Prefetching for Cloud Workloads: An Analysis Based on Address Patterns

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

***Abstract*—Cloud computing is gaining popularity due to its ability to provide infrastructure, platform and software services to clients on a global scale. Using cloud services, clients reduce the cost and complexity of buying and managing the underlying hardware and software layers. Popular services like web search, data analytics and data mining typically work with big data sets that do not fit into top level caches. Thus performance efficiency of last-level caches and the off-chip memory becomes a crucial determinant of cloud application performance. In this paper we use CloudSuite as an example and we study how prefetching schemes affect cloud workloads. We conduct detailed analysis on address patterns to explore the correlation between prefetching performance and intrinsic workload characteristics. Our work focuses particularly on the behavior of memory accesses at the last-level cache and beyond. We observe that cloud workloads in general do not have dominant strides. State-of-the-art prefetching schemes are only able to improve performance for some cloud applications such as web search. Our analysis shows that cloud workloads with long temporal reuse patterns often get negatively impacted by prefetching, especially if their working set is larger than the cache size.**

I. Introduction

Cloud workloads typically work with significantly larger data sets as compared to traditional desktop applications and do not fit into the typical processor's top level (L1/L2) caches. As a result, the performance efficiency of last-level caches (LLC) and the off-chip memory becomes a crucial determinant of big-data application performance and power [1], [2]. There have been several studies on characterizing the micro-architectural and memory-system performance (cache miss rates, TLB miss rates, etc.) behavior of big data workloads [1], [2], [3] on modern computer systems. One observation from previous work is that simple prefetching schemes are inefficient for cloud workloads [1]. Although simple prefetching schemes are proven to bring performance improvement to some extent on SPEC-like workloads [4], [5], there have been continuous enhancements in the prefetching field. State-of-the-art prefetching schemes yield very high performance compared with simple schemes, and are capable of capturing sophisticated data access patterns. In this work, we are interested in revealing whether cloud workloads would benefit from the state-of-the-art prefetching schemes. We believe that answering this question is of crucial value to match processor design to the needs of the scale-out workloads. It also provides prefetcher designers with a new design perspective by analyzing address patterns in cloud workloads which are very different from that of traditional workloads. To the best of our knowledge, this is the first work that analyzes inherent address patterns in memory access streams of cloud workloads in detail.

Generally speaking, we use CloudSuite [1] workloads as our target workload in this paper and we answer the following questions by thorough analysis.

1. Do cloud workloads benefit from LLC prefetching schemes? What kind of coverage and accuracy can be achieved for the prefetch requests in cloud workloads?

2. What is the reason that prevents certain prefetching schemes from improving CloudSuite performance?

3. Are prefetching schemes sensitive to cache capacity?

In this paper, we focus particularly on the behavior of memory accesses at the last-level cache and beyond. We perform a detailed analysis of the LLC, LLC prefetcher and off-chip DRAM accesses of scale-out applications. To understand the inherent memory access behavior of scale-out workloads, this work includes a detailed analysis [6] of the temporal and spatial locality of modern scale-out workloads. We use data reuse distance to capture the temporal locality of programs, and characterize the data memory access patterns in terms of strides per memory instruction and memory reference stream. We model an LLC prefetcher with eight different prefetching schemes, covering a wide range of prefetching work ranging from pioneering prefetching work to the latest design proposed in the last two years. We use prefetching accuracy and coverage to analyze prefetching effectiveness, and evaluate performance impact by comparing performance speedup and cache miss rate reduction.

By analyzing the temporal locality of address patterns, we discovered that there is a correlation among temporal data reuse distance of workloads, cache capacity, and system tolerance of useless prefetching. It is observed that workload performance is more likely to be negatively impacted by useless prefetching when the workload has the following two characteristics. One characteristic is that the workload has a large percentage of long temporal reuse distance in address patterns; the other characteristic is that the working set of the workload is larger than cache capacity. We also discovered that some cloud workloads exhibit both good spatial and temporal locality, i.e., they have dominant stride patterns which can be exploited by prefetching and the majority of their data accesses are with short temporal reuse distances. These workloads gain significant benefit from prefetching. Previous work shows that the LLC capacity in modern processors is over-provisioned for cloud workloads and suggests reducing capacity for power and performance [1]. However, we observed that prefetching requires large cache capacity to show its performance benefits, and there is a correlation between larger cache and better prefetching performance.

The rest of the paper is organized as follows. In section II, we provide a brief background of scale-out workloads and an

introduction of the prefetching schemes that are evaluated in this paper. In section III, we introduce our experiment methodology and metrics. In section IV, we describe in detail the key performance analysis results. Finally, we discuss related work in section V and conclude the paper in section VI.

II. Background on scale-out workloads and prefetching schemes

In this section, the workloads and prefetching schemes experimented in this work are described.

A. Scale-out workloads

Data Serving - There has been a significant increase in the number and diversity of NoSQL database solutions since recent years. Compared with SQL, NoSQL database provides a more flexible storage model and stronger scalability to higher data set sizes/cluster sizes. Several NoSQL data storage solutions [7], [8], [9] are used as back-ups for large Web applications such as Google Earth and Facebook Inbox. In this workload, the 15GB Yahoo! Cloud Service Benchmark (YCSB) data set is used to evaluate the performance of the Cassandra 0.7.3 database. The server load is generated using YCSB 0.1.3 client [10], which sends requests with a 95: 5 read to write request ratio in Zipfian distribution.

MapReduce - MapReduce is the computational model that is able to handle large-scale analysis, cluster/filter large amounts of data processes, and spread computation among a group of machines. These machines first perform a map function in which data are filtered, and then conduct a reduce function in which results from different machines are aggregated. This workload benchmarks a node of a four-node Hadoop 0.20.2 cluster. A Bayesian classification algorithm, which attempts to guess the country tag of each article in a 4.5GB set of Wikipedia pages, runs on it. One map task is started on one core with 2GB Java heap assigned.

Media Streaming - Thanks to high-bandwidth internet connectivity, recent years have witnessed an explosion in the accessibility to media streaming services such as YouTube and NetFilx, etc. Such streaming services take advantage of large computing clusters to process and transmit media files in diverse formats in a high speed. In this workload, the Darwin Streaming Server 6.0.3 is used. It serves videos of varying duration (from 1 min (1.6GB) to 10 min (>10 GB)) by using the Faban driver [11] to simulate the clients. The benchmark setup uses a low bit-rate video stream to shift stress away from network I/O.

SAT Solver - Symbolic execution is heavily used in hardware and software verification. Due to the complexity of this algorithm, it becomes tractable when the computation is partitioned into smaller sub-problems and distributed to the cloud where a large number of SAT solver processes are hosted. Since modern data center consists of heterogeneous machines, a worker-queue model with centralized load balancing is usually applied to re-balance tasks across a dynamic pool of unequal computer resources. Large scale computation is adapted to the worker-queue model meanwhile minimizing communication overhead. Klee SAT Solver is an important component of the Cloud9 parallel symbolic execution engine [12]. It is set up as one instance per core. Input traces are generated by Cloud9 by symbolically executing the command-line printf utility from the GNU CoreUtils 6.10 using up to four 5-byte and one 10-byte symbolic command-line arguments.

Web Frontend - Web services should be fault-tolerant, widely-available and be of dynamic scalability. Such requirements necessitate web services to be hosted in the cloud. There are typically three roles within the web service architectures: a load balancer to distribute independent client requests, a web server to serve client requests, and middleware to store the state in the back-end database. We characterize a front end machine serving Olio, a Web 2.0 web-based social event calendar. Nginx 1.0.10 - with a built-in PHP 5.3.5 module and APC 3.1.8 PHP opcode cache - runs on the front-end machine. A backend dataset (12GB on-disk) is generated using the Cloudstone benchmark [13]. The Faban driver [11] is used to simulate clients as usual.

Web Search - Web search engines get information through indexing, which is a process associating terabytes of data found from on-line resources to their domain names and HTML-based fields. An index serving node (ISN) of the distributed version of Nutch 1.2/Lucene 3.0.1 is analyzed with content crawled from the public internet, which has an index size of 2GB and data segment size of 23GB. It mimics real-world setups by making sure that the search index fits in memory, eliminating page faults and minimizing disk activity. Clients are simulated using the Faban driver and are configured to achieve the maximum search request rate while ensuring that 90% of all search queries complete in less than half a second.

B. Hardware Prefetching Schemes

Modern processors use prefetching to speculatively improve cache efficiency and increase memory level parallelism (MLP). Prefetching has been proven to be effective in reducing cache miss rates by predicting block addresses that will be referenced in the future, and bringing these blocks into the cache prior to the processor's demand request [14], thereby hiding the access latency. In this section, we briefly introduce several classic prefetching schemes that have been pioneering work in this field as well as state-of-the-art schemes, which were published within the last decade.

One Block Lookahead (OBL) [15] is one of the earliest prefetcher designs. As its name suggests, once the prefetching trigger access happens, OBL prefetches one cache block sequentially, i.e., if trigger access is cache block "i", then OBL prefetches cache block (i+1). The trigger access can be either a cache block access or a miss. The idea of OBL can be extended to prefetch k blocks ahead, where k is referred as prefetch degree.

Stride-N [4] is a type of common hardware prefetcher in today's microprocessors. Stride-N prefetcher ties the data access stream to instruction addresses and observes patterns. It stores the last access address and the last access stride in a hardware-managed table, and indexes the table with the load instruction address every time it loads new data. When a stride pattern is discovered, the stride prefetcher stages the (last address + N * stride) addresses into cache, where N is the prefetch degree.

Stream was originally proposed by Jouppi et al. [5], and since then there have been multiple schemes extended from

his work. One implementation is to detect spatial locality and stream direction within each page. Different from the Stride-N design, stream prefetcher doesn't require program counter (PC) information to train the detector. When adequate number of streaming accesses are detected within a preset memory range, the stream prefetcher initiates a prefetch request to the next address following the stream direction.

Spatial Memory Streaming (SMS) [16] is one of the most well-known prefetching schemes proposed in the last decade. During program execution, data accesses exhibit multiple repetitive patterns that span across large memory regions. SMS correlates spatial data access patterns with the instructions and/or data which initiate these patterns, and streams predicted blocks into the cache ahead of demand accesses. SMS records the spatial data access pattern within a memory region of small fixed size, which captures the layout of cache blocks accessed near one another. These spatial data access patterns are indexed by combining the program counter (PC) and the block offset within the memory region that surrounds the cache block. When another memory region is touched first time and a matching entry is found through PC+offset indexing, SMS uses the previously recorded spatial data access pattern to predict the remaining accesses within the newly encountered spatial region.

Access Map Pattern Matching (AMPM) prefetching won the first Data Prefetching Contest (DPC1) [17]. AMPM implements a simultaneous pattern detection scheme of multiple strides from observed history. AMPM stores the spatial layout of accessed cache blocks within a memory region in an access map. The access map is an indexed structure that keeps access history information for active memory regions. The pattern matching unit of AMPM is a combinatorial logic for detecting strides using the history information in the memory access map and the current access. To achieve high coverage, AMPM simultaneously looks for all possible strides within the memory region accessed. k_i is confirmed as a stride when x, $(x - k_i)$, $(x - 2k_i)$ or $(x - (2k_i + 1))$ have been accessed.

Best Offset (BO) [18] prefetching scheme is the winner of the second Data Prefetching Contest(DPC2). BO prefetcher predicts uniform stride access patterns by evaluating a list of candidate offsets. Each candidate offset is associated with a score. Score of an offset tells how helpful this offset would have been in the past. 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 a Recent Requests (RR) table, which holds base addresses of demand accesses that generate prefetch. A match in RR table indicates that line *X* could have been prefetched with offset *O'*, and the score of offset *O'* is incremented. The offset with the highest score is chosen as the best offset. The BO generates prefetch line address by adding a "best offset" value to the demand access address.

Variable Length Delta Prefetcher (VLDP) [19] is one of the state-of-the-art prefetcher designs. Different from prefetchers which predict regular streams with uniform strides, VLDP distinguishes itself by its ability to predict complex multi-delta access patterns. For each active physical page, multiple recent delta access sequences happened to the same page is maintained in a Delta History Buffer (DHB). The delta access history information from the DHB is used to look up a Delta Prediction Table (DPT) which results in the delta prediction. With the help of DPT, VLDP can correlate previously occurring delta history with a subsequently occurring delta, and makes history based predictions about future deltas.

Signature Path Prefetching (SPP) [20] is another state-of-the art lookahead prefetching algorithm. SPP captures a memory access pattern within a physical page and compresses previous strides into a 12-bit history signature, i.e., new history signature is compressed via conducting a series of XOR and shift operations on current access and old history signature. The signature is used to index a Pattern Table (PT), which stores potential next-stride patterns that correspond to specific history signature. One signature may correspond to multiple next-stride patterns. To help make prefetch decision from the multiple next-stride patterns, each pattern is associated with a counter, whose value is compared to a prefetching threshold.

III. Evaluation methodology

In this paper, we perform a detailed analysis of the LLC, LLC prefetcher and off-chip DRAM accesses of the scale-out applications. The LLC addresses are captured using the SIMICS [21] full-system simulator. We configure a multicore system with private L1 and L2 caches, and a shared last level cache (LLC). Detailed system configuration is listed in Table I. Our infrastructure collects the program counter of each instruction that triggers the LLC access, as well as the corresponding physical memory access address, access type (e.g. Load/Store/Eviction), and the inter-instruction distance information of the LLC accesses. Six applications from the CloudSuite and three SPEC CPU2006 benchmarks with most LLC activity are chosen to illustrate differences between address patterns in SPEC workloads and Cloudsuite. The representative phases of these workloads are captured and are used to generate LLC access traces with our tracing infrastructure. Then we perform several levels of off-line analysis of the collected traces. Table II summarizes the simulation intervals as well as the number of LLC accesses and LLC misses of each workload. A group of eight prefetching schemes are selected, which are representations of both classic and the state-of-the-art schemes. They are evaluated as LLC prefetchers, as part of a trace based cycle-level cache simulator. The configurations of each prefetching schemes are listed in Table III. We integrate

TABLE I: System Configuration

Processor	16 cores,
L1 cache	64KB, 4-way associative, 64B cacheline, LRU
L2 cache	256KB, 8-way associative, 64B cacheline, LRU,
L3 cache	8MB, 8-way associative, 64B cacheline, LRU, 16 MSHR
Main Memory	DDR3_1600K, 4 channels, 1 rank/channel

TABLE II: Workload characteristics

Benchmarks	Simulation length	LLC accesses	LLC misses
Data Serving	4 billion instructions	36,134,150	14,993,597
MapReduce	4 billion instructions	38,119,693	16,644,104
SAT Solver	4 billion instructions	33,271,637	23,254,707
Web Frontend	4 billion instructions	10,311,403	2,277,452
Web Search	4 billion instructions	22,613,857	3,830,574
Media Streaming	4 billion instructions	65,596,871	7,303,085
mcf	1 billion instructions	79,490,811	47,682,507
bwaves	1 billion instructions	23,176,132	22,677,841
tonto	1 billion instructions	158,098	35,737

TABLE III: Prefetcher Configuration

Scheme	Configuration	Is prefetch degree fixed?	Trigger signal
OBL		Yes, 1	On access
Stream	Detect 64 streams, stream window of 16	Yes, 2	On access
Stride-3	Track 1024 memory instructions	Yes, 3	On access
AMPM	52 memory map entries. 4KB region	No	On access
BO	42 candidate offsets, 64 RR table entries	No	On miss
VLDP	Track 128 pages, 8 deltas per page. Maintain 4 DPT, 64 entries per table.	No	On access
SMS	Maintain 64 entries per AGT, 256x8 entries perf PHT. 512B region. No rotation.	No	On new region
SPP	Maintain 512x2 entries per SP, 4096x4 entries pref PT	No	On access

our LLC model with Ramulator [22], an cycle-accurate main memory model. The Ramulator is used in CPU trace driven mode. The traces feed into the Ramulator contain both LLC demand request misses and prefetch requests, as well as the number of instructions between previous and current memory accesses.

We use the following metrics to characterize memory access patterns and assess prefetching performance in this work.

Reuse distance - Data Reuse distance, also known as Mattson's stack distance [23] is a very powerful metric to capture the temporal locality of programs. It captures the number of other unique references that appear between one address and the next use of the same address. Essentially it captures the number of intervening references between reuse of an address. If references are put into a stack, it indicates the depth at which some reused data can be located in the stack. The percentage of data references that exhibit a specific reuse distance can be computed. The distribution of such a metric for a variety of reuse distances provides an excellent picture of the potential performance of the workload with various cache sizes. By capturing the stack distance distribution, essentially we capture the performance of multiple cache sizes in one simulation using a single very large fully associative cache-like model. By doing so, the stack distance approach not only provides performance of caches with different sizes but indicates the total memory footprint of each workload. For example, if the workload has a combined instruction and data footprint smaller than the stack depth, the amount of valid data in the cache (assuming the cache is invalid at simulation start) represents the total memory footprint of the workload.

Global/Local stride patterns - We characterize the data memory access patterns of the big-data applications in terms of strides per memory instruction (local) and memory reference stream (global). For local strides, we define a stride as the difference between consecutive effective memory addresses localized per memory instruction. We then use this information to estimate the most frequently used stride values per memory instruction and the number of memory references that it was used for. For global stride analysis, we define a stride as the difference between consecutive memory addresses and analyze the stride-based behavior when seen across the entire global stream of memory accesses. This approach of characterizing and portraying the stride access patterns in terms of 64-byte blocks is similar to the approach adopted by Joshi et al. [24] for SPEC CPU2000 benchmarks. Both local and global strides are computed at the granularity of 64-byte cache blocks.

Prefetcher coverage and prefetcher accuracy - These are the two commonly reported metrics for evaluating a prefetcher design. A prefetch request becomes useful when the cache line brought in by the prefetch request is referenced. Prefetch coverage is the number of useful prefetch requests over the number of cache misses in non-prefetching situation. Prefetch coverage represents how many cache misses are covered through prefetching and it is the metric to show the effectiveness of a prefetching scheme. Prefetch accuracy is the number of useful prefetches over the number of total prefetch requests generated. Due to the speculative nature of prefetching, a fraction of prefetch requests are not useful but consume bandwidth and cache space instead. There are two factors that determine whether or not a prefetch request is useful, timeliness and accuracy. If a prefetcher predicts a wrong address, then the requested cacheline will never be referenced. Moreover, even when prefetcher requests the correct data that will be accessed in the future, timeliness of the request still makes a difference, i.e., a prefetched line arriving in cache too early may have been evicted before referenced, or a prefetched line may still be outstanding in the memory hierarchy when a demand request to the same line is issued. Prefetch accuracy is the metric to evaluate the additional overhead on memory system caused by useless prefetching.

IV. Results and Analysis

A. Prefetching impact on performance

1) Overall performance: In this work, we evaluate eight prefetching schemes on six CloudSuite workloads and three benchmarks from SPEC CPU2006 suite. We show performance related metrics in Figure 1 [1] and we illustrate prefetching accuracy and coverage in Figure 2. From Figure 1 we observe that prefetching schemes help to reduce LLC miss rate in five CloudSuite workloads except for MapReduce, but their performance does not follow the same trend with miss rate. The reason is that the cache miss rate metric does not take the useless prefetch overhead into account, which includes additional cycles in DRAM accesses and pipeline stalls. Prefetch requests share resources like MSHR and memory transaction queue with demand requests. Backend pipeline stalls when running out of these shared resources and the pipeline stall problem is exacerbated by useless prefetch requests. For example, the OBL prefetching scheme reduces cache miss rate by 10% in both Data Serving and SAT Solver, but causes performance

[1]Note that SPEC speedup ratio may be different from the one given in the proposal of each prefetching scheme. Because prefetching is applied at LLC level in this work, whereas it is applied at L1/L2 (non-LLC) level in the original proposal.

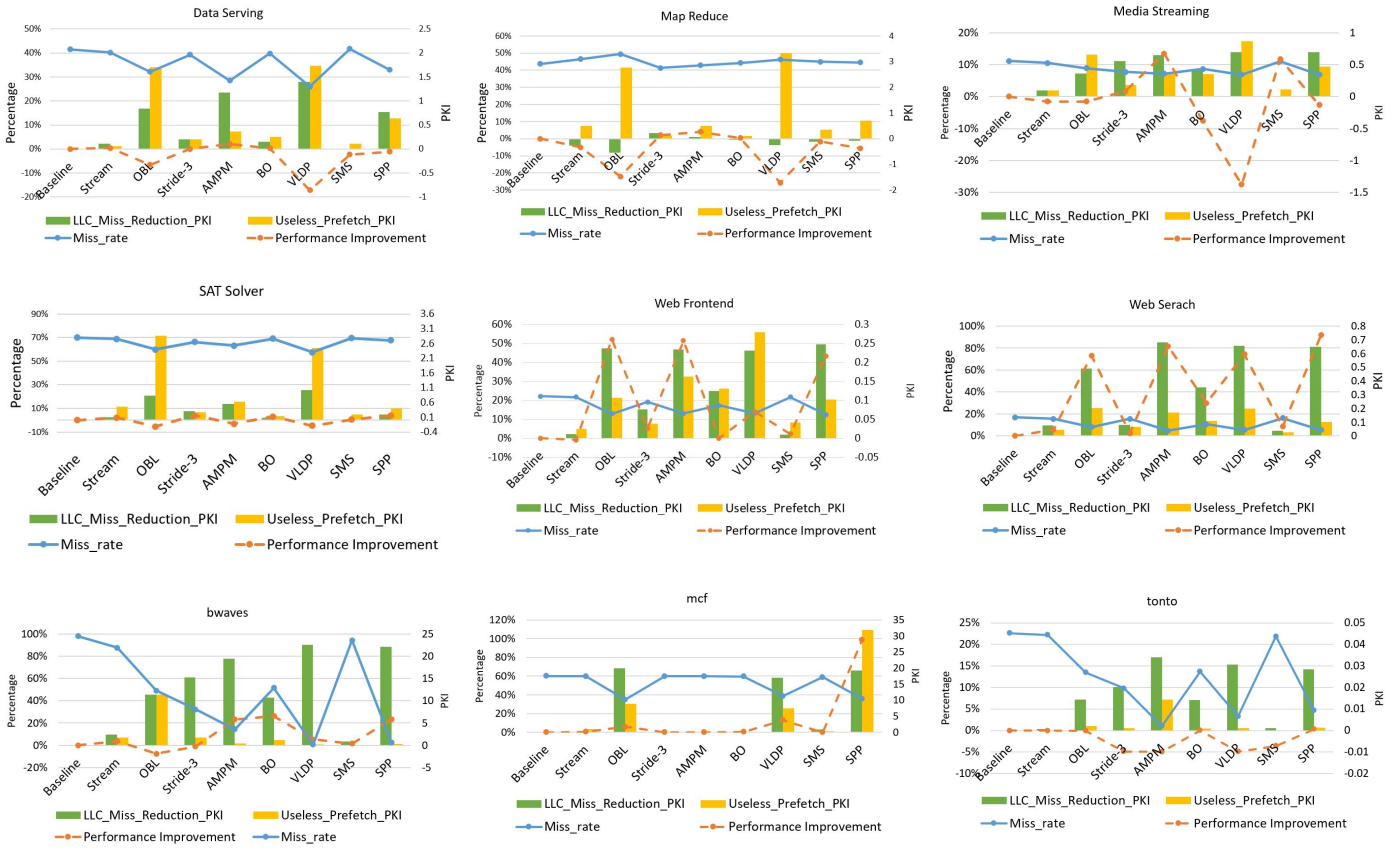


Fig. 1: Performance Impact: Left axis presents the percentage change in performance and miss rate, and right axis presents LLC miss reduction and useless prefetches per kilo instruction

degradation of 7% and 5% respectively. Comparing LLC miss reduction per kilo instruction (PKI) and useless prefetch per kilo instruction, we can tell that the benefit of prefetching is outweighed by useless requests placing extra burden on the main memory.

Among the six CloudSuite workloads, Web Frontend and Web Search substantially benefit from prefetching, especially from the OBL, AMPM, VLDP, and SPP schemes. All prefetching schemes exhibit a minimum accuracy of 60% on Web Frontend, and a minimum accuracy of 70% on Web Search. OBL, AMPM, VLDP and SPP schemes maintain high accuracy while generating adequate number of prefetch requests to eliminate more than 65% of cache misses in the baseline.

An interesting observation is that OBL, the simplest prefetching scheme, is the most cost-effective prefetcher solution for Web Frontend and Web Search applications when considering the ratio of performance to design complexity. OBL achieves the highest performance gain (52%) in Web Frontend, and improves Web Search performance by 73%, which ranks third after AMPM and SPP.

AMPM and VLDP show similar prefetch accuracy (around 60%) on Media Streaming, and both schemes reduce the same amount of LLC misses. However, AMPM increases performance by around 12%, whereas VLDP shows detrimental impact on performance. The reason is that VLDP generates prefetch requests much more aggressively than AMPM does. Though percentages are similar, the absolute number of useless prefetches by VLDP is much higher than the one by AMPM. In fact, VLDP has three times the number of useless prefetches per kilo instruction compared with AMPM. Consequently, the negative impact of the 40% useless prefetches dominates the overall prefetcher performance.

SAT Solver is generally more insensitive to prefetching than other CloudSuite applications, as its performance variation is always within 5% even when OBL and VLDP generate a large number of useless prefetch requests. Data Serving experiences either performance degradation or negligible performance variation with prefetching.

MapReduce is more sensitive to useless prefetches compared with other CloudSuite workloads. As shown in Figure 1, MapReduce suffers from increasing cache misses with prefetching, while other workloads have a higher tolerance. Cache thrashing happens when aggressive prefetching schemes such as OBL and VLDP bring in cachelines which are not to be used in near future and evict out cachelines which will be re-accessed. Therefore, the cache pollution caused by prefetching schemes results in increasing LLC miss rate and thus performance degradation.

2) Address pattern - spatial locality: Spatial locality is an important characteristic of memory access patterns that is exploited heavily by prefetchers. In this section, we present our analysis of the spatial locality in LLC access streams of the CloudSuite workloads. We plot a cumulative distribution of the most frequently used stride values and the percentage of

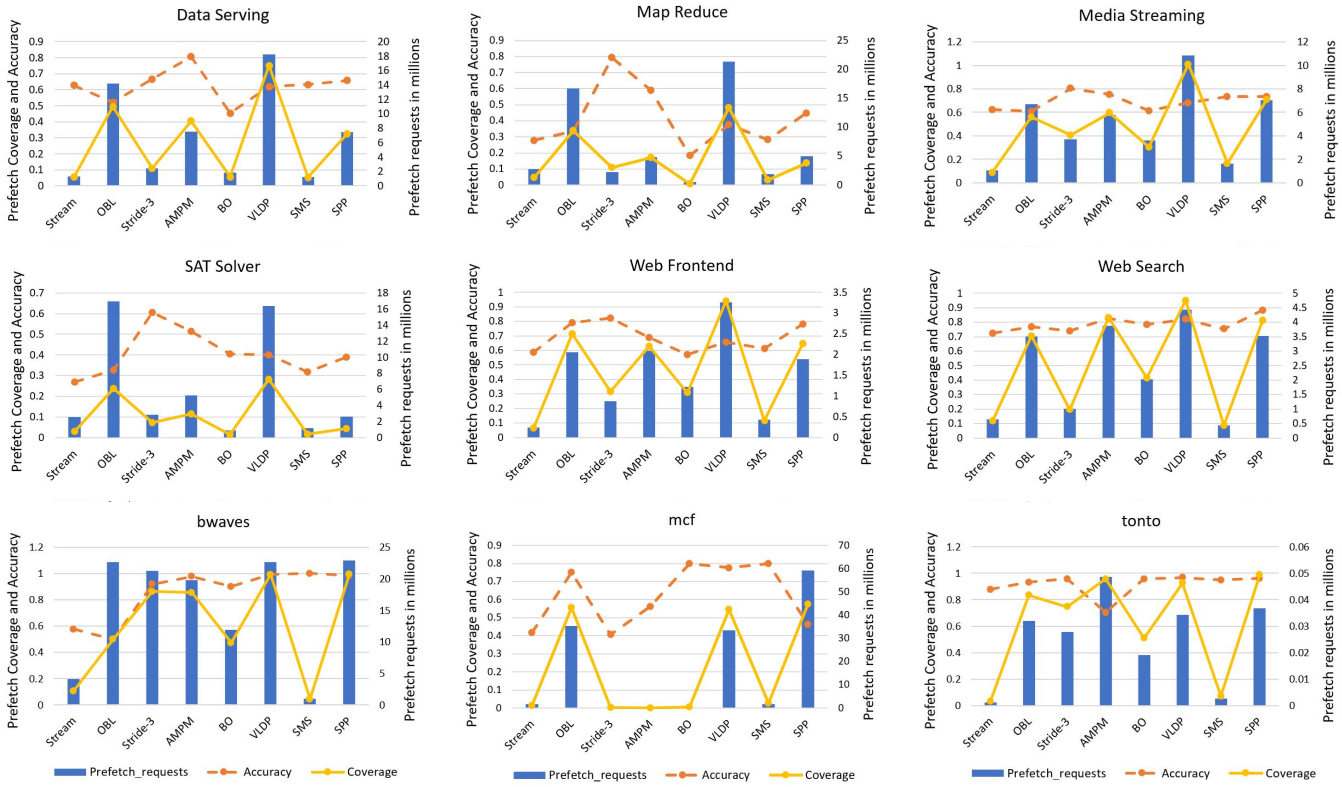


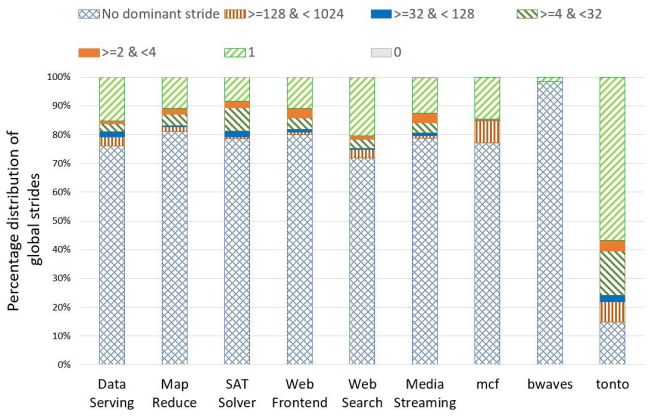
Fig. 2: Coverage and Accuracy Analysis: Left axis presents prefetching accuracy and coverage, and right axis presents number of prefetch requests in millions

total memory references that they make. Figure 3b shows the local stride distribution of the CloudSuite and SPEC workloads at a granularity of a 64-byte block, binned into categories 0, 1, 2, etc and no dominant stride categories. Figure 3a shows the global stride distribution of the CloudSuite and SPEC workloads at a granularity of a 64-byte block, binned into categories 0, 1, 2, etc and no dominant stride categories.

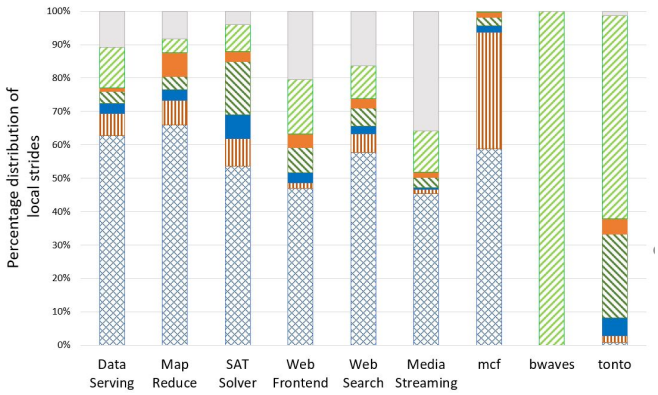
We can observe that most of the big-data workloads do not have good spatial locality at either global or per-memory instruction granularity. In terms of global locality, the most common global stride is 1, but it occurs rather infrequently (less than 15% of the time). Similarly, the local stride characterization shows that the Web Search and Data Serving workloads have a local stride of 1 for approximately 10% of the memory references, while other workloads do not possess any significant dominant local stride patterns. This behavior is expected as most cloud applications either work on data structures that have irregular memory layouts or act on random queries. On the other hand, bwaves and tonto benchmarks (both from the SPEC CPU2006 suite) have very dominant local and global stride patterns. As a result, they are very suitable candidates for prefetching solutions. Prior research has found similar strong stride-based correlations in other SPEC CPU2006 suite workloads as well as TPC-H benchmarks [25], [26], [27].

From Figure 3b, we are able to draw a conclusion that prefetching schemes relying on detecting stride patterns of each instruction will not be effective for CloudSuite workloads. Although Stride-3 has the second best prediction accuracy among all evaluated prefetching schemes, it fails to exploit data locality within each memory instruction and it generates relatively small number of prefetch requests. Therefore Stride-3 has lower coverage and brings negligible performance speedup. Stride-3 is only able to keep track of small local strides (e.g. less than 32), which accounts for less than 15% of local strides in CloudSuite. This small percentage determines the upper bound of performance improvement that Stride-3 prefetcher could achieve in an oracle situation, i.e., assuming data accesses with stride less than 32 are all missed in the baseline and Stride-3 prefetcher makes timely prefetching without polluting the cache. Hence the nature of the workloads exposes the insufficiency of Stride-3.

3) Address pattern - temporal locality: In order to understand why CloudSuite applications have different sensitivity to prefetching, we perform a study of the temporal locality of LLC access patterns in the CloudSuite. Processor caches, including the LLC are designed to exploit the temporal reuse of individual memory elements to minimize the number of accesses to the off-chip DRAM memory. Temporal locality of a program dictates how the miss rate of a processors cache will change as its capacity is varied. Often the miss rate does not decrease linearly as the size of a cache is increased, but stays at a certain level and then makes a sudden jump to a lower level when the capacity becomes large enough to hold the next important data structure. This temporal locality characterization information is represented on the reuse distance graphs for the big-data workloads and SPEC CPU 2006 applications in Figure 4. The x-axis shows the reuse distance (from 0 to 256000, note that each x-axis point refers to a reuse distance value between itself and its previous point) and the y-axis represents



(a) Global stride pattern Distribution



(b) Local stride pattern Distribution

Fig. 3: Spatial locality analysis: Global/Local stride patterns in LLC access streams

the percentage of LLC accesses that have a corresponding reuse distance. Correlating the reuse distance values with an approximate cache size configuration yields that the percentage of references that have a reuse distance of say, less than 512, 8000 and 32000 will fit into a last-level cache of size 32 KB, 512 KB and 8 MB respectively.

Based on the reuse distance distribution, we categorize the nine workloads in Figure 4 into three types. The Web Frontend, Web Search, Media Streaming and tonto benchmarks fall into the first category, where majority of reuse distances are less than 8K. Dominant working set size of applications in the first category fits within a 2-4MB LLC. This implies that these applications have high tolerance to cache pollution from prefetching, when used with an 8MB LLC.

The Data Serving, MapReduce and mcf benchmarks fall into the second category, where more than half of the reuse distances fall between 8K and 256K. Although more than 80% of working set fits within an 8MB LLC, cache pollution can have a detrimental impact on performance. Prefetching causes additional cache block evictions compared with no prefetching. Under non-MRU replacement policy, the chance of a cache block with longer reuse distance to become the victim block is higher than a cache block with shorter reuse distance. That is to say, when LLC capacity is just enough to hold a block with a long reuse distance in the cache until it gets reused, prefetching prevents the cache block reuse by replacing it with a prefetched line. Number of cache misses increases when that prefetch request is useless. From Figure 1 we can observe that prefetching causes more cache misses (negative cache miss reduction) and performance degradation for MapReduce.

The SAT Solver and bwaves benchmarks fall into the third category, where at least half of data accesses have reuse distance larger than 256K. Applications belonging to this category have instruction and data working sets significantly larger which do not fit even in a 16MB LLC. The performance impact of prefetching on SAT Solver and bwaves are completely different. Prefetching schemes can still capture address pattern of bwaves and significantly improve performance. The reuse distance in bwaves is actually infinite, indicating that there are few LLC access reuse in bwaves, and there is little negative impact on cache misses from useless prefetching.

4) Sufficient bandwidth: Figure 5 shows the off-chip memory requests (divided into read, write and eviction traffic) and the off-chip bandwidth consumption of the cloud applications as a fraction of the available per-core off-chip bandwidth. Scale-out workloads experience non-negligible off-chip miss rates, however, the MLP of the applications is low due to the complex data structure dependencies, leading to low aggregate off-chip bandwidth utilization even when all cores have outstanding off-chip memory accesses. Among the scale-out workloads we examined, Media Streaming is the only application that uses up to 15% of the available off-chip bandwidth. Due to the fact that there is little memory level parallelism in cloud workloads, bandwidth is not the major concern when designing prefetching scheme for cloud applications.

B. *Prefetching sensitivity of LLC capacity*

In this section, we study whether larger cache capacity could help prefetching schemes in reducing cache misses and whether prefetching combined with larger cache could improve CloudSuite performance. We select LLC cache size of 8M, 16M, and 32M, and show the normalized performance in Figure 6. The performance we demonstrate is normalized to the baseline with the corresponding LLC capacity, i.e., performance with prefetching in a 32M case is normalized to performance without prefetching in the same 32M case.

On one hand, a prefetcher which makes correct address prediction but sends out prefetch requests too early can benefit from a larger cache. Timeliness is an important factor that influences the performance of a prefetching scheme, but it is difficult to ensure every correct prefetch request arrives in cache in a timely fashion. First of all, it is because data access latency is not fixed. L1 prefetching requests may be serviced by the L2 cache when there is a hit, or may reach main memory in the worst case. Even two DRAM-served prefetch requests may take different number of cycles. For example, one main memory request may have a row buffer hit while the other one with a row buffer conflict needs to wait for extra cycles. Moreover, it requires a lot of front-end information to better predict when the instruction that accesses the prefetched line will be issued, and prefetchers usually do not have such information. Therefore, when a prefetched cache line that is to be accessed hundreds of cycles later fills in the cache, a large cache may hold this prefetched line until

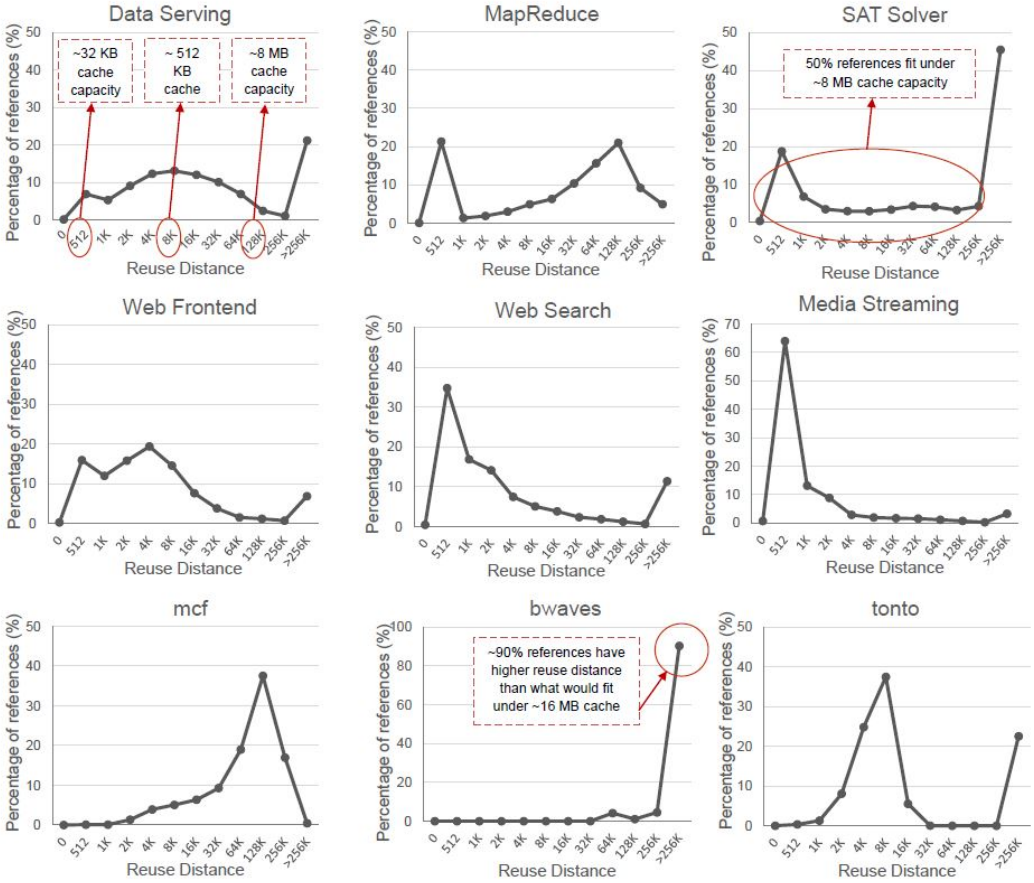


Fig. 4: Temporal locality analysis: The figure shows the approximate reuse distance panel. Y-axis presents percentage of memory references and x-axis presents the reuse distance.

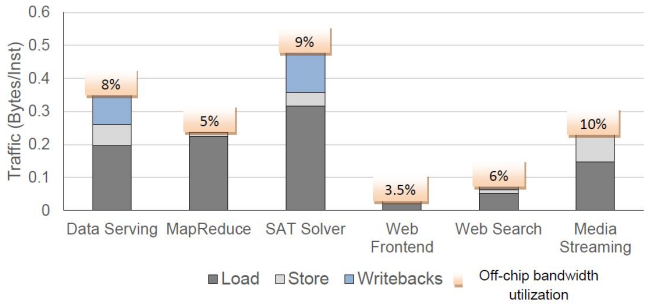


Fig. 5: Off-chip memory traffic

it is accessed. On the contrary, a smaller cache may have kicked the line out earlier due to capacity limit. For example, SMS requires larger cache capacity to show the prefetching benefit on CloudSuite workloads. As capacity increases from 8M to 32M, SMS is able to bring an additional 20%, 29% and 28% performance improvement for Web Frontend, MapReduce and Media Streaming respectively. Performance gain comes from an increase in useful prefetch requests. Since the training and prediction processes are irrelevant to cache capacity, the increase in useful prefetch requests is due to fact that prefetched blocks stay longer in a larger cache as compared to a smaller cache. Therefore, the chance that a prefetched block is accessed before getting evicted goes up. For an inefficient prefetchers, e.g. a prefetchers with lower accuracy, larger cache size helps to avoid performance degradation by mitigating the performance degradation and preventing useless prefetched lines from evicting useful blocks.

On the other hand, workloads with long address reuse distance and working set just fitting in cache (i.e. the second category discussed in temporal locality subsection) are more likely to gain benefit from increasing cache capacity. Taking MapReduce as an example, none of the prefetching schemes is able to improve its performance in the 8M case, while SMS and AMPM improve performance by 26% and 37% in the 32M case respectively. From Figure 4 we could tell that a 16MB LLC can hold 95% of working set size, and around 60% has reuse distance longer than 8K. It indicates that a useless prefetch request has a high probability to cause additional cache misses, because the prefetch request may evict a LRU block which becomes LRU due to the long reuse distance but is to be accessed in the future, and there is a large portion of cache blocks with long reuse distances in Map Reduce. Whereas when cache capacity goes up, the chance that any cache blocks gets evicted decreases, and MapReduce has a higher tolerance for useless prefetch requests.

C. Additional micro-architectural insights

When evaluating prefetching schemes, we observe a scenario when two prefetchers generate the same amount of

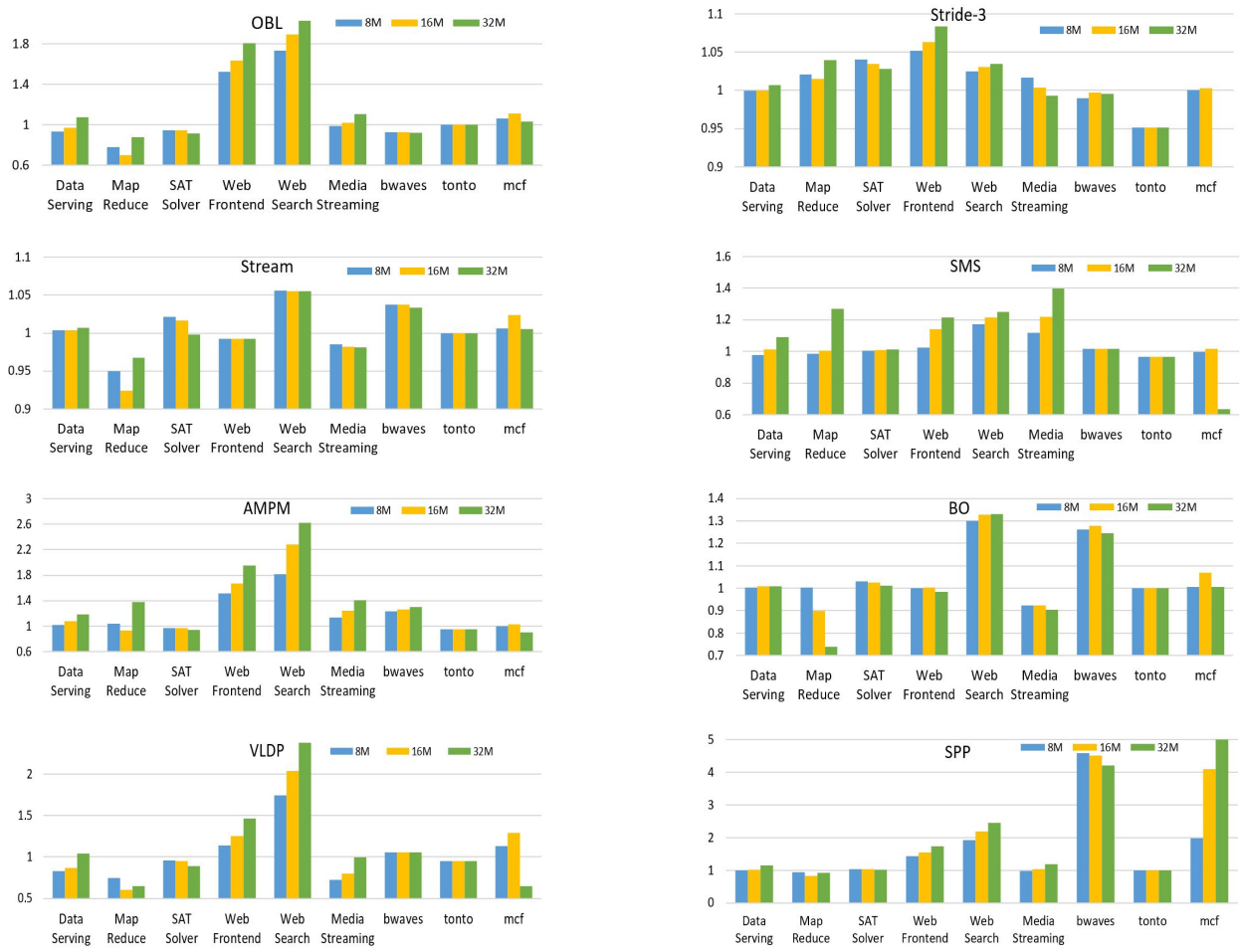


Fig. 6: Prefetching Sensitivity of LLC Size.

prefetch request and achieve similar prefetch accuracy, but gain dramatically different performance improvement. An obvious example is VLDP and SPP when running bwaves. These two schemes have the same accuracy, same coverage, same number of prefetch requests, but show four times difference in performance. The reason is that SPP has a large number of DRAM row buffer hits compared with VLDP. DRAM row buffer locality is an important metric that affects the overall latency of DRAM memory accesses. Any requests that cause row buffer conflicts are very expensive because several additional memory cycles get consumed for precharging, reading out and closing a row per memory access. We examine the effects of two different address mapping schemes on row buffer locality of the scale-out applications (see Figure 7). It can be observed that Addr-Map2 provides more than 21% improvement in row buffer hit rate as compared to Addr-Map1. We suspect that as the scale-out workloads have limited MLP, spreading the accesses over more channels does not yield any better benefits. Rather improving the spatial locality within individual DRAM rows helps the scale-out applications more. This is in contrast with the behavior of SPEC CPU2006 applications which have more inherent MLP and thus benefit more from Addr-Map1.

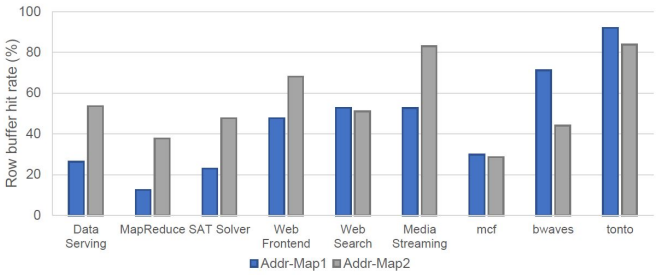


Fig. 7: Row buffer locality [Addr-Map1 - RoRaBaChCo, Addr-Map2 - ChRaBaRoCo]

V. Related Work

Ferdman et al. [1] evaluated six scale-out workloads and identified several causes of microarchitectural performance inefficiencies in big-data workload performance. They observed that the execution-time breakdown of scale-out workloads is dominated by stalls in both application code and operating system. Unlike traditional desktop workloads, scale-out workloads suffer from high instruction-cache miss rates. They also observed that most of the stalls in scale-out workloads arise due to long-latency memory accesses. Zheng et al. [2] inferred that stalls due to kernel instruction execution greatly influence the front end efficiency. However, data analysis workloads have higher IPC than scale-out workloads, and also suffer from notable from end stalls but L2 and L3 caches are effective for them. Deep dive analysis [28] of a

data analysis workload on a modern-day hardware system reveals that big data analysis workload is bounded by memory latency. Lee et al. [29] performed study on the benefits and limitations of hardware and software prefetching using SPEC CPU2006. Their work focused on the comparison between hardware and software prefetching, discussed when one type of prefetching works better than the other, and suggested cooperative hardware/software prefetching. However, none of these micro-architectural studies covers detailed experiments on prefetching for cloud workloads or thorough analysis of memory access behavior and its patterns.

VI. Conclusion

With the emergence and growing relevance of several cloud application domains, analyzing and understanding the inherent patterns in the memory access streams of emerging applications is essential to design efficient memory hierarchies to optimize application performance. While several previous studies have performed micro-architectural level studies and concluded that cloud workloads form a distinct workload class from desktop workloads, there is little work in characterizing memory access patterns and studying how prefetching schemes may affect cloud applications.

In this paper, we evaluated a variety of prefetching schemes as the last-level cache prefetcher and conducted a detailed analysis of temporal and spatial locality behavior of modern scale-out workloads. The data reuse distance/stack distance is used to analyze the working sets and temporal locality. We characterized spatial locality of data memory access patterns in terms of both strides per memory instruction and memory reference stream. We discovered that more than half of cloud workloads gain performance benefit from modern prefetching schemes, due to their high accuracy and coverage. Our analysis showed that there are two factors, temporal reuse distance distribution and working set size, inherent in cloud workloads that determine whether prefetching results in positive or negative performance impact. We also found that in general, larger cache capacity could help prefetching schemes in reducing cache misses because large caches are less detrimentally affected by useless prefetches. Prefetching in conjunction with a large cache could improve cloud workload performance.

VII. Acknowledgements

This work has been supported partially by NSF grant CCF-1337393 and SRC under Task ID 2504. We also wish to acknowledge the computing time we received on the Texas Advanced Computing Center (TACC) systems. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the National Science Foundation or other sponsors. We would also like to thank the anonymous reviewers for their helpful suggestions to improve the paper.

References

[1] 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.

[2] W. Gao, Y. Zhu, Z. Jia, C. Luo, L. Wang, Z. Li, J. Zhan, Y. Qi, Y. He, S. Gong, X. Li, S. Zhang, and B. Qiu, "Bigdatabench: a big data benchmark suite from web search engines," *CoRR*, vol. abs/1307.0320, 2013.

[3] R. Panda, C. Erb, M. LeBeane, J. H. Ryoo, and L. K. John, "Performance characterization of modern databases on out-of-order cpus," in *2015 27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Oct 2015, pp. 114–121.

[4] T.-F. Chen and J.-L. Baer, "Effective hardware-based data prefetching for high-performance processors," *IEEE Transactions on Computers*, vol. 44, no. 5, pp. 609–623, May 1995.

[5] N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *[1990] Proceedings. The 17th Annual International Symposium on Computer Architecture*, May 1990, pp. 364–373.

[6] R. A. Uhlig and T. N. Mudge, "Trace-driven memory simulation: A survey," *ACM Comput. Surv.*, vol. 29, no. 2, pp. 128–170, Jun. 1997. [Online]. Available: http://doi.acm.org/10.1145/254180.254184

[7] "Cassandra," wiki.apache.org/cassandra/FrontPage.

[8] "MongoDB," mongodb.org.

[9] "VoltDB," http://voltdb.com.

[10] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10. New York, NY, USA: ACM, 2010, pp. 143–154.

[11] "Faban Harness and Benchmark Framework." http://java.net/projects/faban/.

[12] L. Ciortea, C. Zamfir, S. Bucur, V. Chipounov, and G. Candea, "Cloud9: A software testing service," *SIGOPS Oper. Syst. Rev.*, vol. 43, no. 4, pp. 5–10, Jan. 2010. [Online]. Available: http://doi.acm.org/10.1145/1713254.1713257

[13] W. Sobel, S. Subramanyam, A. Sucharitakul, J. Nguyen, H. Wong, A. Klepchukov, S. Patil, O. Fox, and D. Patterson, "Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0," 2008.

[14] D. Kadjo, J. Kim, P. Sharma, R. Panda, P. Gratz, and D. Jimenez, "B-fetch: Branch prediction directed prefetching for chip-multiprocessors," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec 2014, pp. 623–634.

[15] A. J. Smith, "Cache memories," *ACM Computing Surveys (CSUR)*, vol. 14, no. 3, pp. 473–530, 1982.

[16] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Spatial memory streaming," in *33rd International Symposium on Computer Architecture (ISCA'06)*, 2006, pp. 252–263.

[17] 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

[18] P. Michaud, "A best-offset prefetcher," http://comparch-conf.gatech.edu/dpc2/, 2015, [The 2nd Data Prefetching Championship (DPC2)].

[19] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti, "Efficiently prefetching complex address patterns," in *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Dec 2015, pp. 141–152.

[20] A. L. N. R. Jinchun Kim, Paul V. Gratz, "Lookahead prefetching with signature path," http://comparch-conf.gatech.edu/dpc2/, 2015, [The 2nd Data Prefetching Championship (DPC2)].

[21] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hållberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner, "Simics: A full system simulation platform," *Computer*, vol. 35, no. 2, pp. 50–58, Feb. 2002. [Online]. Available: http://dx.doi.org/10.1109/2.982916

[22] Y. Kim, W. Yang, and O. Mutlu, "Ramulator: A fast and extensible dram simulator," 2015.

[23] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation techniques for storage hierarchies," *IBM Syst. J.*, vol. 9, no. 2, pp. 78–117, Jun. 1970. [Online]. Available: http://dx.doi.org/10.1147/sj.92.0078

[24] A. Joshi, A. Phansalkar, L. Eeckhout, and L. K. John, "Measuring benchmark similarity using inherent program characteristics," *IEEE Transactions on Computers*, vol. 55, no. 6, pp. 769–782, June 2006.

[25] A. Phansalkar, A. Joshi, L. Eeckhout, and L. K. John, "Measuring program similarity: Experiments with spec cpu benchmark suites," in *IEEE International Symposium on Performance Analysis of Systems and Software, 2005. ISPASS 2005.*, March 2005, pp. 10–20.

[26] K. Hoste and L. Eeckhout, "Microarchitecture-independent workload characterization," *IEEE Micro*, vol. 27, pp. 63–72, 2007.

[27] R. Panda and L. K. John, "Data analytics workloads: Characterization and similarity analysis," in *2014 IEEE 33rd International Performance Computing and Communications Conference (IPCCC)*, Dec 2014, pp. 1–9.

[28] A. Yasin, Y. Ben-Asher, and A. Mendelson, "Deep-dive analysis of the data analytics workload in cloudsuite," in *2014 IEEE International Symposium on Workload Characterization (IISWC)*, Oct 2014, pp. 202–211.

[29] J. Lee, H. Kim, and R. Vuduc, "When prefetching works, when it doesn’t, and why," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 1, pp. 2:1–2:29, Mar. 2012. [Online]. Available: http://doi.acm.org/10.1145/2133382.2133384