Characterizing PBBS: Speculation in the memory and pipeline

Henry Chen, Runyi Li, Nicholas Pedro, Taha Usman

I. Introduction

The Problem Based Benchmark Suite (PBBS) is a collection of benchmark problems defined in terms of I/O specification with varying algorithms and implementations [1]. This paper aims to provide a workload characterization of the PBBS benchmark suite for single-core systems using speculation in the memory and pipeline. Over a variety of benchmarks, we contrast prefetcher configurations, and we explore the performance of branch predictors.

In this study, we draw on previous works which have explored Sequential Program Prefetching [2], and characterized the PARSEC benchmark suite [5]. These prior works highlighted the importance of analyzing performance across a variety of benchmarks. Thus, we selected from the PBBS library a subset of 4 benchmarks to focus our exploration on, as these benchmarks offered varying results which reflect their diverse code implementations. These benchmarks are outlined in Table 1. We chose serial BFS to emphasize the importance of underlying code data structures, which lead to contrasting results with other benchmarks (explored later, so read on). Next, we analyse Sample Comparison Sort (which we refer to as "comparison sort") and Word Count Serial as both examples are representative of tasks that have inherit temporal locality, which becomes important for prefetch analysis. Finally, we chose Word Count Histogram to emphasize how the code implementation of the task is just as important as the problem itself. We find that system performance varies between these two implementations, despite the fact that they solve the same problem. Overall, we intend for this selection of workloads to present interesting results, and better reflect practical workloads that involve solving a variety of different problems.

		Instruction Count	Operations
Serial BFS	No Prefetch	131924171	236377322
	Stride	128325290	230220876
	Tagged	117528752	211751742
Sample Comparison Sort	No Prefetch	177020935	293980685
	Stride	179275901	297707155
	Tagged	177018030	293974724
Word Count Serial	No Prefetch	87411974	194016860
	Stride	87412078	194017120
	Tagged	82924753	184001224
Word Count Histogram	No Prefetch	122537682	230064321
	Stride	124538635	233827566
	Tagged	124535718	233822327

Table 1. Instruction Count and Number of Operations Including Micro Ops

II. Exploration 1

In this section, we contrast the performance of using no prefetcher against using two different prefetchers (stride and tagged) for both the data and instruction caches at once. Across varying benchmarks, we expect adding a prefetcher to either data or instruction L1 cache should decrease the simulation time and decrease the miss rate of their respective L1 cache. This is supported by previous works which found that cache memory prefetching in large computer systems improved performance at low cost [2]. The degree of these improvements will depend on the benchmark and the type of prefetcher used.

A. Experimental setup

Simulations are conducted on each benchmark using three different virtual systems constructed within the gem5 simulator. The first system uses no prefetcher while the other two use prefetchers on both their data and instruction caches. Of the two with prefetchers, one system uses the `StridePrefetcher`, which looks for consistent strides in access patterns and prefetches the next predicted address. The other system uses the `TaggedPrefetcher` with a degree of 2, which fetches the next 2 cache lines after a demand access.

To reduce the influence of any other variables and ensure fairness in our evaluation, the other characteristics of all systems are equivalent. Each system uses 8GB of `DDR3_1600_8x8` main memory and each processor is clocked at 75Mhz. Both L1 instruction and data caches are 2-way set-associative, with a 4KB size, where tag, data, and response latencies are 1 cycle. They have 8 Miss Status Holding Registers and each can track 20 memory accesses (See Appendix A).

Inputs to the benchmarks were generated when necessary using the input generation scripts included with PBBSbench (see Appendix A). Finally, all benchmarks were ran using gem5, by supplying a direct path to each compiled binary file and the appropriate arguments (See Appendix A). Inputs were kept constant between simulations using the three systems to ensure fairness.

B. Results and discussion

To evaluate overall prefetching effectiveness, we focus on simSeconds as this is what we would like to improve in real-world applications. Across our chosen spread of benchmarks, we obtained varying results, which reflect the specific nature of

the benchmarks. However, as seen in Appendix A, processor CPI is directly correlated with simulation time, which makes sense since processor frequency is fixed, so a decrease in simulation time is only possible if CPI decreases. Thus, we can infer simulation time based on CPI, and our results are easier to explain with CPI, so they will be the focus of our following analysis.

Beginning with Serial BFS, we see that both stride and tagged perform far worse than using no-prefetching (see Appendix A). In comparison to other benchmarks, this result is very extreme. However, this result makes sense as if we examine data cache pre-fetch accuracy (Appendix B), we see that in comparison to other benchmarks, data cache and instruction cache prefetch accuracy (Appendix C) is significantly lower than on other benchmarks. Other metrics such as prefetch coverage and unused fetchs metrics are also particularly poor. However, we can observe that far more prefetchs are actually issued in the data cache for this benchmark in comparison to other benchmarks. So overall, we seem to be issuing far more useless prefetchs than in other benchmarks, which ends polluting our cache and reducing accuracy. By polluting the cache, we reduce the overall cache hit rate, which in turn negatively impacts performance. We believe the reason for this is due to the nature of the benchmark itself. Although tree searching might seem like a big loop with structured reads, because of the tree structure itself, subsequent node reads might actually be in memory regions/indexes far apart, leading to many useless speculative prefechs. This example highlights how simply targeting loop structures and instruction patterns is not sufficient, we also need to take into account the fundamental structure of the data itself, which may be challenging if not impossible to achieve in hardware. An interesting future consideration would be whether we can identify and leverage common data structures using software instead (possibly by restructuring the underlying data structure in a way that creates better spatial locality), and if this could lead to serious benefits.

In contrast, in comparison sort and serial word count, we see that stride prefetching yields noticeable performance improvements, whereas tagged prefetching is actually detrimental to performance. In both cases, tagged issues far more prefetchs than stride (the difference is larger for serial word count, which explains why tagged is even worse in word count than comparison sort); yet, the accuracy is significantly worse for tagged in comparison to stride. Based on the data cache coverage, we can see that both prefetchs are eliminating a significant amount of cache misses. However, simply eliminating cache misses is not enough, we must also consider the additional overhead to achieve this increased coverage. Though in absolute terms, tagged can issue more useful prefetchs than stride (Appendix B), the relatively poor accuracy means that the overhead of polluting the cache with unused fetchs far out weights these absolute gains (we simply issue far more useless prefetchs than useful prefetchs). Consequently, we see an increase in CPI in tagged compared to both stride and no prefetching. In comparison, stride prefetching in both of these benchmarks is the highest accuracy we have observed across all benchmarks. We have passed the critical point where we make enough useful prefetchs relative to useless prefetchs that the benefits of increased coverage outweigh the increased overhead, leading to decreased CPI and faster simulation time. We think this can be explained by the relatively ordered nature of these algorithms, where we can actually exploit spatial locality. However, we suspect that this spatial locality is fairly limited in nature, and thus our tagged prefetcher can easily prefetch too aggressively (taking too large of a region of data), leading to negative results. However, based on our instruction cache results, it appears that tagged actually has fairly good accuracy on the instruction cache across all of our benchmarks. So a future extension could be looking at pairing a stride prefetcher for the data cache with a tagged prefetcher for the instruction cache, which could yield the most optimal results.

Finally, when examining word count histogram, we found that both stride and tagged prefetching yielded noticeable improvements over no prefetching, with stride prefetching again slightly outperforming tagged prefetching. In comparison to other benchmarks, both stride and tagged prefetching have relatively high accuracy, and achieve relative high coverage (tagged achieved the highest coverage compared to all benchmarks here). Thus, we can conclude that in this benchmark, both stride and especially tagged are successfully eliminating a significant amount of cache misses, which reduces CPI. Moreover, the relatively high accuracy means that the overhead trade off is not too significant compared to the benefits of increased data cache hit rate, so overall we can achieve performance improvements. Considering that both the histogram and serial word count solve the same problem, the observed variation in performance uplift must be due to the specific code implementation (though we could not determine exactly why).

In summary, our chosen experimental setup yielded a wide variety of different scenarios, which highlights the importance of the underlying workload on expected results. Contrary to our initial hypothesis, using prefetching does not always yield speedups, it can in fact be detrimental to performance in many instances. We also highlighted future extensions of our work, including exploring configuring common data structures to better leverage prefetching and heterogeneous prefetcher configurations.

III. Exploration 2

Branch prediction is a critical component of modern processors, influencing execution efficiency and overall performance. This exploration aims to characterize the accuracy of different branch predictor configurations within the gem5 simulation framework. Specifically, we compare the Local Branch Predictor against the Tournament Branch Predictor and also different sizes of the latter to evaluate their effectiveness across various benchmarks. By isolating branch predictor as the primary variable, we aim to understand how different predictor sizes and strategies impact accuracy, ultimately informing architectural design decisions for speculation in memory and pipeline execution.

			Issued	Unused	Useful	Accuracy	Coverage
L1d	Serial BFS	Stride	198880	66322	3108	0.015628	0.016483
		Tagged	336938	278301	5558	0.016496	0.027016
	Sample Comparison Sort	Stride	16556	1066	3861	0.233209	0.287812
		Tagged	35339	13042	2649	0.07496	0.165604
	Word Count Serial	Stride	12998	1915	2732	0.210186	0.06784
		Tagged	120625	54600	7932	0.065758	0.152395
	Word Count Histogram	Stride	13281	1076	2160	0.162638	0.14975
		Tagged	40069	13420	5344	0.13337	0.339949
L1i	Serial BFS	Stride	0	0	0	0	0
		Tagged	72	25	13	0.180556	0.419355
	Sample Comparison Sort	Stride	0	0	0	0	0
		Tagged	6618	2349	1120	0.169235	0.427971
	Word Count Serial	Stride	0	0	0	0	0
		Tagged	135655	44919	13247	0.097652	0.278826
	Word Count Histogram	Stride	0	0	0	0	0
		Tagged	930	302	141	0.151613	0.398305

Table 2. Prefetch Stats for Stride and Tagged Data & Instruction L1 Cache

A. *Experimental setup*

To evaluate the impact of different branch predictor configurations on performance, we employ a simulation-based methodology across four PBBS benchmarks: serialBFS, sampleSort, wordCounts histogram and serial. In this section, we describe the system configuration, branch predictor variations, and the simulation framework used for data collection.

Simulations are conducted separately for each benchmark using different branch predictor configurations within a fixed system setup. The baseline system uses the gem5 simulator with the X86TimingSimpleCPU. Specifically, we evaluate the Local Branch Predictor with the specified default Tournament Branch Predictor and Tournament Branch Predictor small and large against each other to assess their impact on accuracy.

The system setup remains constant across all simulations to ensure a fair comparison. Each simulation is executed using the same input set and runtime configuration, allowing for an isolated evaluation of the branch predictor's impact. The system is configured with the following parameters: - **Processor**: X86TimingSimpleCPU (Fixed clock speed and execution model). - **Memory**: 8GB DDR3 1600 8x8 main memory. - **Branch Predictors**: - **LocalBP**: Standard local branch predictor used as a baseline. - **TournamentBP (default)**: Configured with: - Local Predictor: 2048 entries, 2-bit counters, 2048-entry history table. - Global Predictor: 8192 entries, 2-bit counters. - Choice Predictor: 8192 entries, 2-bit counters. - **TournamentBP (small)**: Configured with: - Local Predictor: 512 entries, 2-bit counters, 512-entry history table. - Global Predictor: 1024 entries, 2-bit counters. - Choice Predictor: 1024 entries, 2-bit counters. - **TournamentBP (large)**: Configured with: - Local Predictor: 4096 entries, 2-bit counters, 4096-entry history table. - Global Predictor: 16384 entries, 2-bit counters. - Choice Predictor: 16384 entries, 2-bit counters.

Each benchmark is executed under these configurations to analyze the variation in accuracy. The simulations are performed using the gem5 simulator by specifying the path to the compiled benchmark binary and the corresponding arguments. These arguments remain fixed across all runs to ensure consistency. For each benchmark, results are extracted from the 'stats.txt' file, with the second dump used as the region of interest. This selection ensures that performance measurements capture the steady-state execution of the benchmark, excluding initialization and de-initialization phases.

This methodology provides a controlled environment to isolate the impact of branch predictor configurations, allowing for a detailed analysis of prediction accuracy and its effect on benchmark execution.

B. *Results and discussion*

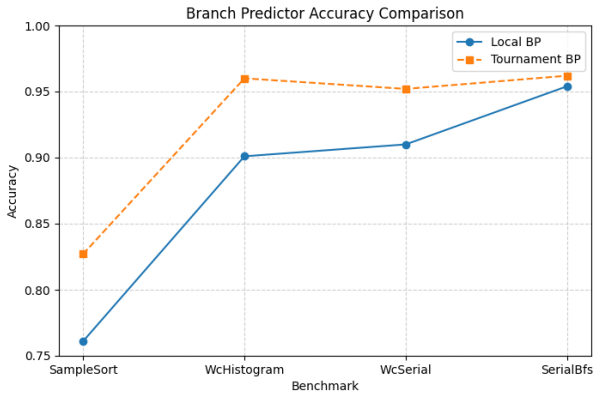

Fig. 1. Branch Predictor Accuracy Comparison between Local BP and Tournament BP

In this section, we analyze how branch prediction accuracy varies across different benchmarks and predictor configurations. Specifically, we compare the performance of the Local and Tournament branch predictors and investigate how predictor size impacts accuracy.

Figure 2 illustrates that the Tournament Branch Predictor consistently achieves higher accuracy compared to the Local

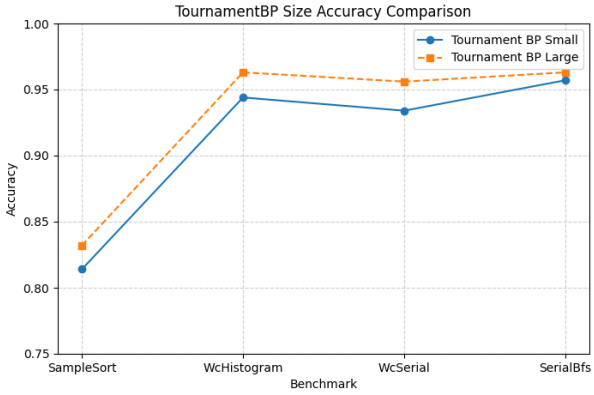

Fig. 2. Impact of Tournament BP Size on Prediction Accuracy

Branch Predictor across all benchmarks. The magnitude of improvement varies depending on the workload, with ‘SampleSort‘ exhibiting the largest accuracy gap (0.827 vs. 0.761). This suggests that ‘SampleSort‘ has complex branch behavior that the Local BP struggles to capture, whereas the Tournament BP, leveraging both local and global history, performs better.

‘WcHistogram‘ and ‘WcSerial‘ show relatively smaller differences between the two predictors, indicating that their control flow is more predictable. Notably, ‘SerialBfs‘ achieves the highest accuracy for both predictors, suggesting that its branch patterns are highly structured and easier to predict.

Examining the benchmark source code provides further insights into these accuracy variations. ‘SampleSort‘, being a sorting algorithm, likely involves branches that depend on unpredictable data-dependent comparisons, making branch prediction more challenging. On the other hand, workloads such as ‘SerialBfs‘, which likely exhibit structured looping behavior with fewer unpredictable branches, see high accuracy across both predictors. These observations highlight the importance of workload characteristics in determining branch prediction effectiveness.

Figure 3 examines the effect of increasing the Tournament BP size on prediction accuracy. The results indicate that increasing the predictor size improves accuracy across all benchmarks, but the extent of improvement varies. ‘SampleSort‘ benefits the most from a larger predictor, likely due to its irregular branch behavior that benefits from additional history tracking.

Interestingly, ‘WcSerial‘ shows a slight decrease in accuracy with a larger predictor, which could be attributed to aliasing effects or overfitting to less useful branch history. Meanwhile, ‘SerialBfs‘ sees little improvement, suggesting that its branch patterns are already well-predicted with a smaller predictor.

These findings indicate that while larger predictors generally improve accuracy, the benefits diminish for workloads with simpler control flow. Thus, choosing the appropriate predictor size requires considering both the workload characteristics and the diminishing returns of increasing hardware complexity.

IV. Conclusion

Overall, characterizing the Problem Based Benchmark Suite for single-core configured systems using the gem5 simulator provided valuable insight into the performance implications of speculation in the memory and pipeline. It underlined the ”no free-lunch” principle in prefetching, where speculation in memory was some times beneficial but at other times detrimental to performance. Overall, this stresses the importance of considering specific source code implementation when estimating the potential value of including memory prefetching. Finally, the workload characterization showed that while larger branch predictors generally improve accuracy, the benefits are less apparent for workloads with simpler control flow, which must be weighed against increased hardware complexity.

References

[1] Anderson, Daniel, et al. ”The problem-based benchmark suite (PBBS), v2.” Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. 2022.
[2] Smith, Alan Jay. ”Sequential program prefetching in memory hierarchies.” Computer 11.12 (1978): 7-21.
[3] Chen, Tien-Fu, and Jean-Loup Baer. ”Effective hardware-based data prefetching for high-performance processors.” IEEE transactions on computers 44.5 (1995): 609-623.
[4] Yeh, Tse-Yu, and Yale N. Patt. ”Two-level adaptive training branch prediction.” Proceedings of the 24th annual international symposium on Microarchitecture. 1991.
[5] Bienia, Christian, et al. ”The PARSEC benchmark suite: Characterization and architectural implications.” Proceedings of the 17th international conference on Parallel architectures and compilation techniques. 2008.

APPENDIX A: PREFETCHING SYSTEM STATISTICS

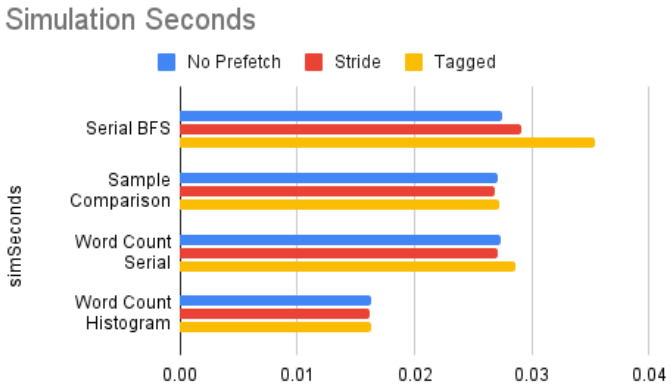

Fig. 3. Result from my implementation

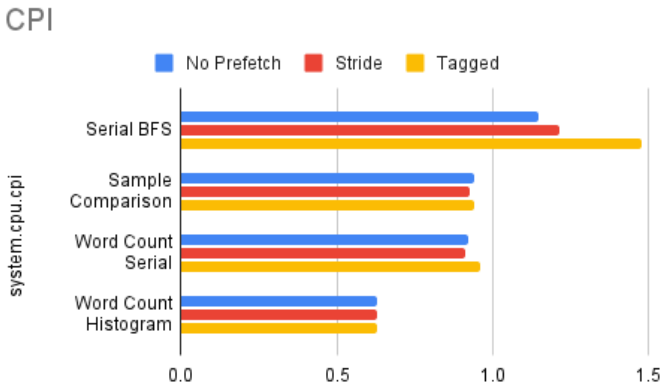

Fig. 4. Result from my implementation

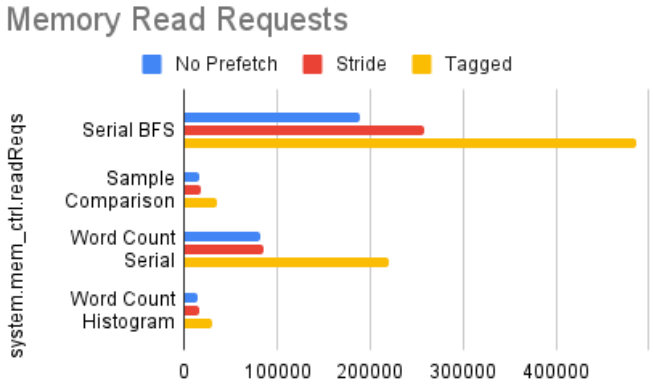

Fig. 5. Result from my implementation

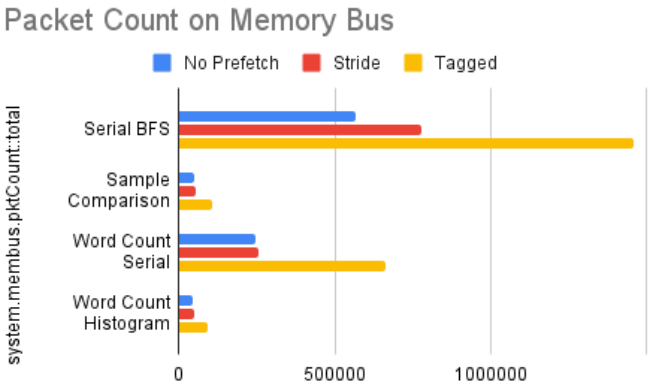

Fig. 6. Result from my implementation

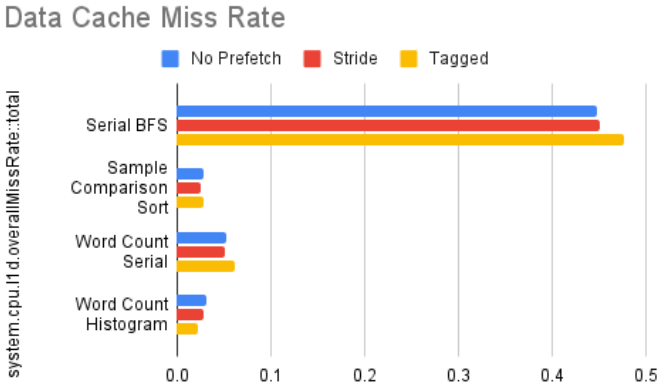

Fig. 7. Result from my implementation

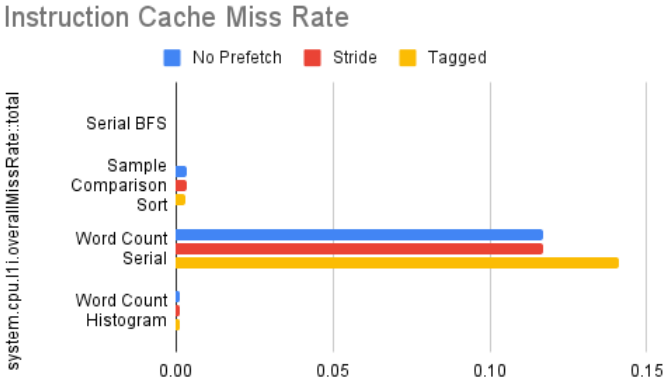

Fig. 8. Result from my implementation

APPENDIX B: PREFETCHING DATA CACHE STATISTICS

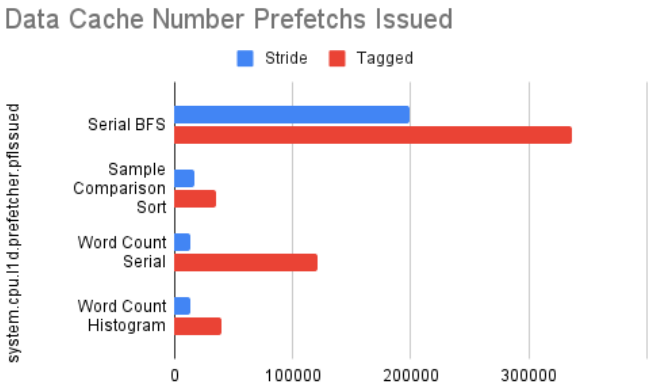

Fig. 9. Result from my implementation

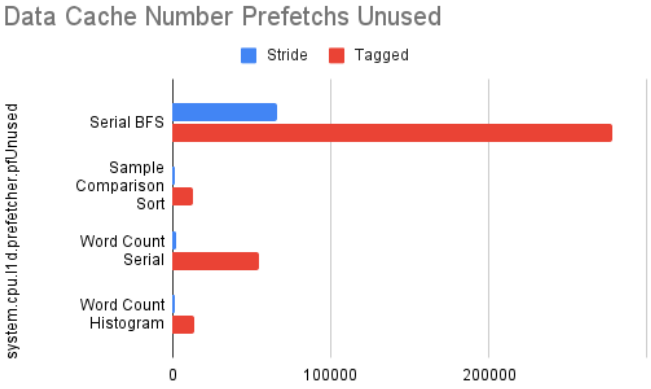

Fig. 10. Result from my implementation

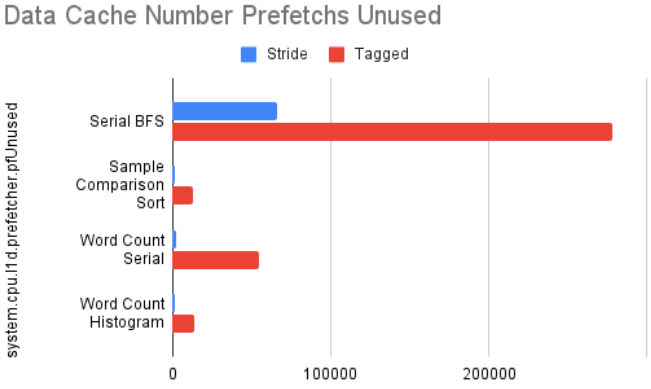

Fig. 11. Result from my implementation

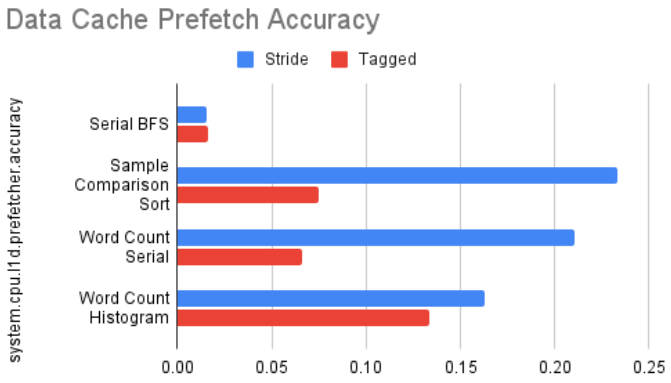

Fig. 12. Result from my implementation

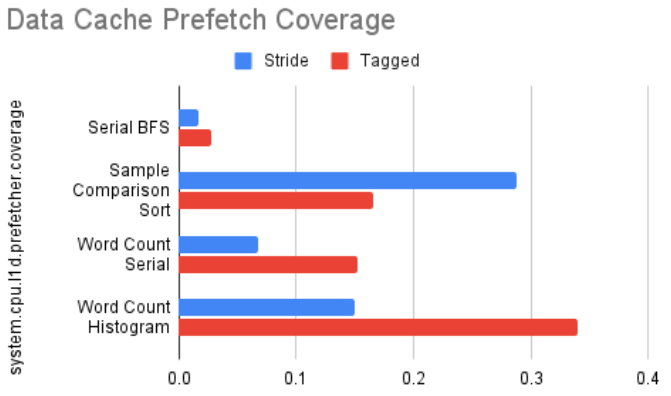

Fig. 13. Result from my implementation

Appendix C: Prefetching Instruction Cache Statistics

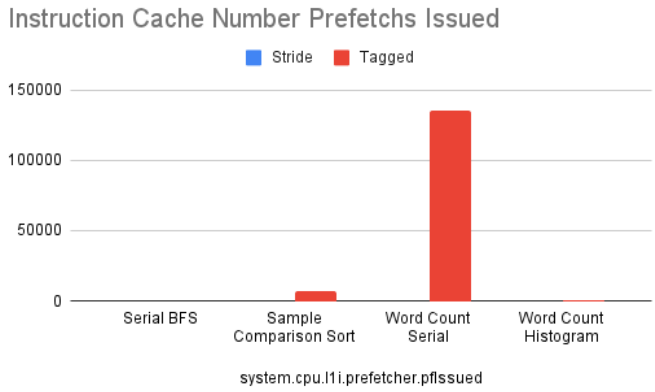

Fig. 14. Result from my implementation

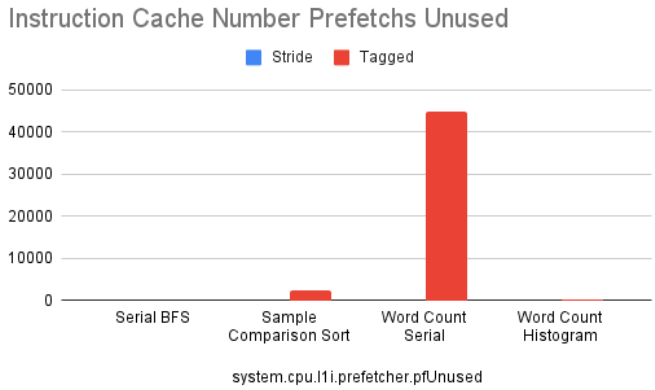

Fig. 15. Result from my implementation

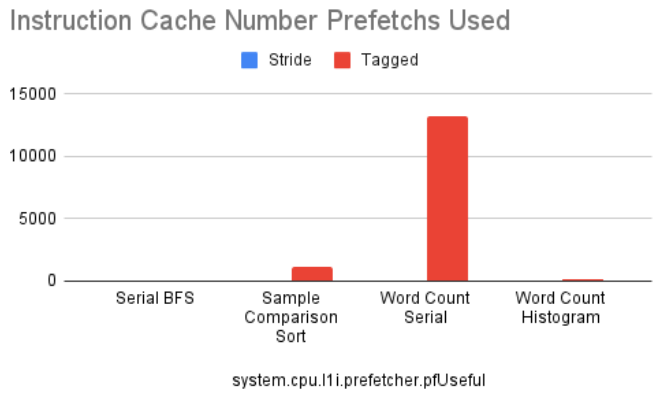

Fig. 16. Result from my implementation

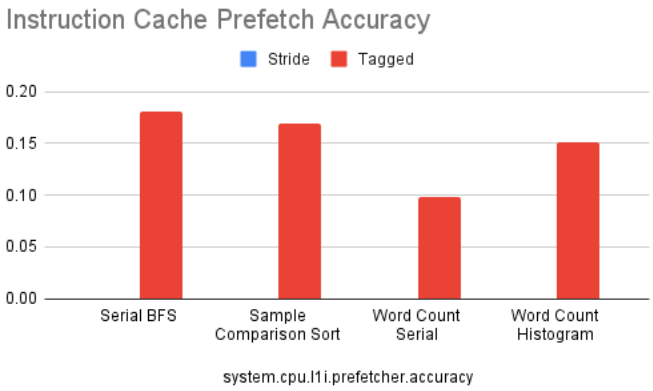

Fig. 17. Result from my implementation

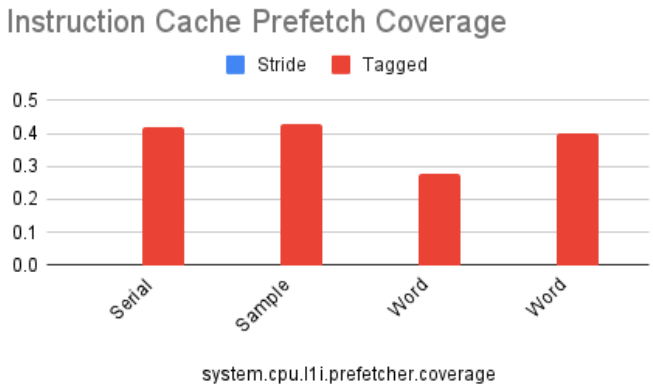

Fig. 18. Result from my implementation