A specialized bouquet of instruction prefetchers

Ayush Agarwal*, Sankalan Baidya† and Soham Joshi‡
Department of Computer Science, IIT Bombay
Mumbai, Maharashtra, India
Email: * ayushagarwal@cse.iitb.ac.in, † somekoln@cse.iitb.ac.in , ‡ sohamjoshi@cse.iitb.ac.in

***Abstract*—This work is built upon IPCP 1.0 by Biswa, which uses a bouquet of prefetchers. We are proposing specialized prefetchers for different categories like Graphs, Sat-solvers, SPEC and Server workloads. This is based upon the intuition that different types of traces will follow different types of access patterns. The ideas used in this work have been built upon from various `DPC3` submissions [1] [2] [3].**

***Index Terms*—Prefetechers, Computer Architecture, DPC3**

I. Introduction

Hardware prefetching is a technique used by computer processors to fetch data from memory (into the cache) before it's actually needed, reducing the time spent waiting for data to arrive. This is achieved by analyzing memory access patterns and predicting which data is likely to be needed next, and fetching it in advance.

In this report, we examine how we have explored ideas from the DPC3 submissions and beyond in order to improve upon the performance of the `IPCP` prefetcher. Moreover, we will analyse the increase or decrease in performance via various prefetching strategies in the contexts of graph analytics, SAT Solvers, SPEC, and Server Workload traces. All code, traces used to generate results can be found here. We have used the Champsim simulator for all data generated in this report.

The authors of [1] have built an IP (instruction pointer) based prefetcher than can predict majority of the access patterns. However, we find that the themes of accuracy and coverage described in [2] can be extended to [1] quite easily. As it turns out, [3] can be integrated with [1] as well. These changes often work well with some types of traces and don't work with others. We explore these ideas in the sections below.

II. Graph Analytics

Prefetching in the context of graph analytics has posed a significant challenge in recent times. This is because hardware prefetchers are designed to work well with applications that exhibit predictable memory access patterns. However, graph analytics algorithms, which involve traversing large graphs with irregular memory access patterns, pose a challenge for hardware prefetching. The traversal order of a graph is often dependent on the input data, which makes it difficult for a hardware prefetcher to accurately predict which data to prefetch. Additionally, the size of the graphs used in graph analytics can exceed the capacity of the prefetcher's buffer, making it difficult to store enough prefetched data to improve performance.

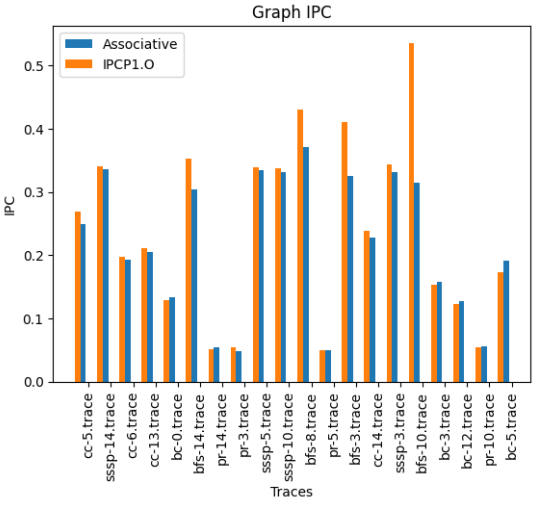

Fig. 1. This figure shows the IPC (instructions per cycle) for IPCP prefetcher, associative table approach, clearly, IPCP is better

A. Our solution

We observed that the `IPCP` prefetcher does a good job of predicting strides but, it essentially uses a 1-way associative cache whenever IP-stride tables are required. That is, it uses only the generic parameter `IP` for indexing. However, the themes described in [2] help us here as we can modify these look up tables in order to generalise them to *multi-parameter* look up tables. Analogous to the paper, we use `IP` and `IP + addr` as our indices, and a k-way associative cache with `IP` used to hash the sets, with `IP + addr` giving us the tag. However, this approach does not work as well as we thought it would. The results are given in fig. 1.

B. A simpler approach

The previous approach requires making our tables k-associative. However, this causes a significant overhead, along with slower look-ups and insertions. Instead of this, we tried simplifying these tables to be 1-way associative, but we use `IP` to index and `IP + addr` to tag, introducing some level of "specificity". This approach gives us significant performance improvements on graph traces. (ref fig.2, 3, 4, 5)

C. More specificity in indexing

We can make the above approach even more specific. Instead of indexing using the `IP`, we now use `IP + addr`

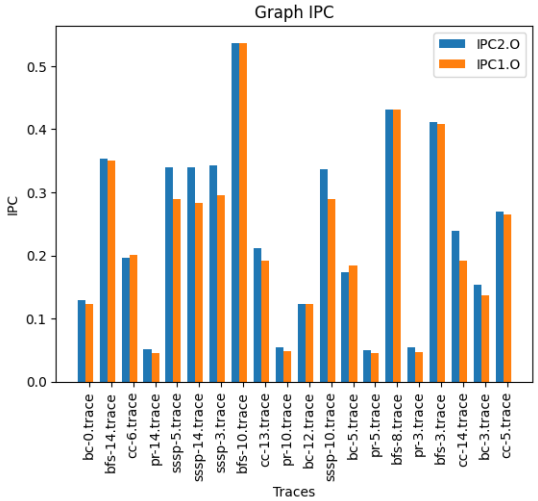

Fig. 2. This figure shows the IPC (instructions per cycle) for the IPCP prefetcher, when our prefetcher incorporates indexing with IP and tagging via IP and address of the line

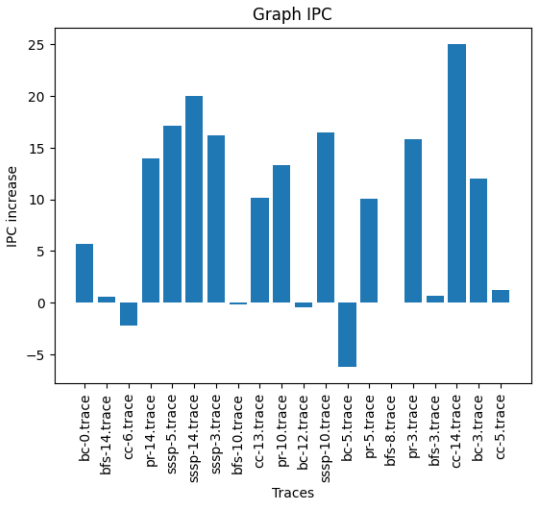

Fig. 3. This figure shows the IPC (instructions per cycle) relative to the IPCP prefetcher, when our prefetcher incorporates indexing with IP and tagging via IP and address of the line

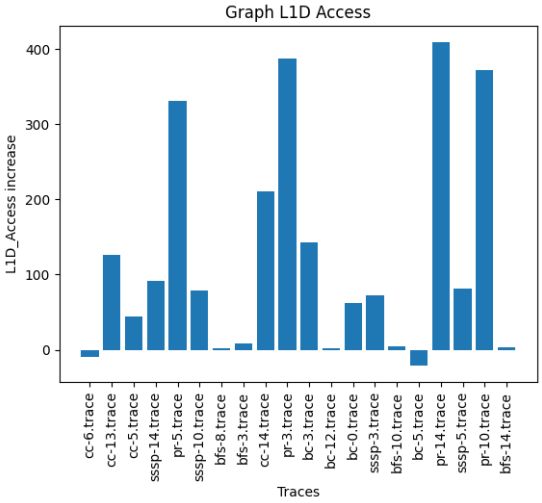

Fig. 4. This figure shows the Accuracy (hits per accesses) in L1D cache level, relative to the IPCP prefetcher, when our prefetcher incorporates indexing with IP and tagging via IP, address of the line

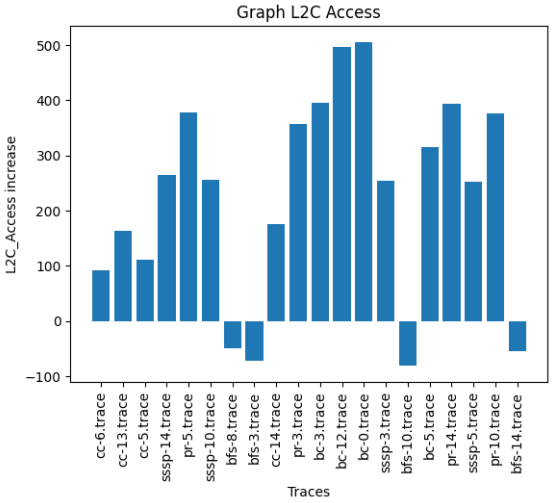

Fig. 5. This figure shows the Accuracy (hits per accesses) in L2 cache level, relative to the IPCP prefetcher, when our prefetcher incorporates indexing with IP and tagging via IP, address of the line

for both tagging and indexing. This ofcourse leads to lesser coverage, more accuracy. But, how does this affect our performance metrics? As it turns out, the IPC drops by a significant amount as a consequence of less coverage (even if the IP is same, most of the time, addresses are different). Hence, we have omitted the results for this approach.

III. SAT Solvers

Prefetching in this context is again a challenge. This is because hardware prefetching is not effective for SAT solvers because SAT solvers typically access memory in a highly irregular and unpredictable manner. SAT solvers use backtracking search, which requires them to continually traverse different paths in a large search space, making it challenging for a hardware prefetcher to predict which memory locations to prefetch. Additionally, the size of the search space can be quite large, which means that the hardware prefetcher's buffer may not be able to hold enough data to make a significant impact on performance.

A. *Our solution*

Keeping the previous optimisations, we look to build the prefetcher further. First, let's have a look at the results with these optimisations alone. (ref fig. 6 7 8).

In order to improve the performance, we turn to [3]. Since `IPCP` is based upon integrating various stride prefetchers in order predict the next stride correctly, we tried integrating the next line prefetcher into `IPCP`. We have done this at the `LLC` (last-level cache), as that level did not have a prefetcher in the first place. As a result, we observed the following.

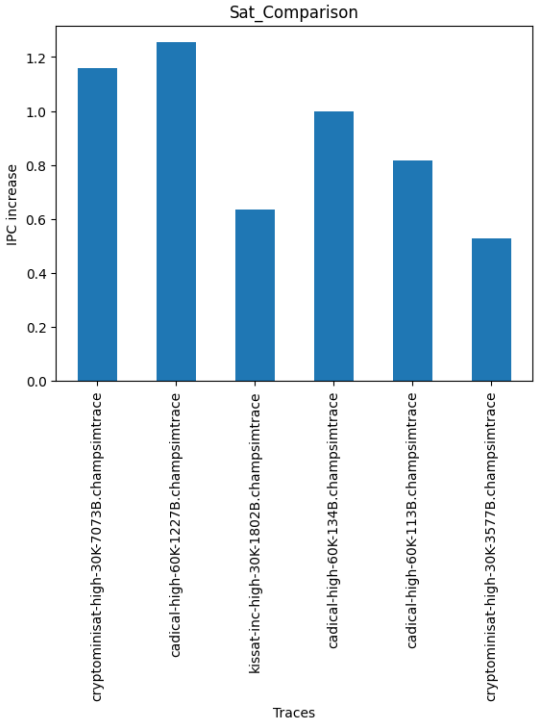

Fig. 6. This figure shows the IPC (instructions per cycle) relative to the IPCP prefetcher, when our prefetcher incorporates next line prefetcher in LLC

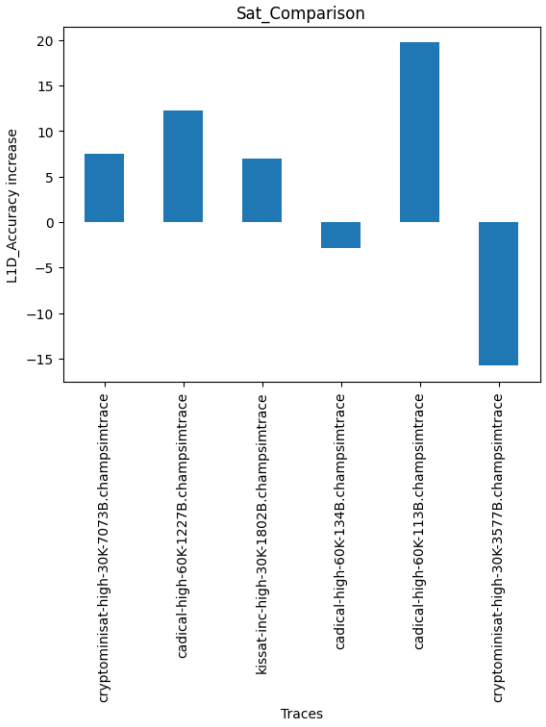

Fig. 7. This figure shows the Accuracy (hits per accesses) in L1D cache level, relative to the IPCP prefetcher, when our prefetcher incorporates next line prefetcher in LLC

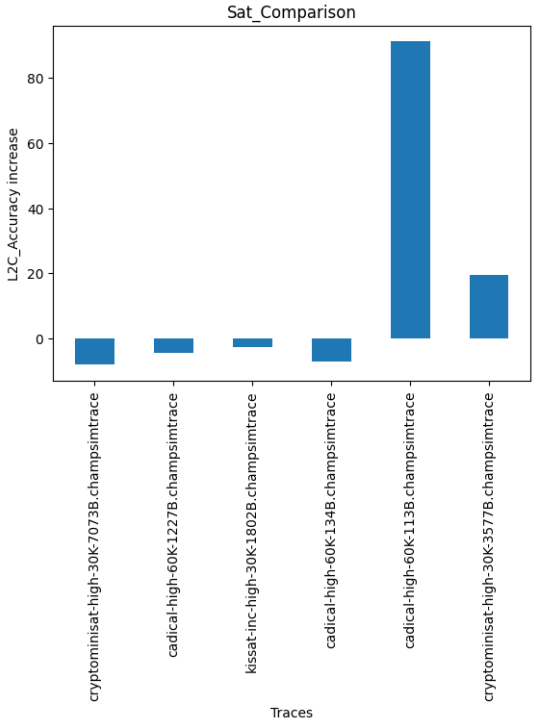

Fig. 8. This figure shows the Accuracy (hits per accesses) in L2 cache level, relative to the IPCP prefetcher, when our prefetcher incorporates next line prefetcher in LLC

Since `LLC` has a high size, we can explore putting other prefetching policies in LLC and comparing the results as a scope for further improvement.

IV. SPEC TRACES

Why is checking performance on SPEC traces important? This is because the benchmarks developed by SPEC are widely used by industry and academia to measure and compare the performance of different computer systems and components, including processors, memory, storage, and compilers. SPEC benchmarks cover a range of applications and workloads, including scientific and engineering simulations, multimedia processing, and database management.

Let's check the results on SPEC traces with the optimisations done so far.

A. Our approach

We need to optimise this further. In order to do so, we tried out the following approach. In order to reduce to losses in SPEC, we observe that replacing `IP + addr` with `IP + cl_addr` helps. Our observations are given below.

V. SERVER WORKLOADS

Hardware prefetching on server workloads poses a challenge due to reasons quite similar to what we have seen earlier. These workloads often involve a mix of different applications with varying memory access patterns. A hardware prefetcher is designed to predict memory accesses based on a program's

access patterns, but in a server environment, applications are frequently switching, making it difficult for the prefetcher to learn the access patterns of each application. Furthermore, server workloads often involve large datasets that can exceed the prefetcher's buffer size, limiting its ability to prefetch data effectively.

Via the current modifications, we observe the following.

A. Our approach

We need to optimise this further. In order to do so, we tried out the following approach.

VI. Results

Give a summary of the results here.

Conclusion and remarks

This report goes through some optimisations which show a sizeable improvement over the prefetchers presented in DPC3, albeit in application specific traces. The final improvements in IPC are approximately 10% in graph, 10% in SAT, 10% in SPEC and 10% in Server Workloads. (update actual values later)

Acknowledgements

We would like to thank the instructor of CS230, Biswa, for giving us the opportunity to work on this project which allowed us to explore a taste of real-life research. We would also like to thank all the CS230 teaching assistants for solving queries throughout the labs, which in turn allowed us to have sufficient background to work on this project.

References

[1] S. Pakalapati, B. Panda, "Bouquet of Instruction Pointers: Instruction Pointer Classifier-based Hardware Prefetching".

[2] M. Bakhshalipour, M. Shakerinava, P. L. Kamran, H. S. Azad, "Accurately and Maximally Prefetching Spatial Data Access Patterns with Bingo,".

[3] M. Shakerinava, M. Bakhshalipour, P. L. Kamran, H. S. Azad,"Multi-Lookahead Offset Prefetching".