Assignment-4: Data Prefetcher for LLC

Sai Gautham Ravipati EE19B053[1], **Shashank Nag EE19B118**[1]
[1]Department of Electrical Engineering, Indian Institute of Technology, Madras

1 Introduction

Cache prefetching refers to fetching data from the memory onto the cache, even before the said data block is requested by the processor. In doing so, we speculate that a particular data block might be requested by the processor in the future. By prefetching such a block into the cache, we are able to eliminate the ensuing cache miss latency when the processor requests the block, as a result of which, there is a performance improvement. However, aggressive prefetching might lead to eviction of useful data from the cache, and hence a balanced approach is to be taken for optimal performance.

Hence, there have been various prior works on data prefetching techniques, that seek to prefetch the most meaningful cache block onto the cache. In this assignment, we explore a few custom data prefetching techniques, that are primarily based on distance based and PC based prefetching. We compare the benchmarks obtained for our policies against the baseline prefetching policies, which include next-line prefetching, ip-stride prefetching and no prefetching. We also discuss some other possible explorations that could be carried out in future. All the simulations were carried out using the ChampSim simulation environment.

2 Existing Policies

Some of the most commonly used prefetchers include the next line prefetcher, and the stride based prefetcher. In case of the next line prefetcher, we naively prefetch the next cache line in addition to the currently requested cache line. On the other hand, in case of stride based prefetching, the cache line after a specific stride is prefetched. IP - stride, refers to a special case of this, where the stride is chosen by observing the stride corresponding to the previous accesses from the same Program Counter/ Instruction Pointer. We profile the performance of these policies on the provided traces, and the performance is as indicated in the graphs of Figures-2,3. For comparison, we also consider a system with no prefetching employed.

3 Custom Policies

3.1 Distance based Dynamic Stride Prefetching

We propose a Distance based Dynamic Stride Prefetching policy, where the stride is adaptively chosen based on the distance between subsequent cache block accesses. This is primarily inspired by a similar implementation for TLB accesses, in the work by Kandiraju et al.[1]

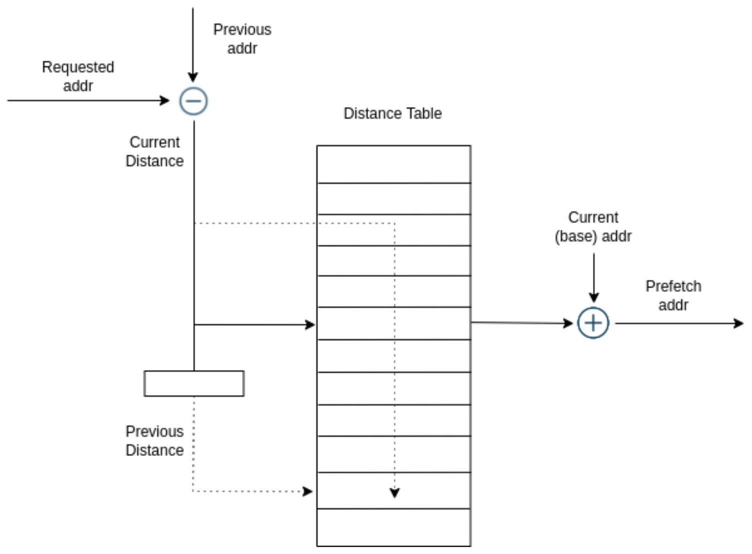

Figure 1: Distance based Dynamic Stride Prefetching Policy

As indicated in the Figure-1, in this technique, we maintain a table with entries that indicate the "distance" between two consecutive data accesses, i.e., the difference in the addresses of the data blocks that are requested by the processor consecutively. This table is itself indexed using the current distance. When a cache block is requested, we prefetch the data block that is at a distance as indicated by the entry in the table, indexed by the current distance. Once the prefetch is done, we update the current access distance to the table, against the index corresponding to the distance in the previous access. Effectively, we predict the distance of the block that is likely to be accessed next, given the distance of the currently accessed block from the previous block, and prefetch the same.

To compare the performance across different policies, we prefetch only one block per access request. The results obtained upon bench-marking are indicated in Figures-2,3. The distance based dynamic stride prefetching policy, has several advantages as it can support a wide range of access patterns. This includes the case where subsequent accesses are uniformly separated by a common distance, and also the case where subsequent accesses increase in distance progressively. These cases could easily occur in case of matrix accesses, and the next access distance would most likely be a function of the current distance. Though an IP based indexing scheme might be able to identify the stride in case of a uniform distance based access pattern, the same can be more efficiently captured in a distance based scheme in a significantly lower space. Further, the progressively increasing data access are more efficiently captured in the policy implemented by us, where the indexing is done using distances. A few limitations of the policy include the fact that incase the access pattern changes, the distance based policy might take additional time to learn and adapt to the modified distance patterns, as opposed to a IP indexed policy. As a result of which, at every switch in the access distance pattern, there would be a performance degradation until the table is updated with the values corresponding to the new access pattern.

Trace	No	Next Line	IP-stride	Dynamic Stide	Mixed Stride
lbm	7648362	8121962	8832662	8330639	9768312
gcc	627784	1034239	738378	667397	662283
omnetpp	1833140	2718729	1887648	1837642	1772130
perlbench	1144	8983	3148	2974	7827
mcf	18681943	21562570	24937070	19208787	20660086
wrf	216531	228805	301143	236367	259184
xalancbmk	717760	1073389	739197	749707	1128602
cam4	1077063	1175701	1125634	1089741	1105910
leela	381849	518925	1188028	505790	514632
deepsjeng	44274	50926	44280	43595	45895
fotonik3d	2643590	4609355	5010921	2621530	2189227
roms	222373	344626	973653	335126	347819
nab	187184	339267	199068	514632	425476
xz	71612	92232	99617	79212	80813

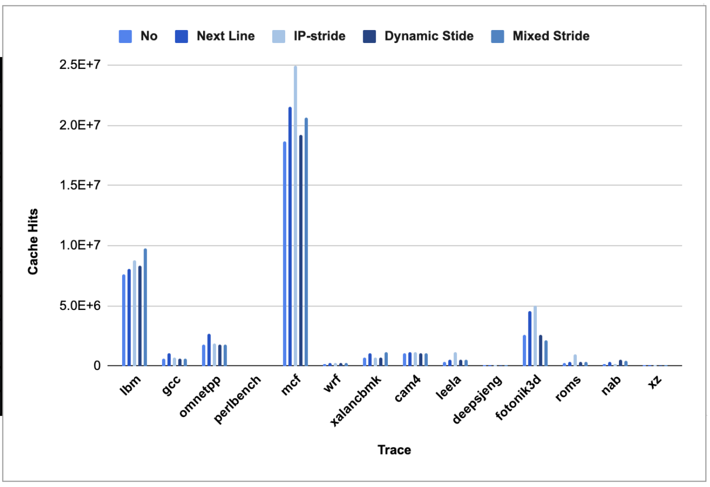

Figure 2: LLC Hits Benchmarks for the different prefetching policies, and our custom policies

3.2 Mixed Dynamic Stride Prefetching

In order to deal with the limitations of the above implemented policy, we propose a Mixed Dynamic Stride based Prefetching policy. In this policy, we employ double prefetching at every access - one based on the Distance based prefetching described above, and the other a IP-based prefetching. As indicated in the Figure-4, in case of the IP-based prefetching, we maintain another distance table containing entries corresponding to the distance between consecutive addresses, that is indexed using a hash of the current IP. In addition to the prefetching performed for the block indicated by the first table, we also prefetch another block with the stride indicated by this IP-indexed table. Once completed, this table is updated with the current access distance against the entry indexed by the current IP.

As discussed previously, the Distance based stride can cater to the varying data access patterns that could be encountered in a program, while the IP-based prefetching can adapt to change in access patterns more quickly. As a result, we envisage that a prefetching scheme based on both of these would complement each other, and provide the best of both worlds. However, a limitation of this policy is that in case of a change to irregular access pattern, we would be prefetching twice the amount of unnecessary data onto the cache, which might lead to unnecessary eviction of useful data.

Trace	No	Next Line	IP-stride	Dynamic Stide	Mixed Stride
lbm	0.208971	0.206322	0.20778	0.216827	0.216554
gcc	0.84981	0.860744	0.847892	0.843032	0.784563
omnetpp	0.539954	0.554404	0.539688	0.527842	0.437652
perlbench	1.90475	1.91209	1.90465	1.90598	1.90738
mcf	0.126914	0.125675	0.124774	0.117928	0.0926765
wrf	1.14056	1.14241	1.13962	1.13859	1.13102
xalancbmk	0.879273	0.881142	0.879059	0.877015	0.854537
cam4	1.03903	1.04259	1.03877	1.03587	1.01788
leela	0.68498	0.782955	0.682173	0.769358	0.769097
deepsjeng	1.25014	1.25081	1.25014	1.24948	1.25031
fotonik3d	0.530968	0.495981	0.498283	0.521246	0.280944
roms	1.18849	1.35481	1.17985	1.31701	1.29439
nab	1.30716	1.3093	1.30705	1.30694	1.30579
xz	1.30607	1.31679	1.30574	1.31349	1.31226

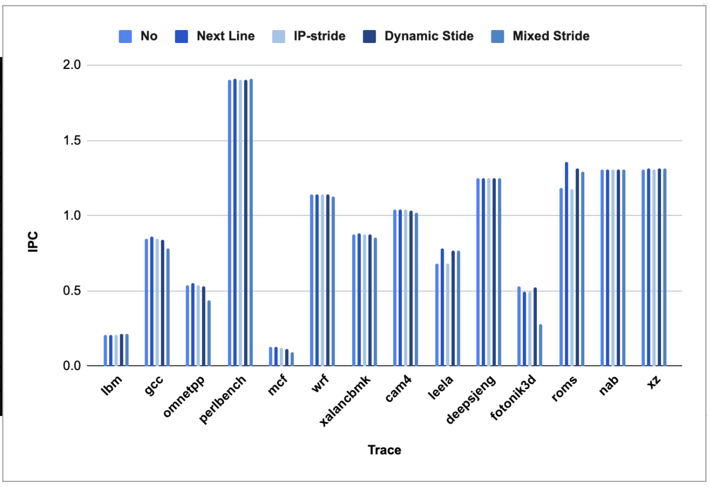

Figure 3: IPC Benchmarks for the different prefetching policies, and our custom policies

3.3 Some other experiments

We also implemented a Confidence based Dynamic Prefetcher, where multiple cache blocks are prefetched, if a regularity in the data access pattern is observed. However, for the given traces, this policy performed poorly, and hence wasn't profiled. Nevertheless, this is an idea that might be worth exploring, and could be adaptively integrated with other policies. We also attempted profiling the performance of SOTA prefetching policies, such as, the work by Pakalapati et al.[2] - however, potentially due to portability issues, these perform poorer than no prefetching on the given traces.

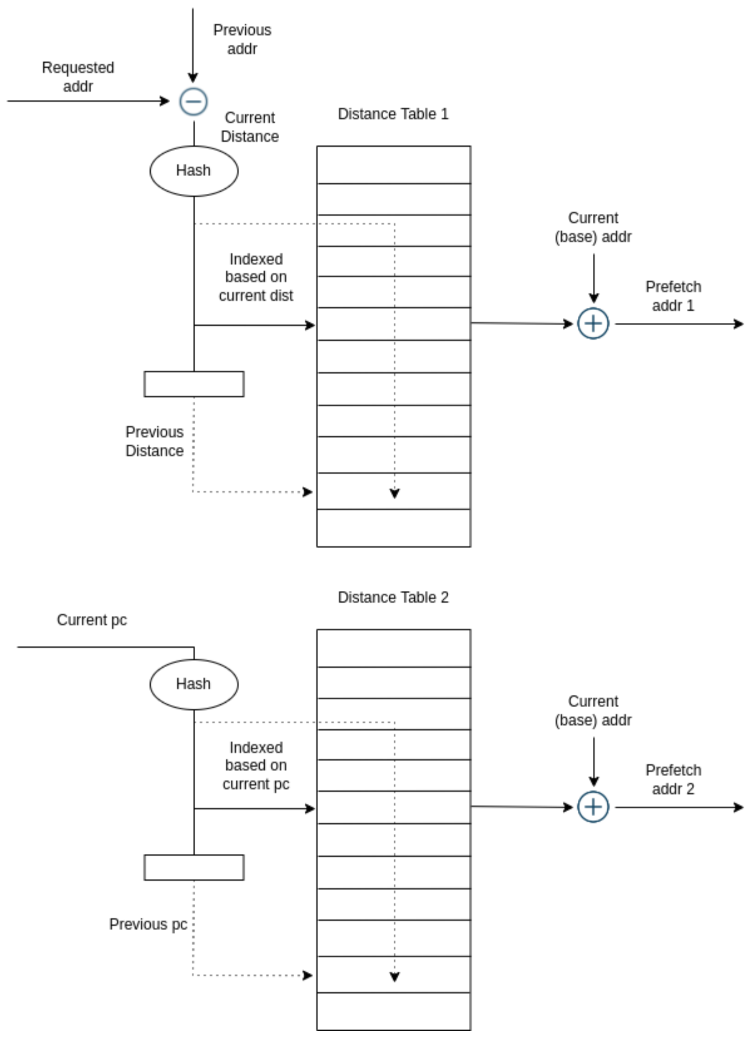

Figure 4: Mixed Dynamic Stride Prefetching Policy

4 Conclusion

In this assignment, multiple prefetching policies have been benchmarked against the provided traces. The following inferences were drawn:

- From the profiling, we could clearly see that the performance of the prefetching policy is dependent on the program corresponding to the traces, and the underlying data access patterns.
- The naive next-line prefetching policy seems to be working suprisingly well for the given traces. Similarly, the SOTA policies do not give promising results on the given traces, which might be due to inconsistencies with the versions of the ChampSim simulator.
- Our proposed dynamic stride based policy and mixed stride policy perform comparably against the baseline policies, and outperform others for some of the traces.

5 References

[1] Gokul Kandiraju, & Anand Sivasubramaniam, "Going the Distance for TLB Prefetching: An Application-driven Study"
[2] Samuel Pakalapati & Biswabandan Panda, "Bouquet of Instruction Pointers: Instruction Pointer Classifier-based Hardware Prefetching"