2017 IEEE 35th International Conference on Computer Design

SelSMaP: A Selective Stride Masking Prefetching Scheme

Jiajun Wang, Reena Panda and Lizy Kurian John
The University of Texas at Austin
{*jiajunwang, reena.panda*}*@utexas.edu, ljohn@ece.utexas.edu*

***Abstract*—Although prefetching concepts have been proposed for decades, new challenges are introduced by sophisticated system architecture and emerging applications. Large instruction windows coupled with out-of-order execution makes program data access sequence distorted from cache perspective. Big data applications stress memory subsystems heavily with their large working set sizes and complex data access patterns. To address such challenges, this work proposes a high performance hardware prefetching scheme, SelSMaP. SelSMaP is able to detect both regular and non-uniform stride patterns by taking the minimum observed address offset (called a reference stride) as a heuristic. We evaluated SelSMaP with CloudSuite workloads and SPEC CPU2006 benchmarks. SelSMaP achieves an average CloudSuite performance improvement of 30% over non-prefetching system. With one to two order of magnitude less storage and much less functional logic, SelSMaP outperforms the highest-performing prefetcher by 8.6% in CloudSuite workloads.**

I. Introduction

Emerging applications pose additional challenges to hardware prefetcher designs. Prior research [1] has shown that predicting streaming or uniform stride access behavior alone is not sufficient to improve memory subsystem efficiency for emerging applications such cloud workloads. Only less than 30% of global memory reference stream exhibit uniform stride access pattern in CloudSuite workloads. Prefetching schemes that are able to detect both uniform and non-uniform stride patterns becomes essential. Also, prior research studies [1], [2] have shown that cache capacity sensitive applications are prone to get negatively affected by useless prefetch requests. Both over-prefetching and address misprediction can generate useless prefetch requests. Useless prefetch requests not only waste memory bandwidth and cache capacity, but also cause additional cacheline evictions and result in extra cache misses. For application whose data accesses exhibit long temporal reuse distance meanwhile still benefit from caching, cacheline insertion of useless prefetched blocks will lead to cache thrashing under which scenario cache misses rockets. Therefore, an efficient data prefetching scheme should be able to avoid over-prefetching.

In order to address these challenges, we propose SelSMaP, a **Sel**ective **S**tride **Ma**sking **P**refetching scheme. SelSMaP is able to detect both regular and non-uniform stride patterns through leveraging a selective stride mask based on the minimum observed address offset between two consecutive accesses (called a reference stride).

Uniform stride pattern detection becomes challenging when cache access order is different from uniform program access order. For example, a software developer arranges cacheline accesses "A, A+2, A+4, A+6, A+8, A+10, A+12" in program order. After going through out-of-order execution engine, cache access order may become "A+4, A, A+2, A+6, A+10, A+8, A+12". Thus the observed cacheline address offset sequence is not uniform, but may become "-4, +2, +4, +4, -2, +4". To tackle this problem, SelSMaP takes a reference stride as a heuristic, and generates a stride mask based on the reference stride. The rationale behind the reference stride is that the real stride will never be more than the minimum observed address offset between two consecutive accesses. The minimum observed offset, which is 2 in the example, is chosen as the reference for pattern detection in the SelSMaP.

Non-uniform stride or multi-delta pattern can be captured by SelSMaP if it can be rephrased as multiple uniform stride accesses. For example, a memory access stream of "B, B+1, B+10, B+2, B+3, B+11, B+12, B+4, B+5" is observed in the SPEC CPU2006 milc benchmark, which contains irregular delta sequence of "+1, +9, -8, +1, +8, +1, -8, +1". With the help of the minimum observed delta and stride mask with fixed window range, SelSMaP is able to extract two regular (stride of one) streams with base address B and B+10 from original memory access stream.

Self-trained prefetching degree is achieved at the granularity of individual prefetch request generation in SelSMaP. Some prefetchers adjust prefetch degree based on the feedback of cache miss rate variation, which is an accumulated affect from previous program phase. However, such approaches fail to address the presence of multiple patterns in a phase. A prefetching scheme may be highly confident in detecting one stream and less efficient in other coexistent streams, and ideally should apply high prefetch degree to the former stream and low degree to the latter stream. Adopting same prefetching degree to both streams would result in either losing prefetching opportunity or generating useless prefetch requests. Therefore, a finer granularity of prefetch degree control is essential. SelSMaP meets such requirement by evaluating the confidence of every individual prefetch request and adapting prefetch degree based on the confidence.

We evaluated SelSMaP using both single-threaded SPEC CPU2006 benchmarks and multicore CloudSuite workloads.

1063-6404/17 $31.00 © 2017 IEEE
DOI 10.1109/ICCD.2017.63

Results show IPC improvement of average 30% compared to a non-prefetching baseline. Comparison of SelSMaP with state-of-art prefetchers show average 10% performance improvements in CloudSuite applications with less hardware. The rest of this paper is organized as follows: Section II describes SelSMaP architecture. Section III presents our design evaluation. We conclude our work in section IV.

II. SelSMaP Design

Figure 1 demonstrates a multi-core system with SelSMaP serving as LLC prefetcher. SelSMaP is composed of four function units: a Stride Reference Table(SRT), a Stride Mask Logic(SML), a Decision Making Logic(DML), and a Prefetch Address Computation (PAC) logic. As shown in the figure, SelSMaP monitors all LLC accesses and maintains access history in the SRT. Apart from holding access history information, SRT makes stride pattern prediction (i.e., the stride reference) and feeds it into SML to generate a stride mask. With the stride mask and access history, DML evaluates the confidence of the predicted pattern. If DML decides to trigger prefetching based on the stride reference, then PAC is used to compute prefetching address(es). With the help of these four units, SelSMaP generates prefetch requests for LLC. In this section, we are going to introduce the structure and functionality of these four units.

A. Stride Reference Table (SRT)

SRT holds access history information and generates reference stride. SRT is a set associative structure, which is indexed by a region tag obtained from the upper bits of data address. Each SRT entry keeps track of recent memory accesses within an address region, which is a fixed size memory space. An SRT entry consists of four fields. The *Region Tag* field indicates access information about which region is held in that entry. The *Stride Reference* field holds a speculative stride pattern of that region. The *Previous Access* field saves a partial address indicating the latest accessed cacheline. The *Access History* field records which cachelines within that region have been accessed over time.

Constant data address offset has been a clear indication of regular stride data access pattern. However, prefetching scheme with solely previous and current address offset comparison becomes insufficient in modern computer architecture.One ideal case is to keep track of all the previous data accesses address offsets, but is not practical due to limited on-chip metadata storage constraint. To address the pattern detection problem as well as storage constraint, SelSMaP keeps track of the minimum address offset occurred among all data accesses to the same memory region. SelSMaP takes the minimum offset as a reference, and stores the value in the *Stride Reference* field at a cacheline granularity. In order to compute address offset, a *Previous Access* field is required to maintain address of previous data access at cacheline granularity as well.

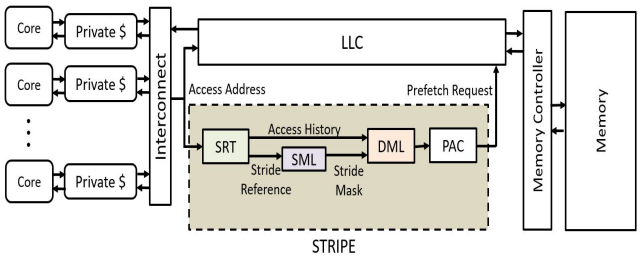

Figure 1: SelSMaP Overview

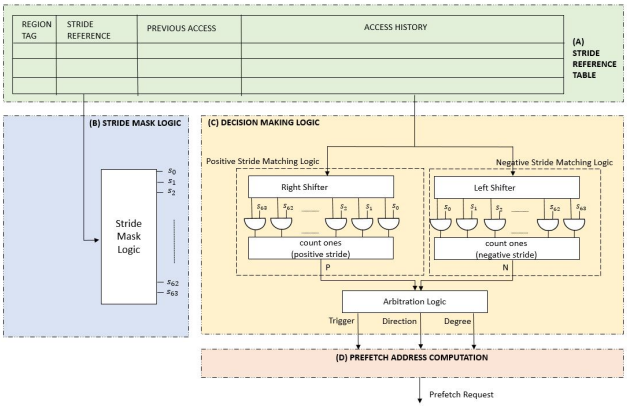

Figure 2: Stride Reference Table and Decision Making Logic

An *Access History* field is used to keep track of memory access history within a region. Access History is stored in a bit vector, where each bit is corresponding to a cacheline in a memory region. The bit position is the same as its region offset, i.e., bit 0 represents the first cache block in a region. When a new SRT entry is assigned, the *Access History* field is reset to all 0s. A bit is set when a demand request is made to the corresponding cacheline.

The size of each field depends upon the region size and the cacheline size. In this paper, our simulation system is configured with 4KB memory region, 64B cacheline, and 48-bit wide physical address. Thus, *Region Tag* is 36-bit wide, indexed by the upper 36 bits of the address, the *Previous Access* field has 6 bits, indexed by address bit [11:6], and the *Access History* field has 64 bits with 1 bit per cacheline. The *Stride Reference* field is set to be 4-bit wide. Although wider Stride Reference field can holds larger offset and the largest address offset can be 6-bit wide, we figure out that wider field does not help much, because stride larger than 16 has a maximum stride match counts of four in a 4KB region, so that the confidence level of large offset is too low to trigger prefetching.

Every time SelSMaP observes a demand access and finds an SRT region tag match, it updates the SRT. In the first step, the cacheline offset between the current and previous access is calculated. If calculated offset is smaller than the value in *Stride Reference*, the old value is replaced. In the second step, *Previous Access* is updated with the current access. In the last step, the bit corresponding to the access

is set in the *Access History* field. However, if there is no region tag match, SRT assigns a new entry where *Region Tag* is filled with the upper bits of access address; *Stride Reference* is initialized to 15; *Previous Access* is updated with the current access; and the bit corresponding to the current access is set in *Access History*.

B. Stride Mask Logic (SML)

SML generates a stride mask based on the stride reference value from SRT. SML is implemented as a lookup table, in which every reference stride is couple with a *stride mask*. A stride mask represents the partial access history pattern if predicted reference stride exists. The width of stride mask, which is 64 here, is set according to the width of access history field in the SRT. Each stride mask is formed by repeated stride patterns in binary form. The number of repetitions, which is eight in our work, is called window size. When the reference stride value is 2, every other position of the stride mask is a logic 1, or *1010101010101010b*. The reason for setting a window size is to eliminate any potential negative effects of one region containing multiple chunks of various stride access patterns. Since stride mask will be matched against real access history, setting a window size could mask out accesses from different chunks.

C. Decision Making Logic (DML)

DML evaluates the confidence of a reference stride using stride mask and makes prefetching direction. As shown in Figure 2, DML consists of two main components: a positive and negative stride matching logic, and an arbitration logic. The two stride matching logic have the same components: a shifter, a series of logic AND gates, and a count-ones logic.

Making a prefetch decision involves the following steps. In the first step, the *Access History* field is loaded into both left and right shifters. Once loaded, the history is shifted until the bit representing the current access is shifted out and 0s are shifted in. This step is to isolate accesses in the positive and negative directions.

In the second step, shift register bits and stride mask are fed into AND logics. This step is to filter the shifted access history with the help of the stride mask, so that history accesses that do not match the speculative access pattern are filtered out. Since speculative pattern is evaluated in both positive and negative direction, stride mask bits are wired in a reverse order in the positive and negative stride matching logic.

In the third step, results from previous step are fed into the corresponding count-ones logic, which counts the number of bits set to one, to generate a *P* count and a *N* count.

In the last step, the arbitration logic determines whether the speculative pattern guided by stride reference is identified, and if so, decides prefetch direction and prefetch degree. The outputs *P* and *N* from the previous step tell how many pattern matches are detected in the positive and the negative direction. If both outputs are smaller than a preset threshold, the arbitration logic considers predicted access pattern as low confidence and sets the *Trigger* bit to 0. Otherwise, the *Trigger* bit is set to 1 to trigger prefetching, whose direction will be determined by the larger of values *P* and *N*. If more positive stride matches are detected in history, *Direction* will be set to 0, and vice versa. If the sum is larger than a predefined cutoff value, a large prefetch degree is applied dynamically.

There is no need to perform decision making operations when a new SRT entry representing a newly encountered memory region is brought into SRT, since it indicates that the observed cacheline access is the first access of that region in a short period.

D. Prefetch Address Computation (PAC)

The PAC generates prefetch address(es) based on signals from DML. If the *Trigger* signal is set by DML, PAC computes the prefetching address by adding or subtracting current access address with stride reference value based on the *Direction* signal. According to the *Degree* signal, PAC may generate more than one prefetch requests.

III. Evaluation

A. Simulation Methodology and Workloads

We evaluate four prefetching schemes, AMPM, BO, stream buffer and SelSMaP. AMPM [3] and BO prefetcher are implemented based on publicly available implementations from the 1st and 2nd Data Prefetching Championship [4]. The stream buffer prefetching scheme adapts the idea of multi-entry stream buffer discussed in Palacharla's work [5]. SelSMaP is configured to consume a total storage of 476 B, which is solely the storage cost of SRT as other components can be implemented with logic only.

B. Single Core Performance Evaluation

Figure3 shows the performance of four prefetchers normalized to the baseline of non-prefetching system. Compared against baseline, SelSMaP attains an performance speedup of 1.76X among prefetch friendly benchmarks and 1.28X among all benchmarks. SelSMaP beats stream buffer across all workloads, and performs 35% better than BO among prefetch friendly workloads and on average 13% among all workloads. SelSMaP and AMPM show similar average speedups among prefetch friendly workloads. For benchmarks that benefit more from SelSMaP than AMPM, SelSMaP outperforms AMPM by 6%. Although SelSMaP is less efficient than AMPM in some workloads, the average performance difference is within 2%. SelSMaP performs better than BO in almost all workloads except for cactusADM (14% less than BO).

Table I: System Configuration

Processor	4-way out-of-order, Tournament Branch Prediction, 32 load store queue buffer
L1 cache	Private, 64KB, 4-way associative, 64B cacheline, LRU
L2 cache	Shared, 2MB, 8-way associative, 64B cacheline, LRU

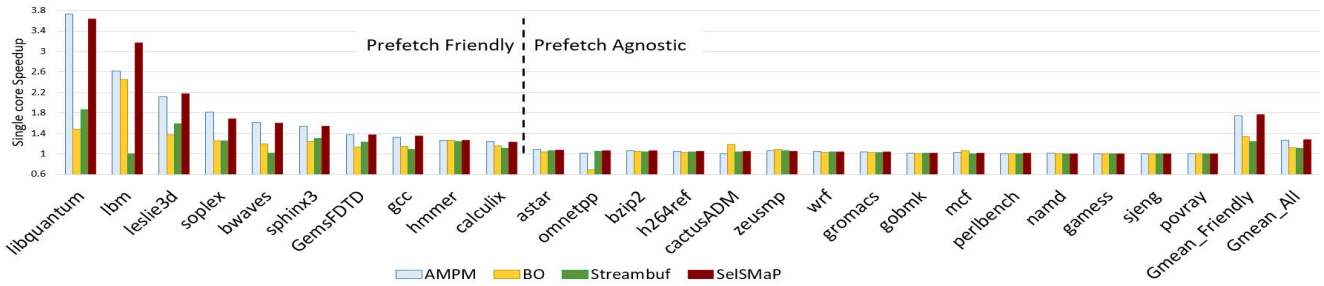

Figure 3: Single core performance normalized to the non-prefetching system

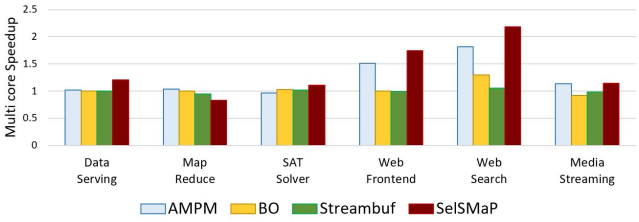

Figure 4: Normalized CloudSuite performance comparison

C. Multicore Performance Evaluation

Figure4 shows prefetching performance on cloud applications in a multicore system. Web Frontend and Web Serving are prefetch-friendly, i.e., they gain performance benefit from most of the prefetching schemes. Among all the evaluated prefetching schemes, SelSMaP shows the highest speedup in these two workloads. The benefit of fewer useless prefetch requests in SelSMaP isn't marked in SPEC benchmarks, but becomes obvious in cloud applications. Those prefetching-friendly SPEC benchmarks exhibit more regular data access compared to CloudSuite, and their dominant working sets can easily fit into an 8M LLC. Thus, useless prefetching isn't harming SPEC benchmarks seriously. However, according to workload analysis in prior work [1], CloudSuite data access pattern shows long reuse distance, and dominant working sets hardly fit into LLC. SelSMaP generates less amount of useless prefetch requests than prefetchers with similar accuracy but more aggressive(e.g. AMPM). Useless prefetch requests consumes cache space, indirectly reduce the effective LLC capacity. Therefore, CloudSuite benchmarks are less tolerant of cache pollution than SPEC.

IV. Conclusion

In this paper, we presented SelSMaP, a high-performance, low-budget LLC prefetching scheme to make prefetch decision based on a referenced stride. The reference stride is picked as the minimum observed offset of two consecutive accesses in the same page. The referenced stride is evaluated by generating a stride masking and comparing it to the access history pattern, which reveals how well matched the history pattern is to the guidance. This prefetching scheme saves on-chip area in comparison to many state-of-the-art prefetchers and uses less logic to achieve high performance.

We evaluate our prefetching scheme running CloudSuite and selected SPEC CPU2006 benchmarks. SelSMaP achieves an average performance improvement of 30% over non-prefetching system. With one to two order of magnitude less storage and much less functional logic, SelSMaP outperforms the highest-performing prefetcher by 8.6% in CloudSuite workloads.

V. Acknowledgement

The authors of this work are supported partially by SRC under Task ID 2504 and National Science Foundation (NSF) under grant number 1337393. Part of this work was collaborated with the Centaur Technology during an internship. We wish to acknowledge the computing time we received on the Texas Advanced Computing Center (TACC) system. Any opinions, findings, conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or other sponsors.

References

[1] J. Wang, R. Panda, and L. K. John, "Prefetching for cloud workloads: An analysis based on address patterns," in *2017 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2017, Santa Rosa, CA, USA, April 24-25, 2017*, 2017, pp. 163–172. [Online]. Available: https://doi.org/10.1109/ISPASS.2017.7975288

[2] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: A study of emerging scale-out workloads on modern hardware," *SIGPLAN Not.*, vol. 47, no. 4, pp. 37–48, Mar. 2012. [Online]. Available: http://doi.acm.org/10.1145/2248487.2150982

[3] Y. Ishii, M. Inaba, and K. Hiraki, "Access map pattern matching for data cache prefetch," in *Proceedings of the 23rd International Conference on Supercomputing*, ser. ICS '09. New York, NY, USA: ACM, 2009, pp. 499–500. [Online]. Available: http://doi.acm.org/10.1145/1542275.1542349

[4] "The 2nd data prefetching championship (dpc-2)," http://comparch-conf.gatech.edu/dpc2/, 2015.

[5] S. Palacharla and R. E. Kessler, "Evaluating stream buffers as a secondary cache replacement," in *Proceedings of 21 International Symposium on Computer Architecture*, Apr 1994, pp. 24–33.