THE EFFECT OF OUT-OF-ORDER EXECUTION ON THE CACHE HIERARCHY

MICHAEL OTT

Supervised by Dr. CALVIN LIN
This document has been approved by my advisor for distribution to my committee.

A thesis submitted to
the University of Texas at Austin
for the Turing Scholars program

Department of Computer Science
College of Natural Sciences
University of Texas at Austin
May 2020
ABSTRACT

Out-of-order execution is used in almost all modern processors due to its ability to vastly improve performance. However, out-of-order execution causes disorder in the cache hierarchy, which means memory accesses will appear out-of-order. We look to identify the extent of this disorder along with its effects on prefetcher performance. Our study on disorder looks into three main areas. First, we look at how much disorder changes in the cache hierarchy through each level of the cache. Second, we look at how varying the reorder buffer size affects disorder. Third, we look at the effect of this disorder on prefetcher performance by comparing prefetcher performance on cores with varying levels of disorder.

First, our study of the cache hierarchy finds that disorder drops significantly when moving to cache levels further from the processor. In particular, on average we see a 53.6% increase of instructions that run in program order from L1D to L2C. Second, from our study of the reorder buffer we find that decreases in reorder buffer size also cause a small decrease in disorder, with a 1% increase on average in the L1D for instructions that run in program order. Third, from our study of prefetchers we find that some prefetchers are sensitive to this disorder and can be negatively impacted by it. Variable Data Length Prefetcher was the most sensitive to disorder of the ones we tested, with an average decrease in coverage by 11% between in-order execution and out-of-order execution.

ACKNOWLEDGMENTS

I would like to acknowledge my mentor Calvin Lin for his support. I would also like to acknowledge Akanksha Jain for her expertise and advice for this project. Finally, I want to thank Anurag Choudhury for being my research partner throughout the entire project.
# Table of Contents

<table>
<thead>
<tr>
<th>Article</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2 Background and Related Work</td>
<td>1</td>
</tr>
<tr>
<td>2.1 Background</td>
<td>1</td>
</tr>
<tr>
<td>2.2 Related Work</td>
<td>4</td>
</tr>
<tr>
<td>3 Methodology</td>
<td>7</td>
</tr>
<tr>
<td>4 In-Order ChampSim Implementation</td>
<td>8</td>
</tr>
<tr>
<td>5 Results</td>
<td>9</td>
</tr>
<tr>
<td>5.1 Disorder Between Cache Levels</td>
<td>9</td>
</tr>
<tr>
<td>5.2 Disorder at Different Reorder Buffer Sizes</td>
<td>10</td>
</tr>
<tr>
<td>5.3 Effect of Disorder on Prefetching</td>
<td>11</td>
</tr>
<tr>
<td>6 Conclusion</td>
<td>14</td>
</tr>
<tr>
<td>7 Future Work</td>
<td>15</td>
</tr>
<tr>
<td>References</td>
<td>16</td>
</tr>
</tbody>
</table>
1 Introduction

Out-of-order execution and data prefetching are two major techniques used in increasing processor performance. Typically, these two techniques are considered separately from each other. However, out-of-order execution causes disorder in the cache hierarchy, which occurs when out-of-order execution causes memory requests to reorder (Jaleel, 2006). This disorder can negatively impact prefetcher performance.

Our goal is to find the extent of disorder in the cache hierarchy as well as how much it affects prefetcher performance. To do this, we first create an in-order core for a simulator, which we use for our data collection on disorder. Second, we look into the extent of disorder at each cache level. Third, we look into the effect of increasing the amount of potential parallelism on disorder. Fourth, we look into the impact of disorder on prefetcher performance. We find that disorder decreases substantially when moving to higher cache levels, but only slightly when increasing the potential parallelism. We also find that some prefetchers are negatively impacted by disorder much more than others.

2 Background and Related Work

2.1 Background

In this section, we go over background knowledge about out-of-order execution and data prefetching. Knowledge of these is important for understanding disorder and existing prefetchers, which we will talk about in the related work section.

Out-of-Order Execution

Almost all modern processors use out-of-order execution because it is one of the key techniques for improving processor performance. However, prior to out-of-order execution, processors used a simpler model known as in-order execution. In-order execution works by
executing all instructions in program order (Hennessy and Patterson, 2007, p.90). Each
instruction is fetched into the pipeline in program order, and each instruction will continue
to maintain this order as it moves through the pipeline until it finally leaves after writeback,
where the result of the instruction is actually seen.

While in-order execution may seem logical, it creates some inefficiencies when dealing
with instructions that are independent from one another. Take for example a case where
there are three instructions, A, B, and C in program order. Let’s say that B is dependent on
the result of A, but C is independent of the other two. In an in-order processor, these would
execute in the same order regardless of this independence, meaning that instruction C would
have to wait for instruction B to execute before executing. This clearly wastes cycles, since
C could execute at the same time as A or B, allowing the processor to save cycles instead of
waiting. This idea is the basis for out-of-order execution.

Out-of-order execution exploits the independence of certain instructions, allowing for
these instructions to be reordered. When combined with superscalar processing, these in-
structions that would normally have to wait are able to run simultaneously (Hennessy and
Patterson, 2007, p.90-91). This enables the processor to not waste cycles waiting needlessly
on instructions. Out-of-order execution is capable of reordering instructions with the use
of a few tools, one of which is the reorder buffer. The reorder buffer which maintains the
proper order of instructions as well as other important information about the instruction.
Instructions can be dispatched out-of-order to functional units once its operands are available
instead of only dispatching in program order. After this, instructions will commit in-order

Looking at the previous example, we can see how out-of-order removes the previous
inefficiency. This time, A, B, and C will be fetched into the reorder buffer. From here, A
and C can be issued at the same time to functional units, and therefore executed at the same
time. After A is finished, B can be issued to its functional unit to execute. In this situation,
C no longer has to wait for B to finish, allowing the processor to finish in the time it takes
to execute two instructions instead of three.

This example can be used to show how instructions can reorder as the reorder buffer size increases. In the previous example, all three instructions were able to fit into the reorder buffer. This allows for instruction C to show up anywhere in the ordering. However, if for example the reorder buffer size was one, then the resulting instruction order would be identical to an in-order run. If the reorder buffer size was two, then instruction C wouldn’t be able to fit into the reorder buffer until A had already left, meaning that C would have to follow A in the resulting instruction ordering. In general, increasing the reorder buffer size allows for more possible orderings of instructions, and therefore should cause an increase in disorder.

**Prefetching**

Memory instructions are one of the biggest causes of slowdowns due to their high latency when accessing memory. One of the methods for combatting this latency problem is the use of a cache hierarchy. The idea behind caching is to move only the most important memory into smaller hardware closer to the processor to reduce the latency required to access it (Hennessy and Patterson, 2007, p.288). This however doesn’t completely solve the issue of memory latency. To make sure the cache is quick to access, it must also be much smaller than main memory, which means that it sometimes can’t hold all of the important data. Instead, the cache will evict elements based on a replacement policy once new data needs to be added. Cache lines that might be important in the far future might be evicted because of data that is needed in the near future. Even with a perfect replacement policy, there is always a possibility of missing in the cache and having to take a performance hit.

Prefetching attempts to address this problem by predicting which memory addresses will be needed in the future. By predicting which memory addresses will be needed and fetching them into the cache ahead of time, the latency caused by the memory access can be completely hidden by overlapping execution with data prefetching (Hennessy and Patterson,
Unlike with a cache replacement policy, a theoretically perfect prefetcher would completely remove latency caused by memory accesses.

## 2.2 Related Work

In this section, we go talk more about the prior work on disorder and frame our work in context of this prior work. We also explain the basics behind the existing data prefetchers which we will be studying.

### Disorder

The concept of disorder in the cache hierarchy has been studied before. Prior research by Jaleel has shown that out-of-order execution can negatively impact the memory system. Increased aggressiveness in instruction scheduling can cause increased cache misses and replay traps that reduce performance. Jaleel also introduces a method of measuring disorder in the cache hierarchy called absolute disorder. This metric is defined as the degree by which memory instructions are issued out-of-order with respect to actual program order (Jaleel, 2006). For each instruction, the absolute disorder is the difference between which order it appears in the out-of-order run and in program order. For example, take instructions A, B, and C in program order. Assume that in the out-of-order run, we see A, then C, then B. The absolute disorder of A would be 0, because it occurs in the same order as program order. The absolute order of B would be 1, because it occurs one spot later than program order, and the absolute disorder of C would be -1, because it occurs one spot earlier.

Using this metric, we can quantify how much out-of-order execution affects the memory system. His findings showed that increasing the reorder buffer size increased the absolute disorder in the cache (Jaleel, 2006). We intend to perform a similar study over many benchmarks to see if we get similar results. Our research will also look at prefetcher performance instead of cache performance to see if we get a similar correlation between increased disorder.
and decreased performance.

**Studied Prefetchers**

Since prefetchers rely on previous cache accesses to predict which memory addresses will be needed in the near future, disorder in memory accesses caused by out-of-order execution could negatively affect prefetcher performance. Because some prefetchers are more reliant on memory access order, we expect to see differences in performance based on this disorder caused by out-of-order execution. The three prefetchers we chose to study for this are the Best Offset (BO) (Michaud, 2016), Irregular Stream Buffer (ISB) (Jain and Lin, 2013), and Variable Length Data Prefetcher (VLDP) (Shevgoor et al., 2015).

Best Offset prefetcher builds on a simple prefetching technique called offset prefetching. The idea behind offset prefetching is to trigger a fetch anytime there is a miss at the previous cache level, or a hit on a prefetched line. The offset prefetcher will then fetch the address plus some offset. BO improves this by training the prefetcher to find the best offset for a given program, hence the name. A large enough offset can counteract the effects of disorder because it allows for more room for the accesses to be out of order (Michaud, 2016). For this reason, we expect BO to be very insensitive to cache disorder.

Irregular Stream Buffer prefetcher targets irregular memory accesses. In many other prefetchers, there is an assumption of physical memory address proximity being directly correlated to related memory accesses. ISB also looks at temporally related accesses by using a structural address space. This structural address space holds correlated memory accesses consecutively. This allows it to store irregular memory access streams as structural sequences. Whenever there is an access to the cache, the next structural addresses in sequence can be prefetched (Jain and Lin, 2013). Because the prefetcher only looks at a single trigger memory access, it is unlikely for disorder to negatively affect this. Moreover, since it can prefetch multiple addresses that succeed it, it can be more robust against disorder. For these reasons, we expect ISB to be insensitive to cache disorder.
Variable length Data Prefetcher uses deltas, which are the differences in addresses of successive memory accesses. VLDP stores delta histories, and uses them to predict future accesses by matching the recent deltas to patterns it has seen before (Shevgoor et al., 2015). Because the delta patterns are based on the order of requests to the cache, disorder in these requests can cause the prefetcher to not properly find a match, or even record incorrect patterns to match to in the future. On top of this, VLDP also only looks at the 3 most recent deltas, which could cause it to not predict far enough into the future, giving less room for it to be robust against disorder. For these reasons, we expect VLDP to be sensitive to cache disorder.
3 Methodology

For all of our data collection, we use the ChampSim simulator. ChampSim is a simulator designed for microarchitecture study, which makes it a good fit for our research. ChampSim is also convenient for us to use because each of the prefetchers we are studying have code written for this simulator. We will also be using the benchmarks from SPEC 2006. SPEC benchmarks are widely used in performance studies because they are useful for stress testing the processor and memory subsystem.

The first goal of our research is to find how much disorder varies when moving through different levels of the cache hierarchy. To do this, we will find the disorder at the L1D, L2C, and LLC levels for each benchmark using the absolute disorder metric mentioned previously. To be able to calculate the disorder metric, we need to analyze the memory instruction requests at each level of the cache. We do this by modifying ChampSim to print out the instruction id of memory instructions whenever a cache receives a request. We then run the simulator both in-order and out-of-order on 1 billion instructions with no prefetchers. The in-order run gives the instructions in program order, which we can then use to get the disorder from the out-of-order run. Any memory accesses not in both due to memory coalescing are excluded from the disorder, since we are unable to calculate its disorder.

The second goal of our research is to find how much disorder varies when increasing the reorder buffer size. This is simply an extension of our first area of study. We repeat the same steps as before, but vary the reorder buffer size, testing sizes of 128, 256, and 512. The third goal of our research is to find the extent to which disorder affects each of the prefetchers we have chosen to study. Each of these prefetchers is written for the L2C, so we don’t run them at each cache level. Instead, we run each of these prefetchers at each of the reorder buffer sizes in the out-of-order simulator, and once in the in-order simulator.
4 In-Order ChampSim Implementation

The current version of ChampSim only implements a simulator for an out-of-order processor. For this project, we modify ChampSim to also be able to simulate an in-order processor. Creating this in-order core took the bulk of the work for this project because we had to keep the interface with the cache hierarchy working properly. This required a lot of time spent understanding the code, and then carefully making the changes while checking for correctness at each step. At each stage of the newly implemented pipeline, we use a mix of manual debugging and sanity checks to validate it. We also create very simple benchmarks to verify that the output predictable cases is correct.

Most of the code we change is in the part of the code that simulates the core. One of the biggest changes is the replacement of the reorder buffer with a blocking pipeline in our in-order version. For this, our pipeline holds information about which instruction is at each stage. Any of the fields held by instructions to help track reorder buffer information is replaced with fields to help track stage information. In the out-of-order version, the reorder buffer is checked at each stage to see which instructions are available to advance. In our in-order version, we instead only check the instruction held by the pipeline at that stage.

Another difference from out-of-order execution is handling memory dependency issues. Due to the nature of in-order execution, we were able to remove most sections dedicated to handling this. RAW dependency is the only type we need to worry about for in-order execution. For registers, we block the operand stage until the dependent write is completed. For memory instructions, the out-of-order processor already blocks load instructions with dependent stores, and our in-order version does something similar by blocking the stage in the same way as registers.
5 Results

5.1 Disorder Between Cache Levels

As mentioned previously, absolute disorder is how far out-of-order memory instructions are issued from program order. Figure 1 shows the absolute disorder for the bzip and bwaves benchmarks at each cache level.

The vertical axis of Figure 1 shows the percentage of memory instructions that occurred with that absolute disorder. The more spread out the graph is across the horizontal axis, the more disordered the memory instructions. Conversely, a large spike around 0 absolute disorder means that there is very little disorder. As can clearly be seen by both examples, disorder decreases significantly as you move further from the processor through the cache hierarchy. In both examples, we see a large decrease in disorder from L1D to L2C, and a smaller decrease from L2C to LLC. In bwaves, the percentage of instructions with 0 absolute disorder jumps from 6.7% to 49.1% when going from L1D to L2C, and 49% to 52.7% from L2C to LLC. In bzip, the percentage of instructions increases from 35.4% to 84.3% when going from L1D to L2C, and 84.3% to 85.1% from L2C to LLC.

This decrease in disorder occurs because only some memory instructions make it from
one cache level to the next. Since all memory instructions are seen in the L1D, there are many instructions to potentially become disordered. Since only some of those instructions reach the L2C, the disorder decreases, because the instructions that make it through are more spread out. The same case occurs when going from L2C to LLC, but because the disorder has already dropped so significantly already, it doesn’t change as much.

5.2 Disorder at Different Reorder Buffer Sizes

The same absolute disorder metric was used to check how varying the reorder buffer size affects the amount of disorder. Figure 2 shows how varying the reorder buffer size affects the amount of disorder on the astar and cactusADM benchmarks.

![Figure 2](image-url)

**Figure 2** Absolute disorder for different reorder buffer sizes in astar and cactusADM.

In this case, the disorder changes very little in relation to altering the reorder buffer size. For astar, the percentage of instructions at 0 absolute disorder changes from 21.7% to 20.2% to 19.4% when increasing the reorder buffer size from 128 to 256 to 512 respectively. For cactusADM, the percentage of instructions at 0 absolute disorder changes from 8.7% to 7.7% to 7.1% when increasing the reorder buffer size from 128 to 256 to 512 respectively. In general, most traces only saw a decrease by about 1% when doubling the reorder buffer size.
size. This result is expected, because increasing the reorder buffer size gives more room for instructions to execute out-of-order. However, our results aren’t quite the same as what was found by Jaleel when studying the same relationship. While Jaleel also found that increasing the reorder buffer size increased the amount of disorder, doubling the reorder buffer size caused increases in disorder of upwards of 10% in their results (Jaleel, 2006). This discrepancy in results could be caused by two different possibilities. One of these is the difference in benchmarks. For their study, SWIM was used for disorder measurements, which we didn’t use at all. However, since we saw similar low disorder increases across all of our benchmarks, this is probably not the issue. The other possibility is based on the simulator used. As mentioned before, we used ChampSim for all of our data collection. Jaleel on the other hand used the Alpha 21264 simulator for his data collection (Jaleel, 2006). The difference in simulators here could be what is causing the large discrepancy.

5.3 Effect of Disorder on Prefetching

Each prefetcher was run with the simulator at the L2C level with in-order execution and each reorder buffer size in out-of-order execution. Figure 3 shows prefetcher performance statistics on each prefetcher for bwaves and calculix.

![Figure 3](image_url) Coverage and accuracy for bwaves and calculix for different run types.
From left to right are the graphs for BO, ISB, and VLDP. Each graph contains the coverage and accuracy for each different type of run. The first type of run is the in-order execution, and the other three types are the out-of-order executions with reorder buffer sizes 128, 256, and 512 respectively. The coverage of the prefetcher is the amount of useful prefetches divided by the total number of cache misses. A useful prefetch in this case, is one that causes a cache hit that would otherwise be a miss. The accuracy is the number of useful prefetches divided by the total number of prefetches.

The accuracy for each prefetcher on these benchmarks is close to 100%, so there isn’t much change between run types. However, there is a sizable difference in the change in coverage between in-order execution to out-of-order execution. Because all of the results are similar between reorder buffer sizes, we will mostly compare in-order execution to the out-of-order execution with a reorder buffer size of 128. For BO, the coverage only dropped from 92% to 90% for bwaves, and from 90% to 88% for calculix. For ISB, it dropped from 63% to 54% for bwaves, and stayed at 92% for calculix. On the other hand, coverage for VLDP dropped from 87% to 46% for bwaves, and 77% to 46% for calculix. This change is very clear, and matches our expectations. As we explained before, BO and ISB are expected to be mostly insensitive to disorder, while VLDP is expected to be sensitive to it. We see the small decrease in coverage for BO and ISB because it is expected for disorder to affect prefetching in some minor ways. The drop of 41% and 31% in coverage by VLDP however is caused by its prediction mechanisms being affected by the disorder. Figure 4 shows results over all benchmarks.

The boxes show the standard deviation around the mean, while the sticks show the minimum and maximum differences. The differences show the drop in each category while going from the one level to the next. As you can see from the graphs, VLDP has the highest average drop in coverage from in-order to a reorder buffer size of 128. On average over all benchmarks from in-order to a reorder buffer size of 128, BO coverage dropped 1%, ISB coverage dropped 1.6%, and VLDP coverage dropped 11%. VLDP coverage also dropped
Results

Figure 4 Change in coverage and accuracy between types for each prefetcher.

the most in each other category, dropping 1.3% from a reorder buffer size of 128 to 256, and 0.6% from a reorder buffer size of 256 to 512.

Accuracy remained mostly the same despite the increase in disorder. On average from in-order to a reorder buffer size of 128, BO dropped 1.7%, ISB dropped 0.3%, and VLDP dropped 3.0%. These changes are much more minor than the decreases in coverage. However, we can still see that VLDP is most affected by the change in disorder.

Notable Exceptions

While looking into the effect of disorder on prefetcher performance, we found that overall, prefetcher performance decreased with disorder, with some prefetchers being more affected than others. This, however, is not the case for all benchmarks. As you might have noticed from Figure 4, there were some outliers where performance actually increased unexpectedly. Figure 5 shows one such example.

As you can see from Figure 5, coverage actually increased in all prefetchers from in-order to a reorder buffer size of 128. Coverage for BO increased by 8.2%, ISB increased by 1.5%, and VLDP increased by 4.5%. While the results are inconsistent with other benchmarks, this benchmark is the only one tested which resulted in all three prefetcers getting performance...
increases with an increase in disorder. This suggests that there is something about bzip in particular which causes this, and that there are other benchmarks that could potentially show similarly unexpected results.

6 Conclusion

We attempted to analyze the extent of disorder in out-of-order systems, as well as its effects on prefetcher performance. We successfully created an in-order version of ChampSim to help us with this analysis. We looked into disorder at each cache level and found that as expected, disorder decreases when moving to cache levels further from the processor. We also looked into how much disorder changes when varying the reorder buffer size. We found that disorder also increases when increasing the reorder buffer size, but not to the extent that previous research had shown. We looked into the effect of disorder on three different prefetchers: BO, ISB and VLDP. We found that VLDP was more affected by disorder than BO or ISB, which agreed with our understanding of how each prefetcher worked. Overall, we confirmed that disorder is an important metric to take into account when designing prefetchers. Because almost all modern processors use out-of-order execution, it is important to make prefetchers robust against disorder.
7 Future Work

Because we found a discrepancy in the amount of disorder seen when varying the reorder buffer size, more work could be done to look into exactly what is causing this. As mentioned before, it is likely that the difference in simulators used in collecting this data is the cause. If this is the case, then it would be important for future work to understand why this is happening, and which simulator is better to use.

We also found that not all benchmarks behave as expected. Some benchmarks, such as bzip, show the opposite effect as expected across all prefetchers. More study could be done into why this is the case, which could open up another area of study.

Another area for future work would be to test more prefetchers. While these three prefetchers validate our current understanding, testing other prefetchers could lead to more interesting results as well as validate our current ones.
References


Jain, Akanksha and Calvin Lin (2013). “Linearizing Irregular Memory Accesses for Improved Correlated Prefetching”. In: *MICRO 2013*.


Shevgoor, Majunath et al. (2015). “Efficiently prefetching complex address patterns”. In: *MICRO 2015*.