Sangam: A Multi-component Core Cache Prefetcher*

Mainak Chaudhuri
Indian Institute of Technology Kanpur

Nayan Deshmukh
Samsung Electronics Co.

ABSTRACT

Data prefetching is an important technique used in all commercial processors. Data prefetchers aim at hiding the long data access latency. In this paper, we present the design of an L1 cache prefetcher that employs three different components organized in a hierarchical manner to address the diversity of access patterns. Each level of the prefetcher uses different amounts of control-flow and data-flow information. The first level of the prefetcher exploits the joint feature of instruction pointer (IP) and the current delta (i.e., control-flow as well as data-flow) in the access stream sourced by the current IP to predict a sequence of deltas with high confidence. To maintain high coverage, when the IP-delta-based prefetcher is unable to offer a high-confidence prediction, we invoke the second level of the prefetcher which employs an IP-based (i.e., only control-flow-based) stride predictor. If the IP-based stride predictor is also unable to discover any stride pattern with confidence, a next-line prefetcher with adaptive degree (only data-flow-based) is triggered at the third level of the prefetcher. The same three-level prefetcher, incorporated at the L2 cache, however, offers marginal benefits when working together with the L1 cache prefetcher. We also augment our three-level L1 cache prefetcher with a technique to optimize L1 cache lookup bandwidth consumed by the prefetches. Additionally, our proposal incorporates a novel technique to maintain the aggressiveness of the L1 cache prefetcher even at the time of resource shortage in the L1 cache controller. Averaged over 46 single-thread SPEC CPU 2017 traces, our proposal achieves a 40.3% speedup over no prefetching. On a four-core configuration, averaged over 100 four-way multi-programmed workloads, our proposal achieves a 19.5% speedup over no prefetching.

1. INTRODUCTION

The widening gap between the processor and memory speeds has motivated the processor designers to explore techniques for hiding the long data access latency. Data prefetching is one of the prominent techniques in this category. The data prefetching problem seeks to infer the upcoming addresses in the demand access stream of an application and prefetch the data blocks residing at those addresses into the processor cache hierarchy. In a multi-level cache hierarchy of today's processors, a prefetcher, depending on the design, can bring data blocks into different levels of the cache hierarchy. However, since the L1 cache is the closest to the processor and the smallest among all the cache levels, it is important to design efficient L1 cache prefetchers with high accuracy and good coverage. Fetching future demand blocks into the L1 cache in a timely manner can offer the best performance. Additionally, we observe that the L1 cache is also the best place for learning any access pattern because an unfiltered access stream can be observed at the L1 cache making the learning process most reliable. For example, in a three-level cache hierarchy with L1, L2, and L3 caches, an IP-based stride prefetcher of degree four incorporated in the L1 cache achieves 30.2% speedup averaged over 46 single-threaded SPEC CPU 2017 traces. The same prefetcher incorporated in the L2 cache achieves only 24.9% speedup. There are two reasons for this speedup gap. First, the L2 cache prefetcher learns the stride pattern from a filtered access stream leading to unreliable learning of strides. Second, the prefetched blocks are not brought all the way to the L1 cache. The first shortcoming of erroneous learning can be addressed by incorporating the prefetcher in the L1 cache, but continuing to bring the prefetched blocks only up to the L2 cache. This optimization improves the speedup achieved by the aforementioned IP-based stride prefetcher to 27.4%. Motivated by these observations, we focus on designing an L1 cache prefetcher that brings prefetched blocks all the way to the L1 cache.

Our proposed prefetcher employs three different components organized in a hierarchical fashion exploiting different amounts of control-flow and data-flow information in the components. The components are an IP-delta joint feature-based delta sequence predictor, an IP-based stride prefetcher, and an adaptive-degree next-line prefetcher. We further augment our L1 cache prefetcher with two important techniques. The first one is a technique to optimize L1 cache lookup bandwidth consumed by the prefetches, while the second one is a novel mechanism to maintain the aggressiveness of the L1 cache prefetcher even at the time of resource shortage in the L1 cache controller. Overall, averaged over 46 single-thread SPEC CPU 2017 traces, our proposal achieves a 40.3% speedup over no prefetching. On a four-core configuration, averaged over 100 four-way multi-programmed workloads, our proposal achieves a 19.5% speedup over no prefetching.

2. RELATED WORK

Recent years have seen a sizable volume of contributions in the area of data prefetching with deep lookahead and improved timeliness [8, 11, 13, 14]. These proposals typically design delta predictors that can lead to a prefetch sequence. Another body of work has explored instruction as well as data prefetching techniques in the context of server workloads [3, 4, 5, 15, 16, 17, 18, 19]. Prefetchers with thread-awareness have been explored for multi-threaded workloads [2, 10]. Specialized prefetchers for irregular memory accesses have been proposed [7]. Criticality-aware prefetching has been explored in the context of a multi-level cache hierarchy [12]. Prefetching and caching policies that do not rely on instruction pointers have been researched [9]. Further, prefetch-aware caching policies have been proposed [6, 20].

3. DESIGN OF PROPOSED PREFETCHER

Our proposal has three component prefetchers, namely an IP-delta-based delta sequence predictor, an IP-based stride predictor, and a next line prefetcher with adaptive degree. The overview of the prefetcher flow is shown in Figure 1. The L1 cache prefetcher

* Sangam means a confluence of rivers, particularly between the Ganges, the Yamuna, and the Rigvedic metaphysical river Sarasvati in the Northern Indian city of Allahabad.

components are looked up on every demand access to the L1 cache and a prefetch sequence is generated. A common base prefetch degree d is maintained for all the three components meaning that at most d prefetches would be injected on every L1 cache demand access. We discuss the details in the following.

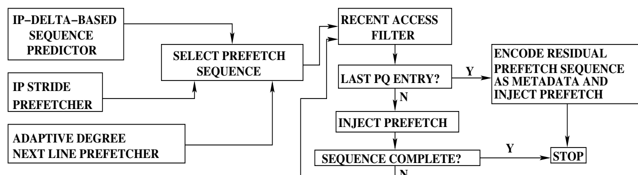

Figure 1: Overview of L1 cache prefetcher. PQ stands for the prefetch request queue.

3.1 IP-Delta-based Sequence Predictor

The IP-delta-based sequence predictor first categorizes all demand accesses based on their source IP. Next, for a given access coming from a particular source IP and having a delta relative to the last access from the same IP, the predictor predicts a sequence of next d deltas. A delta between two accesses is defined as the difference between the offset of the current access (i.e., cache block number within a page) and the offset of the last access. The motivation of using this predictor arises from the observation that within the accesses coming from a particular IP, the sequence of deltas appearing after a given delta Δ is often repeated after the same delta Δ e.g., $\Delta, \delta_1, \delta_2, \delta_3, \delta_4, \ldots, \Delta, \delta_1, \delta_2, \delta_3, \delta_4, \ldots$.

This predictor requires two tagged set-associative tables. The first table, referred to as the IP table (the tables are named after the entity used to index into the table), is looked up using the IP of the current demand access. Each entry of the IP table stores a FIFO list of the last seen $d+1$ deltas corresponding to the index IP. It also stores the last offset to enable computation of the next delta. Additionally, each entry has a partial tag, valid bit, and LRU states. If there are N cache blocks within a page, we encode each delta using $1+\log_2(N)$ bits where the most significant bit of a delta encodes its sign (sign-magnitude representation of delta). The second table, referred to as the IP-delta table, is looked up using a concatenation of the IP of the current demand access and the current delta within the access stream sourced by that IP. Each entry of this table stores a sequence of d deltas seen right after the delta used for indexing into the table, given the source IP of the index. Each of the d deltas also has a two-bit confidence counter. Each entry also stores a partial tag, a valid bit, and LRU states. This table is updated as follows. First, the IP table is looked up with the current IP and the corresponding FIFO list of $d+1$ deltas is updated (a new delta is inserted at the tail and the oldest one is replaced). Next, for each k, the kth delta of the FIFO list is used ($k=0$ being the oldest) along with the IP to index into the IP-delta table. The corresponding IP-delta table entry's $(d-k)^{\text{th}}$ delta is updated with the newly inserted delta in the FIFO list. If the new delta matches the delta at position $(d-k)$ of the IP-delta table entry, the confidence counter at that position is incremented by one using saturating arithmetic; otherwise the confidence counter at that position is reset to zero.

A prediction is obtained by looking up the IP-delta table using the current IP and delta (within the accesses sourced by the current IP). The delta sequence is read out from the matching IP-delta table entry in the case of a hit. If a delta in the read-out sequence has confidence below a threshold τ_c, that delta is predicted as zero. This delta sequence is used to generate a prefetch sequence until a zero delta is encountered. When a zero delta is encountered and the number of generated prefetches is below d, it is checked whether the last two non-zero deltas in the delta sequence are identical. If they are, prefetching continues with that delta; otherwise the prefetching sequence is terminated prematurely. In the case of a miss in the IP-delta table or IP table, no prediction can be generated. We observed that using a hash of the last few IPs along with the last few deltas for indexing into the IP-delta table does improve the accuracy of the predictor, but lowers the table hit rate drastically. As a result, we stick to the single IP and single delta-based indexing scheme for the IP-delta table.

3.2 IP-based Stride Prefetcher

We leverage the stride prediction from the IP table when the IP-delta-based sequence predictor cannot offer a prediction due to IP-delta table misses or low-confidence predictions. This strategy avoids loss in coverage. This prediction comes for free without any additional overhead, since the IP table has to be looked up anyway. After the FIFO list of the matching IP table entry is updated by inserting the current delta, if the last (youngest) two deltas in the FIFO list are identical, this delta is used for generating a prefetch sequence of length d in the cases when the IP-delta table cannot offer a prediction due to miss or low confidence. We also employ this delta predicted from the IP table to complete an otherwise prematurely terminated prefetch sequence due to a low confidence delta in the IP-delta table's predicted sequence, as already discussed.

3.3 Next-line Prefetcher

The IP-delta table prediction and the IP table's stride prediction incorporate an inherent notion of confidence, which is very important for an L1 cache prefetcher to avoid polluting the small cache. In the cases where none of these prediction mechanisms is able to offer a prediction, we employ a next-line prefetcher. With the help of a feedback mechanism, we avoid polluting the L1 cache with inaccurate next-line prefetches. Our proposal inserts all next-line prefetch candidates up to degree d in a small fully-associative next-line buffer (NL buffer). Each entry of this buffer holds a tag, the prefetch degree ($\log_2(d)$ bits) that resulted in the insertion of the tag, a valid bit, and LRU states. Every demand access looks up this buffer for a matching tag. For each prefetch degree $\tilde{d}(\leq d)$, we maintain two counters, one for counting the number of NL buffer insertions at degree $\tilde{d}$ and one for counting the number of demand hits to entries inserted at degree $\tilde{d}$. When a demand access hits an entry, it increments the hit counter corresponding to the degree recorded in the matching entry. At this time, the matching entry is invalidated from the NL buffer. At the time of inserting a prefetch candidate of degree $\tilde{d}$ in the NL buffer, the corresponding insertion counter is incremented if the inserted tag is not already present in the NL buffer. If the tag to be inserted already exists in the buffer and the degree D recorded in the matching buffer entry is more than $\tilde{d}$ (meaning that a higher degree prefetch inserted this tag in the past), we set the degree in the entry to $\tilde{d}$, increment the insertion counter corresponding to degree $\tilde{d}$, and decrement the insertion counter corresponding to the old higher degree D. This gives priority to lower degree prefetches avoiding large lookahead in the next-line prefetcher, which can be quite inaccurate. When generating a next-line prefetch at a distance (or degree) of $\tilde{d}$ from the base demand address, the ratio of the hit and insertion counters of degree $\tilde{d}$ is looked up and if the ratio is above a threshold τ_d, then only the prefetch is injected into the prefetch queue.

3.4 Recent Access Filter

Ideally, every prefetch request should be accurate and should generate a new miss from the L1 cache. This makes sure that the L1 cache lookup bandwidth consumed by the prefetches is not wasted. Unfortunately, it is not easy to figure out which prefetches will hit in the L1 cache without looking up the cache. We approximately answer this question by maintaining a small fully-associative Recent Access buffer for keeping track of the recently seen tags in the demand access stream and the recently inserted prefetches. These tags are most likely to stay in the L1 cache or the L1 miss status holding registers (for the misses). A newly generated prefetch request looks up this buffer and in the case of a hit, the prefetch is dropped. We note that it is important to be conservative in this estimate because dropping a genuine prefetch that would have missed in the L1 cache lowers coverage. Therefore, the Recent Access buffer is kept small.

3.5 Handling Resource Shortage

The L1 cache controller is usually not equipped with resources to prefetch aggressively. One important resource used by the prefetches is the prefetch request queue (PQ)[1]. A new prefetch is inserted at the tail of the PQ. With a small PQ, it is quite possible that the prefetch aggressiveness can get throttled down due to frequently full PQ. One option is to move the prefetcher to the L2 cache where the resources are plenty. However, as already pointed out, learning access patterns from a filtered stream as seen by the L2 cache is not reliable. We propose a simple solution to this problem that leverages communication between the L1 cache prefetcher and the L2 cache. While injecting prefetches, the L1 cache prefetcher checks the occupancy of the PQ. If only one slot is left, it injects one more prefetch and with it piggybacks the information regarding the residual prefetches, if any, that it could not inject due to shortage of PQ entries. The L2 cache prefetcher inspects the piggybacked information (passed as a 32-bit metadata encoded in the prefetch packet), decodes the information, and injects the residual prefetches. These prefetches will be filled up to the L2 cache and will not propagate to the L1 cache. This piggybacking facility is used for IP-delta-based sequence prefetching and IP-based stride prefetching. It is not used if the next-line prefetcher runs out of PQ space, since this prefetcher component is, in general, far less accurate. The piggybacked information uses one of the two possible encodings: a sequence of deltas (for residual IP-delta-based sequence prefetching) and a constant delta (for residual IP-based stride prefetching). The most significant bit of the 32-bit metadata is used to distinguish between these two encodings. The remaining 31 bits encode either a sequence of deltas or just one stride.

3.6 L2 Cache Prefetcher

While an accurate and high-coverage L1 cache prefetcher alone can be quite effective, an L2 cache prefetcher triggering further prefetches on L1 cache misses can offer very deep lookahead overall. However, a very deep lookahead seldom remains accurate. Nonetheless, we replicate our L1 cache prefetcher at the L2 cache also with two simplifications as outlined in the following. First, there is no Recent Access buffer in the L2 cache prefetcher, since the L2 cache access bandwidth is plenty thanks to the L1 cache hits. Second, the L2 cache does not piggyback any residual prefetches to the L3 cache primarily because this facility is seldom needed given the reasonably large L2 cache PQ. We note that the L2 cache prefetcher, on a prefetch access, may have to first inject the residual prefetches coming in the encoded format from the L1 cache and then invoke the L2 cache prefetcher for injecting more prefetches. In both cases, the L2 cache uses a base prefetching degree and the overall number of prefetches can be up to twice the base degree.

[1] The shortage of PQ space hides shortage of miss status holding registers which are used later in the prefetching pipeline.

3.7 Storage Overhead

We calculate the storage overhead of our prefetcher assuming 4 KB pages and 64-byte cache blocks. Therefore, there are 64 cache blocks per page and hence, each delta can be encoded in seven bits using sign-magnitude representation. We set the L1 cache base prefetching degree to four and the L2 cache base prefetching degree to three and two respectively for single- and multi-core. Each entry of the IP table is restricted to 63 bits by appropriately sizing the partial tags. Similarly, each entry of the IP-delta table is restricted to 64 bits. Each entry of the NL buffer is 73 bits, while each entry of the Recent Access buffer is 71 bits. Tables 1 and 2 show the storage overheads of the L1 and L2 cache prefetchers, respectively. The total overhead per core is 63.1 KB. Our proposal uses two threshold parameters, the values of which are summarized in Table 3.[2]

Table 1: Storage overhead of L1 cache prefetcher

Structure	Storage
IP table	128 sets, 15 ways, total bits = 120960
IP-delta table	256 sets, 8 ways, total bits = 131072
NL buffer	64 entries, total bits = 4672
Recent Access buffer	40 entries, total bits = 2840
Auxiliary counters and registers	316 bits
TOTAL	259870 bits

Table 2: Storage overhead of L2 cache prefetcher

Structure	Storage
IP table	128 sets, 15 ways, total bits = 120960
IP-delta table	256 sets, 8 ways, total bits = 131072
NL buffer	64 entries, total bits = 4672
Auxiliary counters and registers	248 bits
TOTAL	256952 bits

Table 3: Threshold values

Threshold	Reference	L1 prefetcher	L2 prefetcher
τ_c	Section 3.1	Single-core: 2 Multi-core: 3	Single-core: 2 Multi-core: 2
τ_d	Section 3.3	$\frac{1}{4}$	$\frac{1}{2}$

4. SIMULATION RESULTS

We evaluate our proposal on the ChampSim DPC3 infrastructure. As per the published official rules, we use the perceptron branch predictor and LRU replacement policy in the L3 cache. The L3 cache has no prefetcher. We use 46 single-thread traces captured from the SPEC CPU 2017 applications for the single-core evaluation. These traces have L3 cache MPKI of at least 1.0 in the baseline configuration without any prefetcher. We evaluate our proposal in a four-core configuration using 100 four-way multi-programmed

[2] Source code with tuned parameter values is available at https://www.cse.iitk.ac.in/users/mainakc/sangam_tuned.html.

workloads (45 homogeneous[3] and 55 heterogeneous prepared by uniform random mixing of traces).

Figure 2 shows the speedup achieved by our proposal on 46 single-thread traces. A bar represents the speedup of a trace. We have categorized the traces application-wise. The average speedup is 40.3%. Figure 3 shows the gradual improvement in speedup as different components are added. The speedup figure for each bar is an average over 46 single-thread traces. The leftmost two bars quantify the speedup of adding an IP-based stride prefetcher (L21) and an IP-delta-based sequence prefetcher (L22) in the L2 cache. The bars D1 and D2 respectively show the speedup when these two prefetchers are incorporated in the L1 cache. These results clearly show that adding a prefetcher in the L1 cache is better than adding it to the L2 cache. Next, we gradually add different components of our proposal to the design D2 which is an IP-delta-based sequence prefetcher. The design D3 employs the IP-based stride predictor on top of D2 when the IP-delta-based sequence prefetcher cannot offer a prediction. The design D4 incorporates the recent access filter on top of D3. The design D5 employs piggybacking of residual prefetch information to the L2 cache on top of design D4. The design D6 introduces a next-line prefetcher with best static prefetch degree on top of D5. This prefetcher is triggered when none of IP-delta-based and IP-based prefetchers can offer a prediction. The design D7 replaces the oracle best static degree by the adaptive degree next-line prefetcher incorporating the NL buffer. Finally, the design D8 incorporates the L2 cache prefetcher. As can be seen, the speedup improves gradually as each optimization is incorporated justifying its inclusion in the design. The L2 cache prefetcher brings marginal benefits on top of the fully optimized L1 cache prefetcher represented by the design D7. The right side of Figure 3 evaluates five L2 cache prefetchers (C1 to C5) for comparing against the performance of Sangam. C1, C2, and C3 are the top three performers from DPC2 [1], while C4 and C5 are two recently proposed L2 cache prefetchers—signature path prefetcher (SPP) [8] and kill the program counter prefetcher (KPCP) [9]. All these single-component L2 cache prefetchers fall significantly short compared to Sangam.

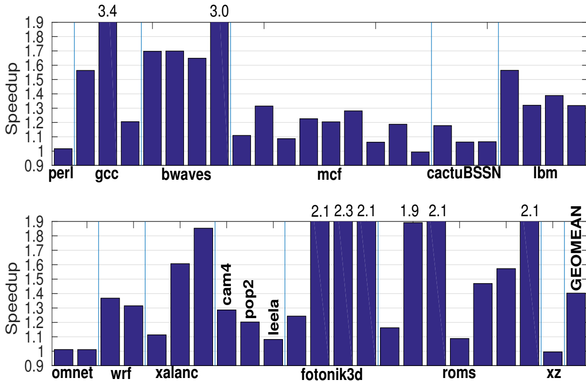

Figure 2: Single-core speedup.

On a four-core configuration, our proposal achieves an average speedup of 19.5% over 100 workloads. For 45 homogeneous workloads, the speedup is 10.2%, while that for the 55 heterogeneous workloads is 27.7%.

5. SUMMARY

We have presented the design of a multi-component core cache prefetcher focusing primarily on the L1 cache prefetcher. The prefetcher incorporates three distinct address prediction components along with two optimizations targeting conservation of L1 cache lookup bandwidth and maintaining the aggressiveness of the prefetcher in the face of resource shortage. Our proposal achieves an average 40.3% speedup on 46 single-thread traces and 19.5% speedup on 100 four-way multi-programmed workloads.

[3] One trace of fotonik3d runs into a deadlock when run in rate mode.

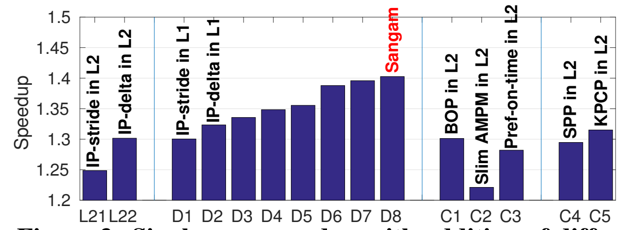

Figure 3: Single-core speedup with addition of different features and comparison with L2 cache prefetchers.

6. REFERENCES

[1] The 2nd Data Prefetching Championship (DPC2). http://comparch-conf.gatech.edu/dpc2/.

[2] L. M. AlBarakat, P. V. Gratz, D. A. Jimenez. MTB-Fetch: Multithreading Aware Hardware Prefetching for Chip Multiprocessors. In *Computer Architecture Letters*, **17**(2): 175-178 (2018).

[3] M. Ferdman and B. Falsafi. Last-Touch Correlated Data Streaming. In *ISPASS* 2007, pages 105–115.

[4] M. Ferdman, C. Kaynak, and B. Falsafi. Proactive Instruction Fetch. In *MICRO* 2011, pages 152–162.

[5] M. Ferdman, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. Temporal Instruction Fetch Streaming. In *MICRO* 2008, pages 1–10.

[6] A. Jain and C. Lin. Rethinking Belady's Algorithm to Accommodate Prefetching. In *ISCA* 2018, pages 110–123.

[7] A. Jain and C. Lin. Linearizing Irregular Memory Accesses for Improved Correlated Prefetching. In *MICRO* 2013, pages 247–259.

[8] J. Kim, et al. Path Confidence based Lookahead Prefetching. In *MICRO* 2016.

[9] J. Kim, et al. Kill the Program Counter: Reconstructing Program Behavior in the Processor Cache Hierarchy. In *ASPLOS* 2017, pages 737–749.

[10] P. Liu, J. Yu, and M. C. Huang. Thread-Aware Adaptive Prefetcher on Multicore Systems: Improving the Performance for Multithreaded Workloads. In *TACO* **13**(1): 13:1–13:25 (2016).

[11] P. Michaud. Best-offset Hardware Prefetching. In *HPCA* 2016, pages 469–480.

[12] A. Nori, et al. Criticality Aware Tiered Cache Hierarchy: A Fundamental Relook at Multi-Level Cache Hierarchies. In *ISCA* 2018, pages 96–109.

[13] S. H. Pugsley, et al. Sandbox Prefetching: Safe Run-time Evaluation of Aggressive Prefetchers. In *HPCA* 2014, pages 626–637.

[14] M. Shevgoor, et al. Efficiently Prefetching Complex Address Patterns. In *MICRO* 2015, pages 141-152.

[15] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi. Spatio-Temporal Memory Streaming. In *ISCA* 2009, pages 69–80.

[16] S. Somogyi, T. F. Wenisch, M. Ferdman and B. Falsafi. Spatial Memory Streaming. In *J. Instruction-Level Parallelism*, vol. 13 (2011).

[17] T. F. Wenisch, et al. Practical Off-chip Meta-data for Temporal Memory Streaming. In *HPCA* 2009, pages 79–90.

[18] T. F. Wenisch, et al. Temporal Streaming of Shared Memory. In *ISCA* 2005, pages 222–233.

[19] T. F. Wenisch, et al. Store-Ordered Streaming of Shared Memory. In *PACT* 2005, pages 75–86.

[20] C.-J. Wu, et al. PACMan: Prefetch-aware Cache Management for High Performance Caching. In *MICRO* 2011, pages 442–453.