SelSMaP: A Selective Stride Masking Prefetching Scheme

JIAJUN WANG, REENA PANDA, and LIZY K. JOHN, Department of Electrical and Computer Engineering, University of Texas at Austin, USA

Data prefetching, which intelligently loads data closer to the processor before demands, is a popular cache performance optimization technique to address the increasing processor-memory performance gap. Although prefetching concepts have been proposed for decades, sophisticated system architecture and emerging applications introduce new challenges. Large instruction windows coupled with out-of-order execution makes the program data access sequence distorted from a cache perspective. Furthermore, 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 nonuniform stride patterns by taking the minimum observed address offset (called a reference stride) as a heuristic. A stride masking is generated according to the reference stride and is to filter out history accesses whose pattern can be rephrased as uniform stride accesses. Prefetching decision and prefetch degree are determined based on the masking outcome. As SelSMaP prediction logic does not rely on the chronological order of data accesses or program counter information, it is able to unveil the effect of out-of-order execution and compiler optimization. We evaluated SelSMaP with CloudSuite workloads and SPEC CPU2006 benchmarks. SelSMaP achieves an average CloudSuite performance improvement of 30% over nonprefetching systems. With one to two orders of magnitude less storage and much less functional logic, SelSMaP outperforms the highest-performing prefetcher by 8.6% in CloudSuite workloads.

CCS Concepts: • **Computer systems organization** → **Superscalar architectures**; *Reduced instruction set computing*; *Pipeline computing*;

Additional Key Words and Phrases: Prefetching, cache, memory subsystem

ACM Reference format:

Jiajun Wang, Reena Panda, and Lizy K. John. 2018. SelSMaP: A Selective Stride Masking Prefetching Scheme. *ACM Trans. Archit. Code Optim.* 15, 4, Article 42 (October 2018), 21 pages.
<https://doi.org/10.1145/3274650>

1 INTRODUCTION

As the world is entering the era of big data, a growing number of applications are working with very large datasets that do not fit into the typical processor's top-level (L1/L2) caches. As a result,

This research was also supported in part by NSF grants 1117895. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the NSF. We also wish to acknowledge the computing time we received on the Texas Advanced Computing Center (TACC) systems. We would also like to thank the anonymous reviewers for their helpful suggestions to improve the article.

Authors' addresses: J. Wang, Department of Electrical and Computer Engineering, 2501 Speedway, Austin, TX, 78712; email: jiajunwang@utexas.edu; R. Panda, Department of Electrical and Computer Engineering, 2501 Speedway, Austin, TX, 78712; email: reena.panda@utexas.edu; L. K. John, Department of Electrical and Computer Engineering, 2501 Speedway, Austin, TX, 78712; email: ljohn@ece.utexas.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

2018 Copyright is held by the owner/author(s). Publication rights licensed to ACM.

1544-3566/2018/10-ART42

<https://doi.org/10.1145/3274650>

(a) Global stride pattern Distribution

(b) Local stride pattern Distribution

Fig. 1. Stride pattern in CloudSuite workloads.

the performance efficiency of the last-level cache (LLC) and the off-chip memory has become the crucial determinant of big-data application performance and power (Ferdman et al. 2012; Wang et al. 2014). Hardware prefetching has been used for decades for mitigating the high latency between processor and memory. Hardware prefetchers are ubiquitous and yet complex structures in current computer systems. Based on previous memory access patterns, hardware prefetchers speculate future data accesses and populate processor caches before the data is referenced. Maximizing speculation accuracy and coverage while minimizing introduced cache pollution are the main challenges in prefetcher design. To overcome these challenges, it requires a hardware prefetcher to detect as many data access patterns as possible and dynamically evaluate prediction confidence and adjust prefetch degree.

Emerging applications pose additional challenges to hardware prefetcher designs. Prior research (Wang et al. 2017) has shown that predicting streaming or uniform stride access behavior alone is not sufficient to improve memory subsystem efficiency for emerging applications such as cloud workloads. Figure 1 shows the global and local stride pattern distribution of CloudSuite workloads. The figure illustrates that only less than 30% of global memory reference streams exhibit uniform stride access patterns. Thus, for the Stream (Jouppi 1990) prefetcher, which relies on a chronological cache access sequence of global memory streams, the performance benefit from prefetching is limited as they are restricted to detecting global stride only. The stride prefetcher, which detects localized stride from the same memory instruction, is only able to gain at most 50% prefetching opportunity from exploiting the local memory reference stream. Therefore, prefetching schemes that can detect both uniform and nonuniform stride patterns are required to address those challenges.

Besides, prior research studies (Ferdman et al. 2012; Wang et al. 2017) 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 cache-line evictions, which lead to extra cache misses. Such negative impact is prone to be exaggerated when running big data workloads that have large memory footprints (Panda and Zheng 2017). Compared to traditional desktop applications, a growing number of emerging applications are working with large datasets that do not fit into modern processor cache hierarchies. Cache capacity and memory bandwidth are precious resources, and their performance efficiency has become crucial. For applications whose data accesses exhibit long temporal reuse distance but meanwhile still benefit from caching, cache-line 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. To address these challenges, we propose SelSMaP, a **Selective Stride Masking Prefetching** scheme. SelSMaP is able to detect both regular and nonuniform 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 the cache access order is different from the uniform program access order. For example, a software developer arranges cache-line accesses “A, A+2, A+4, A+6, A+8, A+10, A+12” in program order. After going through an out-of-order execution engine, cache access order may become “A+4, A, A+2, A+6, A+10, A+8, A+12.” The observed cache-line address offset sequence, which is “-4, +2, +4, +4, -2, +4,” becomes nonuniform. 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.

Nonuniform stride or multidelta 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 the irregular delta sequence of “+1, +9, -8, +1, +8, +1, -8, +1.” With the help of the minimum observed delta and the stride mask with a fixed window range, SelSMaP is able to extract two regular (stride of one) streams with different base addresses B and B+10 from the original memory access stream.

Self-trained prefetching degree is adjusted at the granularity of individual prefetch request generation in SelSMaP. Some prefetchers adjust prefetch degree based on the feedback of cache miss rate variation (Ebrahimi et al. 2009; Pugsley et al. 2014), which is an accumulated effect from the 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 but less efficient in other coexistent streams, and an optimal solution should apply high prefetch degree to the former stream and low degree to the latter stream. Adopting the 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 of the current data stream.

We evaluated SelSMaP using both single-threaded SPEC CPU2006 benchmarks and multi-threaded CloudSuite workloads. Results show an IPC improvement of an average of 30% compared to a nonprefetching baseline in the SPEC CPU2006 suite. Comparison of SelSMaP with state-of-the-art prefetchers shows average 10% performance improvements in CloudSuite applications with less hardware. The rest of this article is organized as follows: Section 2 describes SelSMaP

Fig. 2. SelSMaP overview.

architecture. Section 3 presents our design evaluation. Related works are briefly discussed in Section 4. We conclude our work in Section 5.

2 SELSMAP DESIGN

Figure 2 demonstrates a multicore system with the SelSMaP serving as the LLC prefetcher. The 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, the SelSMaP monitors all LLC accesses and maintains access history in the SRT. Apart from holding access history information, the SRT makes stride pattern prediction (i.e., the stride reference) and feeds it into the SML to generate a stride mask. With the stride mask and access history, the DML evaluates the confidence of the predicted pattern. If the DML decides to trigger prefetching based on the stride reference, then the 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, and then we will walk through an example to demonstrate how a prefetch request is generated in the SelSMaP.

2.1 Stride Reference Table

The SRT holds access history information and generates reference stride. It is a set-associative structure, which is indexed by a region tag obtained from the upper bits of the data address. Each SRT entry keeps track of recent memory accesses within an address region, which is a fixed-size memory space. As shown in Figure 3, 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 cache line. The *Access History* field records which cache lines within that region have been accessed over time.

Our experiments indicate that minimum address offset is a good heuristic for making data pattern prediction as compared to using constant data address offsets. Constant data address offset has been a clear indication of a regular stride data access pattern. However, the prefetching scheme with solely a previous and current address offset comparison has become insufficient in modern computer architecture, because the cache access sequence of such a regular pattern can be out-of-ordered. One ideal case is to keep track of all the previous data access address offsets, but is not practical due to limited on-chip metadata storage constraints. We observe that under most circumstances (e.g., circumstances discussed in Section 1), the minimum offset is often the hidden constant stride of an out-of-ordered data access stream. To address the pattern detection problem as well as storage constraint, SelSMaP keeps track of the minimum address offset occurring among

all the 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 cache-line granularity. In order to compute address offset, a *Previous Access* field is required to maintain the address of the previous data access at cache-line granularity as well.

An *Access History* field is used to keep track of the memory access history within a region. Access History is stored in a bit vector, where each bit corresponds to a cache line in the memory region. The bit position corresponds to the region offset of a cache line; 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 cache line.

The size of each field depends on the region size and the cache line size. In this article, our simulation infrastructure is configured with a 4KB memory region, 64B cache line, and 48-bit-wide physical address. Thus, the *Region Tag* is 36 bits 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 cache line. The *Stride Reference* field is set to be 4 bits wide. Although a wider Stride Reference field can hold larger offset and the largest address offset can be 6 bits wide, we figure out that a wider field does not help much, because within a 4KB region a stride with value larger than 16 cache lines can occur at most four times, which is not high enough to build confidence to trigger prefetching.

Every time SelSMaP observes a demand access and finds an SRT region tag match, it updates the SRT based on the steps demonstrated in Figure 4. In step ①, the cache line offset between the current and previous access is calculated. If the calculated offset is smaller than the value in *Stride Reference*, the old value is replaced. In step ②, the *Previous Access* field is updated with the current access. In step ③, the bit corresponding to the current access is set in the *Access History* field. However, if there is no region tag match, SRT assigns a new entry where the *Region Tag* is filled with the upper bits of the access address; the *Stride Reference* is initialized to 15; the *Previous Access* is updated with the current access; and the bit corresponding to the current access is set in the *Access History*.

2.2 Stride Mask Logic

SML generates a stride mask based on the stride reference value from the SRT. SML is implemented as a lookup table, in which every reference stride is coupled with a *stride mask*. The stride mask represents the partial access history pattern if the predicted reference stride exists. The width of the stride mask, which is 64 here, is set according to the width of the *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 the window size. When the reference stride value is 2, every other position of the stride mask is a logic 1, or *10101010101010b*. The reason for setting a window size is to restrict pattern detection to near neighbors and eliminate the negative impacts of various stride access patterns mixing in the same region. Table 1 lists all one-to-one correspondences between a reference stride and its stride mask.

2.3 Decision-Making Logic

DML evaluates the confidence of a reference stride with its stride mask and makes prefetching decisions. As shown in Figure 3, DML consists of two main components: a positive and negative stride matching logic and an arbitration logic.

We use positive and negative stride matching logic to determine the direction of prefetching, i.e., in positive (forward) or negative (backward) directions. Two stride matching logics have the same components: a shifter, a series of logic AND gates, and a count-ones logic. Access history is

Table 1. Stride Mask Logic Lookup Table

4-Bit Reference Stride	64-Bit Stride Mask
1	0x0000 0000 0000 00FF
2	0x0000 0000 0000 AAAA
3	0x0000 0000 0092 4924
4	0x0000 0000 8888 8888
5	0x0000 0084 2108 4210
6	0x0000 8208 2082 0820
7	0x0000 8102 4081 0240
8	0x8080 8080 8080 8080
9	0x4020 1008 0402 0100
10	0x0801 0080 2008 0100
11	0x0040 0801 0020 0400
12	0x0800 8008 0080 0800
13	0x0008 0040 0200 1000
14	0x0080 0200 0800 2000
15	0x0800 1000 2000 4000

Fig. 3. Stride reference table and decision-making logic.

sent from an SRT entry to both positive and negative stride matching logic, where it gets shifted to the right and left separately. Shifted values perform logic AND operations with the stride mask generated by the SML, and the results are fed into the corresponding count-ones logic. As the name suggests, the count-ones logic counts the number of bits set to one. The outputs of two

Fig. 4. Example on how to update SRT entry.

stride matching logics are fed into the arbitration logic, where the decision on whether to trigger prefetch is made, and prefetch direction and prefetch degree are determined.

Making a prefetch decision includes four 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. Shifting in the right direction isolates accesses in the positive direction, and shifting in the opposite direction isolates accesses in the negative direction.

Afterward, the reference pattern logic retrieves the corresponding *Stride Reference* value from the SRT entry and outputs the corresponding RPV. Since both positive and negative stride patterns are to be detected, 64 bits of RPV are wired reversely in the positive and negative stride matching logic; i.e., the LSB of RPV is paired with the LSB of the shift register in the positive stride matching logic, while in negative stride matching logic the LSB of RPV is paired with the MSB of the shift register.

In the second step, the shift register value and the 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 directions, stride mask bits are wired in a reverse order in the positive and negative stride matching logic; i.e., the least significant bit (LSB) of the stride mask is paired with the LSB of the shift register value in the positive stride matching logic, while in negative stride matching logic the LSB of the stride mask is paired with the most significant bit of the shift register value.

In the third step, results from the 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 an *N* count.

In the last step, the arbitration logic determines whether the speculative pattern guided by the 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 the 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 the values of *P* and *N*. If more positive stride matches are detected in history, *Direction* will be set to 0, and vice versa. Under equal circumstances, we decide direction as positive due to its popularity. Prefetch degree is determined based on the sum of *P* and *N*. 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 cache-line access is the first access of that region in a short period.

Fig. 5. SelSMaP operation.

2.4 Prefetch Address Computation

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 the current access address with stride reference value based on the *Direction* signal. According to the *Degree* signal, PAC may generate more than one prefetch request.

2.5 A Walk-Through Example

We have so far introduced every functional unit of SelSMaP. To put the pieces together, we illustrate the data movement inside SelSMaP using a simple example in Figure 5. For the sake of brevity and clarity, the system is configured to have a cache block size of 16B and cache index bits of 4. SelSMaP is configured to keep track of 16 cache blocks per SRT entry. The first entry shows field status corresponding to a previous memory access to address 0xA040, and the second entry demonstrates field updates when SelSMaP detects a current access to address 0xA060, which is two cache blocks away from the previous access. The bit representing current access in the *Access History* field is shifted out in two opposite directions. The shifted values perform AND operations with stride masks, which are wired in reverse order. The P and N values tell that in the positive and negative direction, there are two and one history accesses matching the pattern of stride of two separately. Based on this information, SelSMaP predicts future access pattern to be positive

Table 2. System Configuration

Processor	4-way out-of-order, Tournament Branch Prediction, 32 load store queue buffer
L1 cache	Private, 64KB, 4-way associative, 64B cache line, LRU
L2 cache	Shared, 2MB, 8-way associative, 64B cache line, LRU
Main memory	DDR3_1600K, 4 channels, 1 rank/channel

stride of two. Since the sum of P and N is smaller than a preset threshold value (say, four), DML asks PAC to generate only one prefetching address with a positive stride of two. Hence, a prefetch request to address 0xA080 is generated by SelSMaP.

3 EVALUATION

3.1 Simulation Methodology and Workloads

Except for the SPEC CPU2006 suite, we evaluate six applications from the CloudSuite (Ferdman et al. 2012): Data Serving, MapReduce, SAT Solver, Web Frontend, Web Search, and Media Streaming. Data Serving focuses on NoSQL data stores. We choose the 15GB Yahoo! Cloud Service Benchmark (Cooper et al. 2010) dataset to evaluate the performance of the Cassandra database. MapReduce is a computational model that is able to handle large-scale analysis, cluster and filter a large amount of data processes, and spread computation among a group of machines. It benchmarks a four-node Hadoop cluster running the Bayesian classification algorithm. The SAT Solver application targets software verification, where computation is partitioned into smaller subproblems and distributed to the cloud, where a large number of SAT Solver processes are hosted. The Web Frontend application hosts web services in cloud. It includes a load balancer to distribute independent client requests, a web server to serve the client requests, and middleware to store the state in back-end databases. The Web Search application gets user request information through indexing, which is a process associate terabyte of data found from online resources to their domain names and HTML-based fields. It benchmarks an index serving node with an index size of 2GB and a data segment size of 23GB of content obtained from the public Internet. Media Streaming services such as YouTube and Netflix take advantage of large computing clusters to process and transmit media files of diverse formats in high speed. It benchmarks serving videos of varying duration to simulated clients. The representative phases of these workloads are captured. The length of representative phases is 4 billion instructions (250M per thread) for CloudSuite and 250M instructions for SPEC. Note that CloudSuite applications are run with 16 cores, whereas SPEC benchmarks are run with a single core.

We compare SelSMaP with three other prefetching schemes: stream buffer, AMPM, and BO. AMPM and BO prefetcher are implemented based on publicly available implementations from the first and second Data Prefetching Championship (DPC1 2009; DPC2 2015). The stream buffer prefetching scheme adapts the idea of multientry stream buffer discussed in Palacharla's work (Palacharla and Kessler 1994). All SPEC CPU2006 simulations are carried out on a GEM5 simulator, and CloudSuite simulations are carried out on an in-house cycle-accurate cache simulator interfaced with a detailed Ramulator memory model. Detailed system configuration is listed in Table 2.

3.2 Cost Comparison

SelSMaP is configured to consume a total storage of 476B, which is solely the storage cost of SRT. SRT consists of 32 entries, with each table entry 119 bits wide, which involves 40 bits for Page

Table 3. Prefetcher Storage Overhead

Prefetcher	Storage Cost	Configuration
Streambuf	512B	4KB region, 64-stream detector
AMPM	4KB	16KB region, 1x52 Access Maps,
BO	790B	46 scores, 2x64 RRT,
SelSMaP	476B	4KB region, 1x32 SRT

Tag, 4 bits for Reference Stride, 6 bits for Previous Access, 64 bits for Access History, and 5 bits for LRU states. Additional logics like adders and comparators are required to perform a Reference Stride field update. The reference pattern logic is a 4-input 64-output lookup table, which can be implemented with multiplexers. To match the reference pattern with shifted access history, one pair of 64-bit shifters and 64 AND gates is required. The count-ones logic could be easily implemented as any population count logic. The arbitration logic consists of one adder to add P and N value and two comparators (one for P and N comparison and another for sum and threshold comparison).

The total storage cost of AMPM is around 3,998B, which is eight times more than the one of SelSMaP. It takes AMPM 3,786B to store a memory access map table with 52 entries. Each table entry consists of a 40-bit address tag, 6-bit LRU status, 4-bit access counter, 18-bit interval timer, and 256-state $\times$ 2-bit access history. In addition to table entries, the access map involves a 3-bit mode register and four 32-bit performance counters. Except for the memory access map table, the adaptive stream filter and stream length histogram cost 672 and 1,024 bits, respectively. The AMPM pattern detection costs much more logic than SelSMaP as well. The pattern matching logic of AMPM requires two 256-bit integer shifters, 2×256 OR and AND gates for pattern matching, one 256-bit priority encoder, and offset adders that are composed of small adder and increment logic.

SelSMaP consumes 40% less storage than BO does. BO is configured to maintain 64×2 accesses in the recent request table and evaluate 46 offsets with a 5-bit score successively in one round, plus holding 15 slots in a delay queue. The total budget size becomes 2×64 entries $\times$ 40-bit tag + 46 scores $\times$ 5-bit + 15 delay queue slots $\times$ 59-bit tag and timer + 82 miscellaneous bits = 6,317 bits (around 790B). Since BO evaluates offsets by searching target address tags in the RR table, it needs logic that can update counters, select which of the 46 candidate offsets to evaluate, and compare offset scores. The combinational logic cost of BO is comparable with SelSMaP's cost, and both are much less than what AMPM costs.

3.3 Performance Evaluation Based on Single-Thread Workloads

IPC. Figure 6 shows the performance of four prefetchers normalized to the baseline of a non-prefetching system. A workload is labeled as “prefetch friendly” if prefetching brings over 10% performance improvement over baseline; otherwise, it is labeled as “prefetch agnostic.” Compared against the baseline, SelSMaP attains a performance speedup of 1.76X among prefetch-friendly benchmarks and 1.28X among all benchmarks. SelSMaP beats the stream buffer across all workloads and performs 35% better than BO among prefetch-friendly workloads and on average 13% better 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, SelSMaP beats AMPM by 2% on average CPU2006 suite performance. The adaptive prefetch degree in SelSMaP contributes to higher SelSMaP performance compared with stream buffer. For

Fig. 6. Single-core performance normalized to the nonprefetching system.

Fig. 7. Prefetch breakdown. Single core.

example, streaming access behavior is observed in libquantum. SelSMaP increases prefetch degree after confirming the stream pattern and hence has higher performance than stream buffer, whose prefetch degree is fixed. The other characteristic that leads to better performance is SelSMaP's ability to detect nonuniform stride or multidelta pattern. The lbm benchmark has multidelta sequences that eventually touch all cache blocks in a memory region. Stream buffer is limited to detect uniform stride pattern and not able to generate any prefetch requests, whereas SelSMaP treats it like streaming accesses coming out of order and issues prefetch requests. SelSMaP performs better than BO in almost all workloads except for cactusADM (14% less than BO). Data access behavior in cactusADM is characterized as having a long memory region reuse distance and huge number of hot regions. Such characteristics result in SelSMaP table entry thrashing and hence worse performance. BO avoids such thrashing because its detection scheme depends on the spatial locality of all accesses within the past period rather than certain memory regions.

Prefetch Timeliness Breakdown and Accuracy. In Figure 7, we show a prefetch request breakdown. Prefetch requests are classified into four types: Useful, Late, Early, and Wrong. Useful prefetched cache lines are the ones that have been accessed at least once before being evicted. A late prefetch happens when the cache line to be prefetched is not valid in cache but is outstanding due to a CPU demand request miss. If a prefetched cache line is evicted without any usage but a demand request to it occurs within 1,000 cycles after eviction, that prefetch request is identified as being early. Wrong prefetch is the one that does not belong to any of the above three types. In general, SelSMaP has higher accuracy than other prefetchers. SelSMaP tends to issue more useful prefetches because its prefetch degree is related to the prefetch confidence, i.e., total number of stride matches in the positive and negative direction, whereas AMPM prefetch degree is related to bandwidth usage and accumulated prefetch accuracy rather than the confidence of its pattern prediction. Besides, SelSMaP only detects one stride value (reference stride value) per prefetch generation stage. If that reference stride value has not been observed in history, no prefetch request would be made. However, all possible stride values are evaluated in AMPM. The

Fig. 8. Normalized DRAM bus transactions. Single core.

Fig. 9. Multicore performance normalized to the nonprefetching system.

exhaustive test increases the possibility for AMPM to find a stride and meanwhile decreases its accuracy because AMPM confirms a stride aggressively. Although BO makes stride prediction based on the score of each candidate stride, its unused prefetch ratio is also higher than SelSMap. This is because BO builds its confidence based on all the past accesses, which have a higher chance to involve multiple streams, whereas SelSMap sets its confidence according to past accesses within the range of memory region, which has a higher chance to contain just one data stream.

DRAM Bandwidth Usage. Figure 8 illustrates the number of memory accesses with respect to the baseline. Ideally, a perfect prefetcher does not lead to memory access increase, because every memory access initiated by the prefetcher should ideally be initiated by a demand request at a future time in the nonprefetching case. However, both prefetch requests to useless addresses and prefetching-induced cache pollution result in additional memory accesses. We use the normalized memory transaction count to evaluate the additional memory system overhead caused by useless prefetching. For the lbm, leslie3d, libquantum, and bwaves benchmarks, the number of bus transactions remains the same when the prefetcher is enabled, while memory bandwidth usage increases a lot in all three prefetcher cases. No additional bus transaction implies that prefetchers do not bring pollution to cache. In all benchmarks, SelSMap does not place much burden on memory compared with the other two competitors. SelSMap either keeps a similar number of bus transactions or brings much fewer unnecessary transactions than AMPM and BO do.

3.4 Performance Evaluation Based on Multithreaded Workloads

IPC. Figure 9 shows the performance impact of various prefetching schemes on multithreaded cloud applications in a multicore system. SelSMap outperforms the other three prefetching schemes among half of CloudSuite workloads. Web Frontend and Web Serving are prefetch friendly; i.e., they gain a performance benefit from most of the prefetching schemes. Among all

Fig. 10. Prefetch accuracy. Multicore.

the evaluated prefetching schemes, SelSMaP shows significant performance improvement in these two workloads. SelSMaP achieves the highest speedup of 2.2X in the Web Search application, beating the second-best prefetcher, AMPM, by 40%. It is also worth noticing that out of four evaluated prefetching schemes, SelSMaP is the only one to show performance improvement (20%) on the Data Serving workload. MapReduce, SAT Solver, and Media Streaming workloads are prefetch agnostic.

Prefetch Accuracy. The key to SelSMaP’s high performance is that it generates an adequate number of prefetch requests and meanwhile achieves high prefetching accuracy. Figure 10 illustrates that SelSMaP prefetch accuracy reaches as high as 80% on the Web Search application. Although the prefetching accuracy of SelSMaP is not always the highest, it does not indicate that SelSMaP makes more wrong address predictions. Actually, SelSMaP makes a fairly accurate address prediction, although some prefetch requests are not generated in a timely fashion such that the prefetch request is still outstanding when the demand request to the same prefetched cache block arrives, and such prefetch requests are not considered useful in our work. In order to achieve high accuracy and put low burden on memory bandwidth, SelSMaP adjusts the prefetch degree based on the local information, i.e., access history of the page to which the prefetched line belongs, whereas other prefetchers tune degree based on global information like bandwidth usage and accumulated prefetch accuracy. When detecting stride pattern and testing its confidence, unlike AMPM or BO, which aggressively test multiple candidate strides to find the optimal stride, SelSMaP evaluates only one reference stride. Although AMPM generates a large number of useful requests, it also introduces a nonnegligible amount of useless prefetch requests whose negative cache performance impact may eliminate the benefit of useful prefetch requests.

DRAM Bandwidth Usage. Figure 11 illustrates normalized DRAM bus transactions of various prefetching schemes on cloud applications. It is observed that in CloudSuite applications, SelSMaP is on average less aggressive compared with AMPM (10% less accesses), and meanwhile SelSMaP achieves better performance than AMPM. Considering the relatively high prefetch accuracy and low additional memory traffic, SelSMaP is able to issue an adequate number of useful prefetch requests and meanwhile limits the number of useless requests. 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 8MB LLC. However, according to workload analysis in prior work (Wang et al. 2017), the CloudSuite data access pattern shows long reuse distance, and dominant working sets hardly fit into LLC. SelSMaP generates fewer useless prefetch requests than prefetchers with similar accuracy but is more aggressive (e.g., AMPM). Useless prefetch requests consume cache space, indirectly reducing the effective LLC capacity. Thus,

Fig. 11. Normalized DRAM bus transactions. Multicore.

Fig. 12. Performance contribution of several SelSMaP components.

useless prefetching does not severely impact SPEC performance, whereas CloudSuite benchmarks are less tolerant of cache pollution than SPEC.

3.5 Sensitivity Study

In this subsection, we study the performance contribution of several SelSMap components. Moreover, we conduct sensitivity studies on both system configuration and SelSMap structure. We specifically pick cache associativity, MSHR size, SRT size, and prefetching trigger threshold of SelSMap. CloudSuite workloads are used as an example.

Performance benefit of several SelSMap components. In order to understand the performance benefit of various SelSMap components, we prepare three versions of SelSMap by adding different SelSMap components one at a time, and illustrate the corresponding performance change in Figure 12. The bottom, middle, and top stacked bars represent our first, second, and third versions. The highlight of the first version is to use the minimum observed offset to detect access patterns. SelSMap’s decision-making logic is not included in this version. Instead, taking an address A as the current access address and k as the minimum observed offset, if both addresses $A - 2k$ and $A - k$ are found in the history, a prefetch request to $A + k$ is generated. There is no dynamic prefetch degree adjustment either, with the prefetching degree fixed at one instead. Based on the first version, we add the SelSMap decision-making logic to get the second version, whose performance changes are illustrated in the middle stacked bar. Based on the second version, we add dynamic prefetch degree adjustment and reach the final version of the SelSMap discussed in this work. From Figure 12, we can see that every SelSMap component makes nonnegligible performance contribute. Taking the Web Frontend as an example, using the minimum observed offset reveals the access pattern in the out-of-ordered access sequence and contributes 28% of its total performance improvement. SelSMap’s decision-making process confirms the confidence of

Fig. 13. Effect of varying MSHR entry number.

a stride by looking at a wider range of address space than only focusing on two addresses, and adding such logic contributes 34% of the total performance improvement. Dynamically adjusting prefetch degree guarantees a sufficient number of prefetch requests and contributes 37% of the total performance improvement.

MSHR size. MSHR is shared between demand and prefetch requests. If there is no available MSHR entry, a prefetch request is dumped. We study how various hardware prefetching schemes are sensitive to MSHR size and illustrate results in Figure 13. We test four MSHR sizes, increasing from eight entries to 32 entries. The number of prefetcher X on workload Y is normalized to the performance of prefetcher X on workload Y with eight MSHR entries. As shown in Figure 13, SelSMaP performance varies within 2%, indicating that SelSMaP is insensitive to MSHR size. For workloads with large memory level parallelism (MLP), MSHR usually keeps a large occupancy, especially when it is shared between demand and prefetch requests. Reducing MSHR size would make MSHR frequently full and leads to a performance degradation due to shared resource contention. However, since CloudSuite applications have small memory-level parallelism and SelSMaP does not send out prefetch requests aggressively, limiting MSHR size has little impact on performance. AMPM is greatly impacted by MSHR size. Specifically, its performance on Web Search jumps by 10% when the MSHR entry number doubles from eight to 16 because AMPM is able to issue more prefetch requests as MSHR gets larger.

Cache associativity. Large associativity helps leverage cache pollution caused by useless prefetching as it gives the cache replacement policy more options to evict unused prefetch block. We perform a study on how sensitive the various hardware prefetching schemes are to the cache associativity and illustrate results in Figure 14. We configure an 8MB LLC with four-way, eight-way, and 16-way, respectively, and the numbers in the figure are normalized to the four-way scenario. As associativity increases, all the evaluated schemes see a performance improvement in the SAT Solver. We can observe a performance jump on the SAT Solver for all the evaluated schemes when associativity increases from four to eight. Beyond the eight-way point, performance improvement has diminishing returns. SelSMaP is more cache associativity sensitive than the other three schemes.

SRT size. We evaluate the performance of SelSMaP with four different numbers of SRT entries: 32, 64, 96, and 128. SRT size will make a difference to performance when a program simultaneously works on a large number of hot pages, or the duration between accesses to the same memory region is so long that the corresponding entry has been evicted from the table by the time of reaccess. Especially in a multicore and multithreaded context, multiple threads usually work on their individual data region to maximize parallelism. Therefore, LLC is more likely to receive data requests to different pages. Figure 15(a) illustrates IPC speedup as SRT entry number increases

Fig. 14. Effect of varying cache associativity.

Fig. 15. Sensitivity analysis.

from 32 up to 128. Illustrated data is normalized to the 32 SRT entries scenario. Still, SAT Solver, Web Search, and Media Streaming gain benefit from larger SRT and get 3% to 5% performance enhancement when entry number increases from 32 to 128. However, considering the tremendous storage cost increment from larger SRT, it is not worthwhile to configure SelSMaP with 128 SRT entries for slightly better performance.

Prefetching trigger threshold. To determine whether to trigger prefetching, SelSMaP compares a preset threshold with the total number of stride matches in the positive and negative direction (i.e., the sum of P and N). We test four threshold values (2, 4, 6, 8) and demonstrate the induced performance change in Figure 15(b). Note that when increasing the trigger threshold value, another gauge that controls prefetch degree is increased proportionally. It is observed that a higher prefetching trigger threshold results in performance degradation in four out of six CloudSuite applications. There is a significant performance drop at the threshold of eight, because few prefetch requests meet this high standard. Generating an adequate number of prefetch requests is a prerequisite to better performance. Although a conservative trigger threshold limits the number of useless prefetch requests, a design tradeoff between prefetching accuracy and aggressiveness always has to be considered.

3.6 Comparison with VLDP

Variable Length Delta Prefetcher (VLDP (Shevgoor et al. 2015)) is one of the state-of-the-art prefetcher designs. Different from prefetchers that predict regular streams with uniform strides, VLDP distinguishes itself by its ability to predict complex multidelta access patterns. Figure 16 compares the performance of VLDP and SelSMaP from the perspectives of speedup, number of

Fig. 16. Performance and bandwidth comparison between VLDP and SelSMaP, with number normalized to the nonprefetching system.

prefetch requests, prefetch accuracy, and memory bandwidth consumption. It is observed that in CloudSuite applications, SelSMaP is on average less aggressive compared with VLDP (20% less bandwidth consumption); meanwhile, SelSMaP achieves better performance than VLDP (up to 30% better performance). The prefetch accuracy of SelSMaP is consistently higher than that of VLDP. Although compared with VLDP SelSMaP generates a similar or smaller number of useful prefetch requests, SelSMaP does not pollute cache or generate additional memory traffic as much as VLDP does. SelSMaP uses DML to filter out low-confidence prefetch requests and minimizes useless prefetch requests compared to other schemes.

4 RELATED WORK

Hardware prefetchers are ubiquitous and yet complex structures in current computer systems. There have been many proposals to make hardware prefetchers more accurate, but in general they increase the design complexity (Baer and Chen 1991; Dahlgren et al. 1995; Dahlgren and Stenstrom 1996; Ebrahimi et al. 2009; Fu et al. 1992; Hur and Lin 2006; Iacobovici et al. 2004; Ishii et al. 2011; Jiménez et al. 2012; Joseph and Grunwald 1997; Jouppi 1990; Kumar and Wilkerson 1998; Nesbit et al. 2004; Nesbit and Smith 2005; Pugsley et al. 2014; Roth et al. 1998; Somogyi et al. 2006). One common approach to identify a data access pattern is to record and learn from the data access history. Different prefetchers look for different signs when digging into history and hence have their own tradeoffs between design cost and effectiveness.

Most prefetching proposals detect stride patterns within either a range of memory address space or accesses initiated from the same instruction. These prefetchers treat access history as a sequence of addresses that either exhibit spatial locality or have the same instruction pointer. Traditional prefetching schemes like stream buffer (Jouppi 1990) and global history buffer (Nesbit and Smith 2005) make use of chronological access order extensively to explore and predict temporal

Fig. 17. AMPM structure.

Fig. 18. BO structure.

locality. However, out-of-order machines have dominated the high-performance computer world, and memory access order can be easily altered by the out-of-order scheme. Hence, new algorithms to explore and utilize spatial locality are developed. A bit vector is used to store data access information in the Spatial Memory Streaming (SMS) (Somogyi et al. 2006) and AMPM prefetchers. A bit vector covers a range of memory address space and each bit represents an individual cache block. Although storage cost is reduced, the logic for detecting access pattern in bit vector becomes much more complex.

Another type of prefetching scheme aims at observing the most popular address offset within a fixed period, and hence it focuses on the chronological order of cache accesses. Since the length of period is preset, a limited number of buffers would be enough to satisfy storage requirements. An even more space-efficient data structure is the Bloom filter, which is first used in the Sandbox prefetcher (Pugsley et al. 2014). Different from previous types of prefetcher, history is not kept for detecting data access patterns but for testing a list of prefetch offsets; i.e., each offset is tested serially to see whether it could have been useful in the past and the one with the highest confidence is used. This kind of prefetcher uses less storage than the previous one but requires a longer training phase.

In this article, we compare SelSMaP with classic designs like stream buffer, as well as state-of-the-art proposals such as AMPM and Best-Offset.

Access Map Pattern Matching (AMPM). As illustrated in Figure 17, the AMPM prefetching employs two components: memory access map and pattern matching logic. AMPM divides the memory address space into regions of fixed sizes, called zones. The memory access map is an indexed structure that records access history information of individual zones. A bitmap records which cache lines inside the zone have been accessed or prefetched, with each bit corresponding to one cache line. AMPM uses this access history to predict strides. However, if the access map becomes stale, prefetching requests from the corresponding memory zone are useless. As such, this structure should be kept up-to-date. To keep storage requirements low and maintain up-to-date memory access maps, AMPM keeps track of only a certain number of hot zones at any time. If a memory access in a new zone arrives, the oldest zone is evicted to make room for the new zone. The pattern matching logic is a combinatorial logic for stride detection by using the history information in the memory access map and the current access. To achieve high coverage, AMPM simultaneously evaluates all possible strides, which are 128 different k_i values, within the zone accessed. k_i is confirmed as a stride when x , $x - k_i$, $x - 2k_i$, $x - (2k_i + 1)$ have been accessed.

When there are multiple accepted k_i strides, AMPM selects one from the priority encoder to make a prefetch request.

Best Offset (BO). The BO prefetcher generates the prefetch line address by adding a “best offset” to the demand access address. The BO prefetcher keeps a list of candidate offsets, and each offset is associated with a score. The offset with the highest score is chosen as the best offset. The BO prefetcher evaluates an offset value by maintaining a table of Recent Requests (RRs) and searching through the RR table to check if the current demand request address could have been prefetched with the current offset. The BO prefetcher is conceptually like the Sandbox prefetcher. Figure 18 illustrates the overall structure of the BO prefetcher. The list of possible offsets is fixed at design time. Every time a load request of address X is observed, all the candidate offsets in the list are evaluated in serial. Address $X - O'$ is looked up in the RR table. A match in the RR table indicates that line X could have been prefetched with offset O' , and the score of offset O' is incremented. The score of a possible offset suggests how accurate this offset would be in the past. A round completes when all offset values in the list are graded. A learning phase ends either when the number of rounds reaches maximum value or when the score of an offset reaches a threshold. The one with the highest score is determined as the best offset in the end of the learning phase. Scores are reset, and a new learning phase starts. The selected offset value will be used as prefetch offset O during the next phase. The RR table holds base addresses of demand accesses that generate prefetch. When a prefetched line Y is filled into cache, address $Y - O$ is written into the RR table, where O is the prefetch offset determined in the previous learning phase. One of the BO design targets is to yield timely prefetch requests. Hence, a delay queue is implemented to hold address X for a fixed time.

5 CONCLUSION

In this article, we present SelSMaP, a high-performance, low-budget LLC prefetching scheme to make prefetch decisions 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 both single-threaded SPEC CPU2006 benchmarks and multithreaded CloudSuite workloads. For SPEC workloads, SelSMaP achieves an average IPC improvement of 23% over no prefetching. Performance is improved on average by 4% and at best 135% compared to the AMPM prefetcher with 88% storage reduction and much less functional logic, and on average by 6% and at best 91% compared to the BO prefetcher with 40% less storage. For CloudSuite workloads, SelSMaP achieves an average performance improvement of 30% over nonprefetching systems, and outperforms the highest-performing prefetcher by 8.6% in CloudSuite workloads.

ACKNOWLEDGMENT

The authors would like to acknowledge Centaur Technology. This work was collaborated with Centaur Technology as independent research. Any design details expressed herein are those of the authors and do not necessarily involve or reveal any current or future design details of the Centaur Technology product.

REFERENCES

- J.-L. Baer and T.-F. Chen. 1991. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing (Supercomputing'91)*. ACM, New York, 176–186. Retrieved from DOI : <https://doi.org/10.1145/125826.125932>
- B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC'10)*. ACM, New York, 143–154.
- F. Dahlgren, M. Dubois, and P. Stenstrom. 1995. Sequential hardware prefetching in shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems* 6, 7 (Jul. 1995), 733–746. Retrieved from DOI : <https://doi.org/10.1109/71.395402>
- F. Dahlgren and P. Stenstrom. 1996. Evaluation of hardware-based stride and sequential prefetching in shared-memory multiprocessors. *IEEE Transactions on Parallel and Distributed Systems* 7, 4 (Apr. 1996), 385–398. Retrieved from DOI : <https://doi.org/10.1109/71.494633>
- DPC1. 2009. The 1st JILP Data Prefetching Championship (DPC-1). Retrieved from <http://www.jilp.org/dpc/>.
- DPC2. 2015. The 2nd Data Prefetching Championship (DPC-2). Retrieved from <http://comparch-conf.gatech.edu/dpc2/>.
- E. Ebrahimi, O. Mutlu, and Y. N. Patt. 2009. Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems. In *2009 IEEE 15th International Symposium on High Performance Computer Architecture*. 7–17. Retrieved from DOI : <https://doi.org/10.1109/HPCA.2009.4798232>
- M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafae, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi. 2012. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. *SIGPLAN Notices* 47, 4 (Mar. 2012), 37–48. Retrieved from DOI : <https://doi.org/10.1145/2248487.2150982>
- J. W. C. Fu, J. H. Patel, and B. L. Janssens. 1992. Stride directed prefetching in scalar processors. In *Proceedings the 25th Annual International Symposium on Microarchitecture (MICRO 25)*. 102–110. Retrieved from DOI : <https://doi.org/10.1109/MICRO.1992.697004>
- I. Hur and C. Lin. 2006. Memory prefetching using adaptive stream detection. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. 397–408. Retrieved from DOI : <https://doi.org/10.1109/MICRO.2006.32>
- S. Iacobovici, L. Spracklen, S. Kadambi, Y. Chou, and S. G. Abraham. 2004. Effective stream-based and execution-based data prefetching. In *Proceedings of the 18th Annual International Conference on Supercomputing (ICS'04)*. ACM, New York, 1–11. Retrieved from DOI : <https://doi.org/10.1145/1006209.1006211>
- Y. Ishii, M. Inaba, and K. Hiraki. 2011. Access map pattern matching for high performance data cache prefetch. *Journal of Instruction-Level Parallelism* 13 (2011), 1–24. Retrieved from <http://www.jilp.org/vol13/v13paper3.pdf>.
- V. Jiménez, R. Gioiosa, F. J. Cazorla, A. Buyuktosunoglu, P. Bose, and F. P. O'Connell. 2012. Making data prefetch smarter: Adaptive prefetching on POWER7. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT'12)*. ACM, New York, 137–146. Retrieved from DOI : <https://doi.org/10.1145/2370816.2370837>
- D. Joseph and D. Grunwald. 1997. Prefetching using Markov predictors. *SIGARCH Computer Architecture News* 25, 2 (May 1997), 252–263. Retrieved from DOI : <https://doi.org/10.1145/384286.264207>
- N. P. Jouppi. 1990. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*. 364–373. Retrieved from DOI : <https://doi.org/10.1109/ISCA.1990.134547>
- S. Kumar and C. Wilkerson. 1998. Exploiting spatial locality in data caches using spatial footprints. In *Proceedings of the 25th Annual International Symposium on Computer Architecture (Cat. No.98CB36235)*. 357–368. Retrieved from DOI : <https://doi.org/10.1109/ISCA.1998.694794>
- K. J. Nesbit, A. S. Dhodapkar, and J. E. Smith. 2004. AC/DC: An adaptive data cache prefetcher. In *Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques, 2004 (PACT'04)*. 135–145. Retrieved from DOI : <https://doi.org/10.1109/PACT.2004.1342548>
- K. J. Nesbit and J. E. Smith. 2005. Data cache prefetching using a global history buffer. *IEEE Micro* 25, 1 (2005), 90–97. DOI : [10.1109/MM.2005.6](https://doi.org/10.1109/MM.2005.6)
- S. Palacharla and R. E. Kessler. 1994. Evaluating stream buffers as a secondary cache replacement. In *Proceedings of 21 International Symposium on Computer Architecture*. 24–33. Retrieved from DOI : <https://doi.org/10.1109/ISCA.1994.288164>
- R. Panda, L. Zheng, and X. John. 2017. Accurate address streams for LLC and beyond (SLAB): A methodology to enable system exploration. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'17)*. 87–96.
- S. H. Pugsley, Z. Chishti, C. Wilkerson, P. F. Chuang, R. L. Scott, A. Jaleel, S. L. Lu, K. Chow, and R. Balasubramonian. 2014. Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA'14)*. 626–637. Retrieved from DOI : <https://doi.org/10.1109/HPCA.2014.6835971>
- A. Roth, A. Moshovos, and G. S. Sohi. 1998. Dependence based prefetching for linked data structures. *SIGOPS Operating Systems Review* 32, 5 (1998), 115–126. Retrieved from DOI : <https://doi.org/10.1145/384265.291034>

- M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti. 2015. Efficiently prefetching complex address patterns. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. ACM, 141–152. Retrieved from DOI: <https://doi.org/10.1145/2830772.2830793>
- S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos. 2006. Spatial memory streaming. In *33rd International Symposium on Computer Architecture (ISCA'06)*. 252–263. Retrieved from DOI: <https://doi.org/10.1109/ISCA.2006.38>
- J. Wang, R. Panda, and L. John. 2017. Prefetching for cloud workloads: An analysis based on address patterns. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'17)*. 163–172.
- L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu. 2014. BigDataBench: A big data benchmark suite from internet services. In *IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. 488–499. Retrieved from DOI: <https://doi.org/10.1109/HPCA.2014.6835958>

Received March 2018; revised July 2018; accepted August 2018