hit, and likewise it also supports a return function. This seems like the best way to decouple the two, and allows for many interesting design decisions on the end of the Stream Buffer. Unbeknownst to the Cache, the SB may have only one FIFO or it may have 32. Further, replacement policies are completely internal to the Stream Buffer. This can allow for modular development and testing of different replacement policies and coherence protocols without needing to change the Cache.

Aside from the Cache and the Stream Buffer, an initially unexpected, but critical aspect of the design was the Request Scheduler, which was effectively a memory controller that could support requests from both the Cache and the Stream Buffer simultaneously. The Request Scheduler exposed two separate Request and Response functions, one for the Cache and one for the Stream Buffer, and was in many ways can be seen as a wrapper for the DDR3 Memory to support multiple concurrent requests.

Test Plan

The test plan for this setup was simple and yet effective. In order to get the most time possible for system design and hardware exploration, I decided to leverage the existing test infrastructure for lab 5 and lab 6, using the same benchmarks for the most part. In addition to the provided benchmarks, I created two additional benchmarks: one to test performance on loops with a simple stride and another to test constant non-sequential strides.

Microarchitecture

To discuss the microarchitecture, and to highlight the intended modularity of the system, it is worthwhile to discuss the three major components separately:

1. Streaming Cache (StreamingCache.bsv)
2. Stream Buffers (StreamBufferController.bsv)
3. Request Scheduler (RequestScheduler.bsv)

Streaming Cache

In order to get the Cache and the Stream Buffers to play nicely, I had to make some updates to the Streaming Cache. Namely, in the request function, I added the ability to both check a Stream Buffer module for hits as well as allocate a Stream Buffer according to a predictive pattern. This simply boils down to some added muxes. Furthermore, I was able to keep the existing direct mapped cache, which was effectively vectors of registers. With respect to the predictive model, I directly integrated it into the Streaming Cache, so that with a simple if/else clause one can invoke a Stream Buffer with a certain stride.

Stream Buffers

The bulk of the work in this project, the stream buffer, is at its core a simple idea, but in practice difficult to implement. Within the Stream Buffer were many statically elaborated rules as well as vectors of structures. As stated previously, the Stream Buffers are essentially FIFOs, so vectors of FIFOs were the main constituent of the microarchitecture. Alongside these FIFOs were registers storing information such as next address to fetch upon a stream hit, whether or not the stream has been allocated, and even the initial address that allocated the stream. The idea behind storing all of this information was to build a Markov model within the stream buffer to aid in non-linear prefetching. Additionally, many rules were statically elaborated based upon typedefs defining the structure of the stream buffer (such as the number of streams and the length thereof).

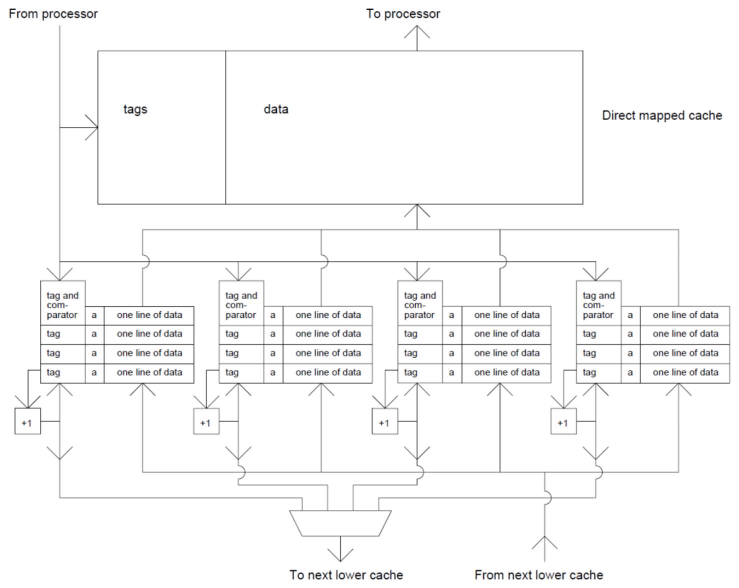

Figure 3: Diagram showing the architecture of the stream buffer, highlighting the FIFOs and the prospect of multiple running parallel.

Request Scheduler

The idea behind the Request Scheduler was to have two FIFOs which accepted requests from both the cache and the stream buffer, and made requests to the DDR3 memory. In the interim, receipts of these requests (given an arbitrary order) were then placed into another FIFO. Upon responses from the DDR3 Memory (which occurred in the order of the FIFO), the receipt was used to delegate the responses to the correct destination upon responses from the DDR3 Memory. This simple design works because DDR3 requests are serviced in order of request. Furthermore, in this design, BypassFifos were used everywhere in order to reduce overhead generated by the Request Scheduler.

Through the use of rather large input FIFOs, this allowed for multiple Stream Buffers requesting cache lines to be serviced at the same time without conflict. The general structure of the receipts used in the middle FIFO ensure that the requests are returned to the proper stream.

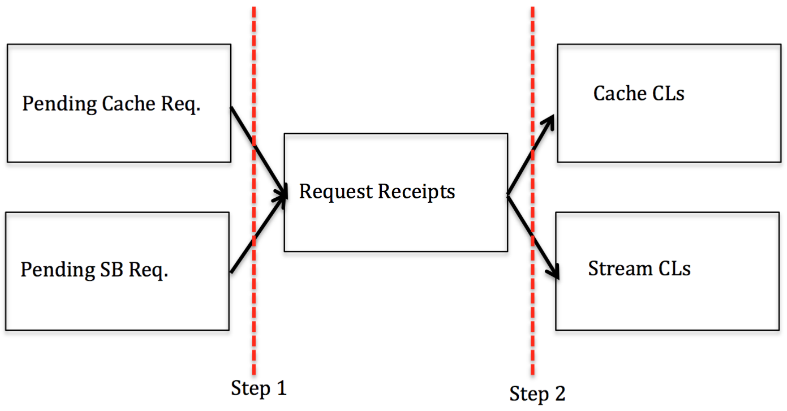

Figure 4: Diagram of the Request Scheduler, with each rectangle representing a FIFO. In step 1, DDR3 Requests are made, and a receipt is passed into the middle FIFO. In step 2, responses from DDR3 are combined with the receipt and shipped to the correct destination.

Implementation Evaluation

Discussion of Challenges

In implementing the Stream Buffer unit, there were several challenges, although almost all of them were conquered. In the beginning, the hardest part was figuring out which algorithms to use and how to recreate Jouppi's original paper. Literature was vague as to the details despite promising that it would work, and that it would work well. Given the sound theory I charged ahead, tackling the problems as they came knowing there would be a light at the end of the tunnel. Luckily, the one area that I did not run into trouble was the synthesis and testing on an FPGA.

The unforeseen big problem that I encountered was the memory request scheduling issues, but luckily that problem was rather easily solved as soon as I realized the nature of requests lent itself to the simple solution I discussed in the previous section on the microarchitecture. Despite one weird rule firing when it wasn't supposed to (which took forever to track down), I got this working just fine. From the standpoint of changes, the addition of this scheduler was one of the biggest.

Although it may seem somewhat silly, one thing that I struggled with was developing a good, modular API for each of the modules. At first, I was keeping the predictor inside the Stream Buffer, but that was requiring too much exposure of the internal workings to the Cache. I eventually settled on combining the prediction policies with the cache to make things cleaner and simpler. Furthermore, I delegated the specifics of buffer reallocation to the stream buffer, which allowed me to unclutter the cache and make it easier to understand.

Results of Implementation

Overall, the addition of a Stream Buffer was a success, and resulted in improved processor performance. The total project ended up being roughly 400 lines of bluespec, 100 lines of additional tests, and a little over 120 hours of work. In this design, I made extensive use of existing IP and infrastructure, including lab6, with my main reused components (outside of the processor) being the special Fifos, the DDR3 Memory, and existing interface designs. However, to make the DDR3 Memory more realistic, I added a delay factor so that memory requests took about 100 cycles

As a result of the implementation, I saw that even with a length one stream buffer, I was getting an average performance increase of 15 percent above the baseline (cache only) IPC and it saturated at about 20 percent with a stream buffer length around 4ish for the assembly tests. This recreates the original paper very well, although the exact metrics don't exactly line up. The main difference in this setup and Jouppi's original paper was the block size of the DM Cache (ours is 16, his was 4). If one was to dilate the curve by a factor of 4, then the trend matches almost exactly. That being said, the stream buffer does not fair as well on the 'real' bench marks. On average getting a mere 5 percent increase, except for the strided access test which was much higher.

Additionally, in my implementation, I was able to get a two-delta stride detector working. In the tests where this would be useful (i.e. a non sequential cache line stride) it showed greatly improved performance. The one thing that I was not able to accomplish was the Markov Predictor. This didn't happen in part due to how vague the literature was, and moreover it would have been very messy and ruin the modularity to get it to work. Since this was a lofty reach goal, I am not too upset that it didn't work. Despite this setback, I was still able to meet my initial project goals of improving processor performance.

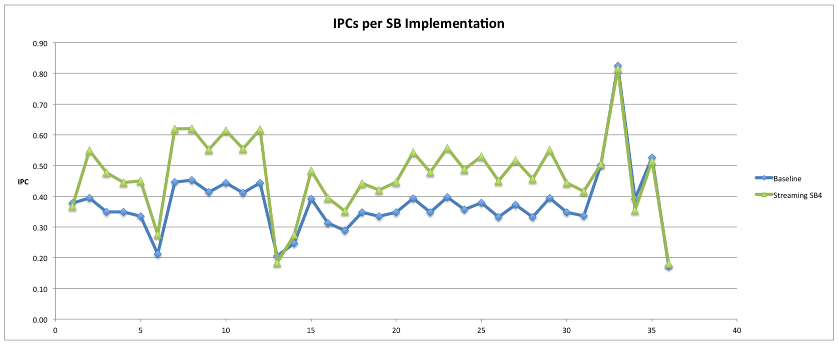

Figure 5: Graph of assembly benchmark performance after adding stream buffer of length 2.

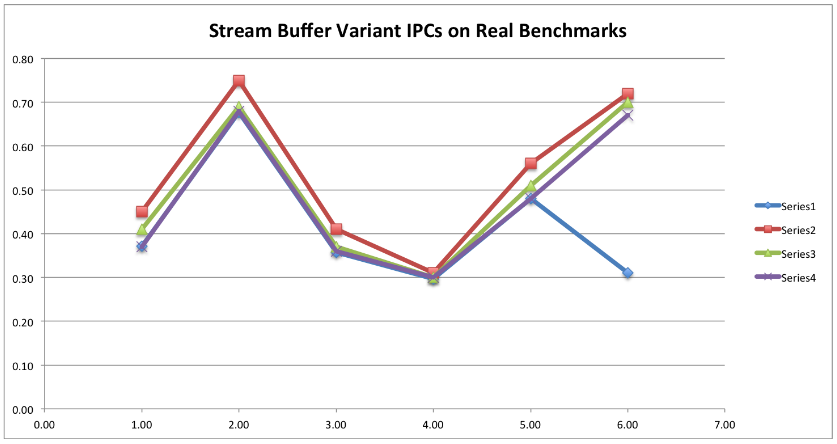

Figure 6: Graph of real benchmark performance after adding stream buffer of length 2.

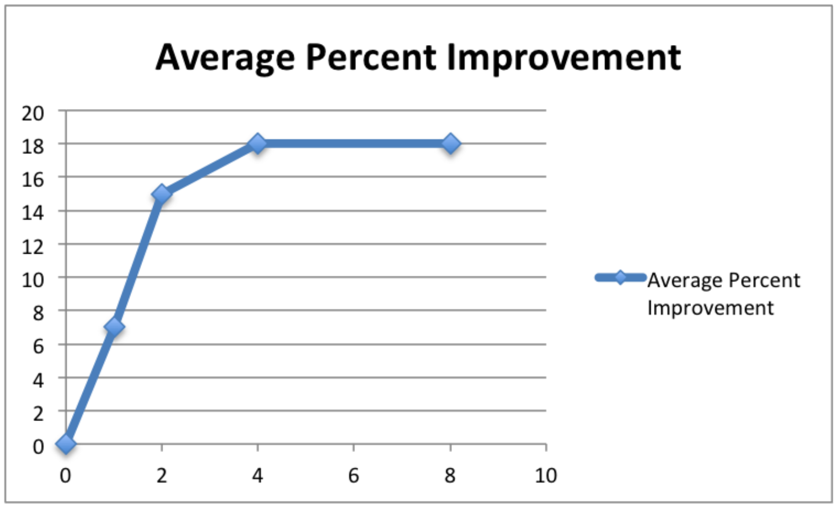

Figure 7: Plot showing the average percent increase in performance as a function of stream length. Note that this closely recreates Jouppi's results when accounting for block size.

Design Exploration

In terms of the design, I would be most interested in observing variations in replacement policy as well as seeing if the Markov Predictor is actually feasible, or if its touted success is that of a few carefully chosen benchmark results. In my project, I did successfully explore both the stream length as well as number of streams. I found that the number of streams had little effect, but I definitely could see my reallocation policy being the problem (FIFO instead of LRU). Furthermore, an interesting exploration would be in developing FIFOs that can have data modified at a given index. This could make cache coherency less of an issue when having a stream buffer in the data cache. Finally, another exploration of policy could be in prediction and allocation of the stream buffers. For instance, it would be cool to do some real research into what goes into a good Markov model. Furthermore, it would be neat to implement this on a real processor, in order to stress test its performance through running an operating system.