Improving Memory Access Performance of In-Memory Key-Value Store Using Data Prefetching Techniques

PengFei Zhu[1(✉)], GuangYu Sun[2], Peng Wang[2], and MingYu Chen[1]

[1] ACSL, Institute of Computing Technology, Chinese Academy of Science, Beijing, China
{zhupengfei,cmy}@ict.ac.cn
[2] CECA, Peking University, Beijing, China
{gsun,wang_peng}@pku.edu.cn

Abstract. In-memory Key-Value stores (IMKVs) provide significantly higher performance than traditional disk-based counterparts. As memory technologies advance, IMKVs become practical for modern Big Data processing, which include financial services, e-commerce, telecommunication network, etc. Recently, various IMKVs have been proposed from both academia and industrial. In order to leverage high performance random access capability of main memory, most IMKVs employ hashing based index structures to retrieve data according to keys. Consequently, a regular memory access pattern can be observed in data retrieval from those IMKVs. Normally speaking, one access to index (hash table), which is also located in main memory, is followed by another memory access to value data. Such a regular access pattern provides a potential opportunity that data prefetching techniques can be employed to improve memory access efficiency for data retrieval in these IMKVs. Based on this observation, we explore various data prefetching techniques with proper architecture level modifications on memory controller considering trade-off between design overhead and performance. Specifically, we focus on two key design issues of prefetching techniques: (1) where to fetch data (i.e. data address)? and (2) how many data to fetch (i.e. data size)? Experimental results demonstrate that memory access performance can be substantially improved up to 35.4 %. In addition, we also demonstrate the overhead of prefetching on power consumption.

Keywords: In-memory key-value store · Data prefetching · Memory controller optimization

1 Introduction

As we have moved into the era of Big Data, a huge number of modern applications that relies on large-scale distributed storage systems have emerged. However, traditional relational database management systems (RDBMS) may be inefficient for many of them mainly due to the fact that features of RDBMS,

Y. Chen et al. (Eds.): APPT 2015, LNCS 9231, pp. 1–17, 2015.
DOI: 10.1007/978-3-319-23216-4_1

such as support of complicated SQL queries, are no longer necessary [4]. Therefore, the key-value (KV) store has become popular and been widely adopted in modern data centers to support various Internet-wide services. These well-known KV stores include BigTable [5], Cassandra [11], Dynamo [7], etc. Although these KV stores can provide higher performance and better scalability than traditional RDBMS, their performance is still limited by their underneath storage infrastructure that is based on hard disk drives (HDD). Thus, in order to satisfy increasing performance requirement of modern applications, the so-called in-memory KV stores (IMKVs) have attracted attention of storage researchers [20].

IMKV normally refers to a Key-Value (KV) store that uses main memory for data storage rather than disk-based storage. As the access speed to main memory is several orders faster than that to disk storage, IMKVs are usually employed for storage systems where response time is critical. These systems are widely adopted in financial services, e-commerce, telecommunication network, etc. [16] Recently, IMKVs become more and more attractive mainly because of advances in memory technologies. On the one hand, the decreasing price per bit of DRAM technology makes it possible to employ IMKVs for modern Big Data applications. On the other hand, various non-volatile memory technologies have potential to improve durability in IMKVs. Consequently, many IMKVs have been proposed from both academia and industry, which demonstrate 10-100x performance improvements over traditional disk-storage counterparts [2,3,18].

Since data are maintained in main memory, the efficiency of data accesses to memory is important for performance of these IMKVs. Compared to disk storage (either HDDs or SSDs), main memory has an intrinsic advantage that it supports high performance random accesses. Thus, unlike disk storage based databases, many IMKVs prefer using hash function based index structure to speed up index processing. Using such a hash index structure, we observe that the access pattern to main memory become more regular than that to traditional databases. Specifically, for those key-value stores based on IMKVs, one memory access to index is followed by another memory access to value data. The data access pattern provides a potential opportunity that data prefetching can be leveraged to improve performance of data retrieval in these IMKVs.

In fact, prefetching techniques have been widely researched in memory architecture design to improve efficiency of data access. Today, commercial microprocessors are equipped data prefetch engines to improve performance of memory-intensive workloads [1,23]. Based on hardware prefetcher, previous works [9,14] evaluate accurate measurement of performance metrics by prefetching technologies. The basic idea of prefetching is to predict data that may be accessed in future according to current states or execution history. Then, these data are loaded in advance into on-chip caches [25] or memory controller [28] to hide access latency. Obviously, the efficiency of prefetching relies on temporal or spatial locality of data access pattern to main memory. For data accesses in IMKVs, the regular access pattern enables the potential of employing prefetching techniques.

Through various different prefetching techniques, there are two major design issues in common. First, how to detect which data should be prefetched (e.g. prefetching address). Second, how many data should be loaded in prefetching (e.g. prefetching size). Previous approaches normally depend on execution states or history to predict prefetching address and prefetching size. However, it is not straightforward to directly apply existing techniques on IMKVs. In order to efficiently handle above design issues, we extensively explore data structures of IMKVs and propose several design techniques optimized for different working environments. The main contributions of this work are summarized as follows,

- With careful analysis of data accesses to in-memory KV stores, we reveal the fact that a regular access pattern can be observed, which can be leveraged for efficient data prefetching.
- Considering the data structure of KV stores, we propose a simple but efficient extension to memory controller to predict the prefetching size.
- In order to identify the prefetching address, we propose two techniques, which are suitable for different cases. The design trade-off between these two techniques is also analyzed.
- Comprehensive experimental results are provided to evaluate efficiency of applying our methods on a real in-memory KV stores.

2 Background and Motivation

In this section, we will present a brief review of data retrieval in hash-indexing based IMKVs using a state-of-art representative. In addition, we will reveal the fact that a regular access pattern to main memory can be observed.

In recent times, various IMKVs have become vital components in modern datacenter storages, including Memcached [2,18], Redis [3], MICA [15], MemC3 [8], and RAMCloud [22]. In these systems, all data are kept in DRAM at all times to provide the lowest possible storage latency for different applications. And most of these systems employ hashing based index structures as it provides a O(1) lookup time. In the rest of this section, we will use RAMCloud as a representative example to introduce how hash indexing works [22].

RAMCloud adopts a simple key-value data model consisting of binary objects that are associated with variable-length keys. Each RAMCloud server contains a collection of objects stored in a log area of DRAM via a log-structured approach and a hash table that points to every live object. As shown in Fig. 1, objects can only be accessed by their keys. It means that every object operation (access) interacts with the hash table. For example, in a read request from a client, the server must use the hash table to locate the object in the in-memory log area.

RAMCloud uses an open hashing method, in which there is only one possible bucket for a specific object in the table. If a bucket is full, additional chained entries are allocated separately to store more object references in the same bucket. RAMCloud servers will have tens or hundreds of gigabytes of DRAM, but the size of each object is likely to be quite small (a few hundred bytes or less). Therefore, the hash table may contain tens or hundreds of millions of individual

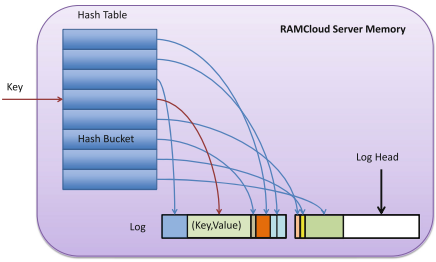

Fig. 1. Hash Table & Log in RAMCloud [22]

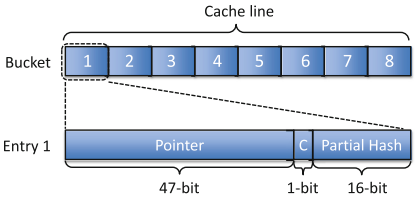

Fig. 2. Hash table bucket in RAMCloud.

entries. This means that the hash table working set is probably too large to be held in the processor's cache, and cache misses become unavoidable. RAMCloud expects each lookup will cause no more than two cache misses: one miss in the hash table bucket, and another to verify the key matches the object in log.

Several optimizations are applied in RAMCloud to reduce cache misses. As show in Fig. 2, each key is hashed into a specific bucket, which contains up to eight 47-bit direct pointers to objects within the in-memory log area. Each hash table bucket is aligned to a CPU cache line (64 bytes on current x86 processors, or eight hash entries). The hash table consists of a continuous array of such buckets. Accessing a bucket will often result in a cache miss, which loads a full cache line from memory. RAMCloud will then traverse all hash entries in the loaded cache line when doing a lookup.

To avoid retrieving each object referenced in the bucket in order to verify a match (i.e. to compare the key stored within the object in the log area), which would likely cause a cache miss, RAMCloud uses the upper 16 bits of each hash table pointer to store a partial hash of the key of the object referred. At this rate, if a bucket contains several valid entries, it is highly possible that at most one will have a matching partial hash in the bucket, so only one object will need to be accessed to compare the key. The remaining 47 bits are sufficient to locate the object within the in-memory log, and the 1-bit "C" flag indicates whether the bucket is chained due to overflow.

In summary, the most common memory access pattern in RAMCloud is: access one of hash table buckets, find one of hash entries in the bucket that matches the partial hash of the object's key, and then follow the pointer to retrieve the object in log area to compare the key and utilize the value of the object. It is confirmed by the memory trace we collected using Pin instrumentation tool [17].

For example, the traces of seven memory requests in a RAMCloud *get* operation are shown in Table 1. The first five traces represent memory requests in hash indexing, while the other two represent the data retrieving in log area. First, RAMCloud loads a bucket, which is exactly a cache line, into the cache. It then scans the entries in the bucket sequentially to find an entry with matching

Table 1. Trace example.

No.	Inst. Count	Inst. Addr.	Type	Mem. Addr.	Req. Size
1	15767034	0x473c10	R	0x82cfc300	64
2	15767695	0x473560	R	0x82cfc300	8
3	15767715	0x473560	R	0x82cfc308	8
4	15767735	0x473560	R	0x82cfc310	8
5	15767792	0x47352a	R	0x82cfc310	8
6	15767872	0x4a5a7a	R	0x7099a5c0	64
7	15767876	0x4a5a7a	R	0x7099a600	64

partial hash, as shown in Trace 2–4. After three failed attempts, RAMCloud finds the right hash, unpack the entry, and follow the pointer to retrieve the key-value pair stored in the log area in Trace 5. Finally it fetches the corresponding records in the log, extract the key, and make a comparison with given key as shown in Trace 6–7.

From this example, we can find that it takes more than eight hundred instructions between the access to the hash table and the one to data. Thus, if we can prepare data in advance before data access requests happen, the memory access performance can be improved. In the next section, we will introduce how to achieve this with data prefetching.

3 Prefetching Architecture Design

In this section, we first provide an overview of prefetching architecture proposed in this work. Then, the key components employed in this architecture are introduced in details.

3.1 Structure Overview

A computer system running IMKV is illustrated in Fig. 3. Note that other cache levels other than last level cache (LLC) are hidden to save space. As shown in Fig. 3, in order to enable dedicated prefetching mechanisms in IMKV, several extra components are added into the memory controller. These components include "index range registers" (IRRs), a "prefetching address control unit" (PACU), and a "prefetching size control unit" (PSCU). The basic flow of prefetching is described in details as follows.

As shown in Fig. 4, while a memory request is processed to access memory (step 1), it is sent to IRR at the same time (step 2). The purpose of IRRs is to detect whether the memory request is accessing the hash index or not. If the request is accessing index, a data prefetching process is triggered. The index data retrieved from memory are sent to PACU to detect the addresses to prefetch data (step 3). Then, these addresses are sent to PSCU to identify the

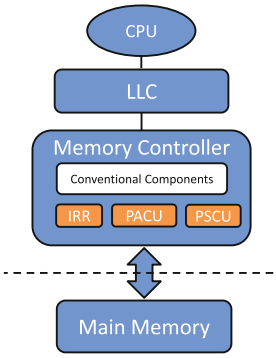

Fig. 3. Architecture overview.

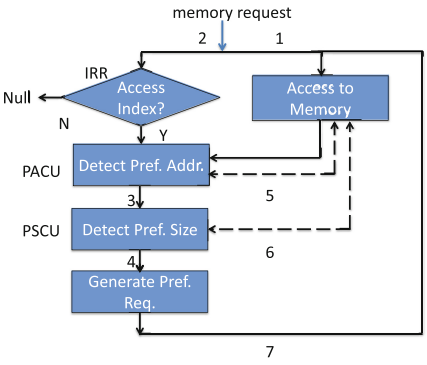

Fig. 4. Memory access flow with prefetching.

size of data to be prefetched (step 4). Note that both PACU and PSCU may also access memory based on the mechanisms adopted in them (steps 5 and 6). With information of prefetching addresses and sizes, the corresponding prefetching requests are generated (step 7). Note that one or multiple prefetching requests may be generated based on output from PACU and PSCU.

Apparently, the efficiency and design overhead of prefetching depends on design of these components. In following subsections, design details of these components will be introduced. Especially, for PSCU and PACU, different architectures are explored.

3.2 Index Range Register (IRR)

As mentioned before, we rely on IRRs to determine whether a memory request is trying to access the hash table of an IMKV so that a proper prefetching is triggered. This modification to memory controller is feasible. In fact memory controllers provide sets of software accessible registers [23]. Since the hash table usually accommodates a contiguous range of virtual addresses, we need two programmable Index Range Registers (IRRs) to keep the start and end address of the hash table. Note that we assume that all memory resource with a memory controller is allocated to one IMKV. Thus, only one set of IRRs are needed.

We address that one obstacle of using IRR is that physical addresses instead of virtual addresses of memory requests are sent to the memory controller for response. One possible solution is to modify hardware to send both physical and virtual addresses to memory controller. However, the design overhead is non-trivial. Instead, we modify the kernel library of memory allocation to ensure that a contiguous range of physical addresses is allocated to hash table during initialization of an IMKV.

We design a set of system calls that allow applications to change the value of IRRs. Currently we have modified applications by hand to insert the system

calls to set the lower/upper bound when the IMKV requests memory allocation for the hash table. Note that this process can be automated by compiler in the future. With the help of IRRs, all memory requests whose addresses are in the hash table range are forwarded to the PACU, which is introduced in the next subsection.

3.3 Prefetching Address Control Unit (PACU)

With the help of IRR, each access to the index of IMKV has potential to trigger a data prefetching to speed up the following request of value retrieval. In order to achieve efficient data prefetching, the first critical design issue is to find out the starting address of value data from index information. However, this process is not straightforward. For example, in RAMCloud, each hash bucket has eight index entries. It means that, there are eight prospective addresses of value data in a single bucket.

Naive Exhausted Prefetching. One simple solution is to prepare all potential value data within the same hash bucket before the address of correct value is computed. For example, in RAMCloud, all value data indexed by valid entries in the same bucket are prefetched. Although these indexes are located in the same bucket, the data may be distributed in different ranks or banks. Thus, it is possible to leverage the parallelism of main memory.

This method works efficiently when the index utilization is low. In other words, if there are too many valid entries in a bucket, we may not gain any benefits from data prefetching. The reason can be explained in two-folds. First, due to limited memory rank and bank numbers, the more values we prefetch at the same time, the higher probability they may conflict during prefetching. Thus, the efficiency decreases as the utilization of index memory increases. Second, prefetching too many data at the same time will also impact other memory accesses. In Sect. 4, we will demonstrate that this simple method cannot work well when the average utilization is more than 50 %.

In order to identify proper value to be prefetched rather than prefetching all of them, we further propose two types of techniques in this work: *in-situ index processing* and *value address prediction*.

In-situ Index Processing. In-situ index processing architecture is extended from the accelerator design called Widx [10]. The basic idea is to processing hash index lookup inside memory controller with dedicated hardware design. Then, the corresponded data is prefetched. Similar to Widx, dedicated processing logic are required to perform hash index lookup. However, the Widx architecture needs to be substantially modified when being adopted in our design. It is mainly because Widx is proposed as a co-processor for relational DBMS, which is different from our target, in-memory KV store, in this work.

First, since the prefetching is simply triggered by accessing hash table, the accelerator for hash index lookup is not explicitly controlled by IMKV. In addition, the extra overhead for communication between CPU and accelerator is

significantly reduced. However, we need a dedicated RAM in memory controller to store the instructions for processing hash function. Second, Widx only works with linked-list style hash index structure. Thus, modification is needed to make our design work with bucket data structure like that in RAMCloud. The advantage of RAM is that in-situ index processing can be extended to work with different hash index structure through uploading dedicated instructions. Third, different with Widx, which only returns index lookup result to processor, our design needs to issue memory prefetching requests based on the lookup result.

Value Address Prediction. Obviously, in-situ index processing can always find out the correct address to prefetch data. However, its major drawback is that substantial design overhead may be induced due to several reasons. First, since the original key is normally required in memory controller for further index processing, extra hardware support is required. In addition, software level modification is needed to identify the key to be forwarded. Second, the design overhead inside memory controller for index processing (e.g. hash function) is non-trivial. In order to overcome this limitation, we further propose alternative techniques based on prediction.

The prediction technique is to leverage the temporal locality in data access patterns. In other words, for data indexed in the same hash bucket, one of them may be accessed repeatedly during a period. For such access patterns with good temporal locality, there is high possibility that the entry containing correct data addressed in the last access will be accessed again. Thus, it is beneficial to prefetch data indexed by this entry.

The key issue of prediction is to indicate the entry containing correct address in the last data retrieval. We propose to design additional hit table in hardware to record the hit history of these index entries. One critical issue is to decide the size of hit table. For example, in RAMCloud, a 2 GByte hash table requires a 12 MByte hit table to fully record hit history of all hash entries. To fully integrate such size hit table in memory controller is not feasible due to both area and scalability issues. Like Centaur [24], we design hit table in a memory buffer between memory controller and main memory, rather than in memory controller or main memory. The hit table shares the same index from key hashing. PACU can acquire hit table entry off-chip when it is triggered by IRRs. Advantages to implement hit table in memory buffer is faster access than in main memory and better scalability than fixed size in memory controller. In case that we have to consider overhead of hit table to reduce its size at huge hash table, it is possible that it cannot fully cover all hashing entries. In such cases, if the key hashing index is out of the range of hit table, a default prediction scheme of loading the first entry in the hash bucket can be employed. In Sect. 4, we evaluate hit table in constant latency, and also maximum size to cover whole hash table. The other critical issue is how to update hit table. Through modification of *get* operation, we propose IMKVs to update hit table every time when data retrieving finish. By this way, hit table contains latest access entries in hash table. Although interferences from different threads and bad temporal or spatial

locality in data retrieving reduce predication accuracy, as show in Sect. 4, we will demonstrate that this simple method still works well when accuracy is as low as 20 %. Especially, the fact that accuracy of predication is increased as hash table utilization is decreased opens a door for IMKVs to control accuracy in need.

3.4 Prefetching Size Control Unit (PSCU)

With the help of PACU, we can decide where to prefetch data. Then, the next critical issue is to determine how many data we should prefetch from main memory. As shown in Fig. 3, we rely on the component called prefetching size control unit (PSCU) in this work. There are two corresponding choices for design of PSCU, which are introduced in the following two paragraphs. Although the determination of prefetching size is orthogonal to the design of PACU, these two components affect each other on the efficiency of prefetching.

Run-Time Size Determination. Similar to in-situ index processing, we can leverage the accelerator in memory controller to detect the prefetching size based on the data stored at prefetching address. In other words, we employ dedicated logic to obtain the size of value dynamically. This method is feasible because the size of a value is normally stored together with value data in KV store. For example, the first several bytes of value data in RAMCloud contains the size information. Obviously, if in-situ processing accelerator is employed in PACU, the hardware can be shared with PSCU to accurately determine prefetching address and prefetching size.

Average Size Profiling. When the naive prefetching or the address prediction technique is employed, it is not efficient to add an accelerator just for determination of prefetching size. Thus, a profiling based technique is preferred for in these two cases.

- *Static Profiling.* A simple but efficient method is to perform static profiling in advance to calculate the average value size. This static method is preferred when the size of values do not vary a lot.
- *Dynamic Profiling.* An alternative is to determine the run-time average value dynamically. This method is more efficient than static one when the value sizes vary significantly. However, it requires extra support to record the history of value size. In order to reduce hardware overhead, we propose software method profiles past accessed size of KVs and update PSCU periodically.

Both static and dynamic profiling methods are based on facts that in parts of IMKVs application, such as financial services, KV-pair's size does not show big variations. When the prefetching size does not match the value size, the efficiency of prefetching is decreased. If the size of prefetched data is smaller than the real value size, supplementary memory request from CPU is needed to fetch the rest of value data. On the contrary, if the prefetching size is larger than the real value size, memory bandwidth is wasted and other normal memory requests may be affected.

4 Evaluation

In this section, we first introduce the setup for experiments. Then, we provide comprehensive results and analysis. In addition, the design overhead is discussed.

4.1 Experiment Setup

We conduct the experiment on customized trace-driven cycle-accurate simulator, which supports a full CMP architecture, including out-of-order multi-issue multi-processors, two-level cache hierarchy, shared-cache coherent protocol, 2-D mesh NOC, main memory controller and DRAM device. Be specific, CPU, cache hierarchy and NOC are modeled by home-made simulator, while the memory controller and DRAM device are modeled by DRAMSim2 [21], and power is evaluated on DRAM device. Performance is evaluated by average memory request latency. The detailed configuration of the experiment and parameters used in simulation are shown in Table 2.

Table 2. Detailed configuration of experiment platform

Unit	Configurations
CPU	8 Intel cores, 4 GHz, 128 instruction windows, 4issue/4commit (one memory op) per cycle, 16 MSHR
L1	Private 16 KB 4-way set associative, 64 B line, LRU, R/W 1/1-cycle
L2	Shared 4 MB 16-way set associative, 64 B line, LRU,
Cache-coherent	Directory based cache-coherent protocol: MESI
NOC	2×4 mesh NOC, one router per node, x-y direction based routing, 8 flits per data packet, 1 flit per control packet
Memory controller	2 memory controllers, 16 GB, 32 entries transaction queue,
	32-entry command queue, FR-FCFS scheduling, open page policy, rank-interleave address mapping
DRAM device	Micron DDR3-1333 Mhz, x8, 8Banks, 32768 Rows/Bank, 1024 Columns/Row, 1 KB page size, BL = 8

The benchmarks are run on RAMCloud [22], which is a widely-adopted multithread IMKV application. We use typical IMKV requests (128 B, 256 B and 512 B value size) from YCSB [6] benchmark to drive KV operations in RAMCloud. To setup initial database in RAMCloud, we generate sufficient KV pairs to initialize RAMCloud's memory, in total 1 GB segmented log and 100 MB hash table. Based on execution of RAMCloud, we collect traces of multithreaded KV operations on real CMP machine and feed traces to simulator. During simulation, we execute 10 billion instructions of KV benchmarks for rapid estimation. Table 3 shows 12 typical mixed workloads of KV operations in IMKV requests, which have an increasing size of average value size in Byte.

4.2 Experiment Results

In order to address the impact of each design factor related to design of PACU and PSCU, the synthetic workloads are first simulated. Then, the evaluation using real workloads is discussed. Note that we use normalized average "memory request latency" as the metric of memory access performance.

Synthetic Workloads Results. In Fig. 5, the impact of prefetching size on performance is evaluated. There are three sets of workloads, in each of which the value size is fixed. Assume that the static profiling method is employed. The prefetching size varies from 128 B to 1024 B to demonstrate its impact. In this experiment, the correct prefetching address is always provided to isolate the effect of PACU. For comparison, the baseline without using any prefetching is also presented and all results are normalized to it.

As shown in the Fig. 5(a), the best performance is achieved when the prefetching size matches the value size. In addition, we can observe that the efficiency of prefetching increases with value size. When the prefetching size is smaller than the value size, the efficiency of prefetching decreases. But, we can find that performance is still improved compared to the baseline. It is because part of value data is prefetched and the rest is requested by CPU through normal access. The results also show that prefetching efficiency is reduced when more data than value are prefetched. It is because its effect on memory bandwidth and other normal requests has offsets its benefits from data prefetching.

In Fig. 5(b), the efficiency of naive exhausted prefetching is evaluated. Similar to last experiment, there are three sets of workloads with fixed value size. In order to isolate the effect of prefetching size, we assume that prefetching size always matches the value size. For each workload, we vary the average hash bucket utilization through software level control, so that prefetching size by naive exhausted method varies from 2 KVs to 8 KVs. Besides the baseline, we also present one set of result using in-situ hash processing when utilization is 4 KVs per hash index. Here, we skip 1 KV case, because hash table utilization is too low, and exhausted method always prefetch correct data as in-situ method. From the results we can tell that the efficiency of naive exhausted prefetching decreases as the utilization of bucket increases. Normally, we cannot gain any

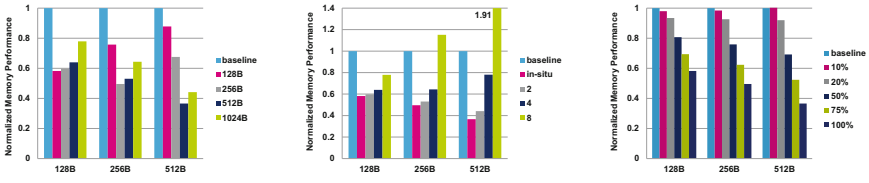

Fig. 5. (a) Effect of prefetching size. (b) Efficiency of naive exhausted prefetching. (c) Effect of prefetching accuracy.

Table 3. Detailed configuration of workloads

Workloads	128B-R	128B-W	256B-R	256B-W	512B-R	512B-W	Avg(Byte)
mix1	96.7 %	0.8 %	1.5 %	0.1 %	0.6 %	0.5 %	134.1
mix2	70.0 %	9.5 %	7.4 %	2.4 %	8.2 %	2.7 %	182.3
mix3	34.1 %	34.0 %	10.4 %	5.5 %	10.3 %	5.7 %	210.1
mix4	37.2 %	19.3 %	24.4 %	5.1 %	8.7 %	5.3 %	219.5
mix5	13.4 %	10.5 %	61.6 %	12.5 %	1.5 %	0.5 %	230.5
mix6	15.0 %	10.0 %	37.5 %	27.5 %	7.5 %	2.5 %	249.6
mix7	17.4 %	6.6 %	38.2 %	15.6 %	15.2 %	7.1 %	282.5
mix8	8.0 %	5.0 %	32.5 %	22.5 %	27.0 %	5.0 %	321.3
mix9	17.1 %	5.1 %	13.1 %	5.0 %	58.6 %	1.3 %	380.8
mix10	7.5 %	2.5 %	15.0 %	10.0 %	37.5 %	27.5 %	409.6
mix11	5.7 %	1.1 %	19.8 %	7.7 %	54.8 %	11.2 %	416.1
mix12	6.4 %	5.2 %	7.3 %	6.8 %	37.8 %	36.8 %	432.0

benefits when the utilization is higher than 50 %. In addition, we can tell that its impact on performance increases with the value size.

In Fig. 5(c), the efficiency of address prediction is evaluated, in respect of prediction accuracy. In this experiment, we assume that the value size is fixed and is known in advance for each workload. Thus, the performance is only affected by the prefetching address prediction. For each workload, we vary the prediction accuracy from 10 % to 100 %. Note that the case of 100 % accurate reflects the result when the in-situ hash processing is employed. We can find that performance is improved by more than 20 % for all workloads when the prediction accuracy is higher than 50 %. In worse case, 20 % predication accuracy still achieves about 10 % improvement. The reason is the temporal locality in IMKVs is different from scientific applications, thus the replacement of cache line by prefetched data doesn't impact much on miss rate of last level cache. It proves the feasibility of prefetching address prediction in PACU, such as hit table method, especially for large value size.

Real Workloads Results. In this section, we present performance simulation results based on real workloads listed in Table 3. In addition, the energy overhead caused by prefetching is also included.

We repeat an experiment similar to that in Fig. 5(a), in which different prefetching sizes are applied with real workloads. Normalized performance results are shown in Fig. 6. Besides the baseline without using prefetching, the in-situ case using in-situ processing for both prefetching address and size is also compared in the Fig. 6. We can find that the efficiency of prefetching relies on proper prefetching size. In-situ case gains best performance due to accuracy. The second optimized prefetching sizes found in the figure for different workloads are closed

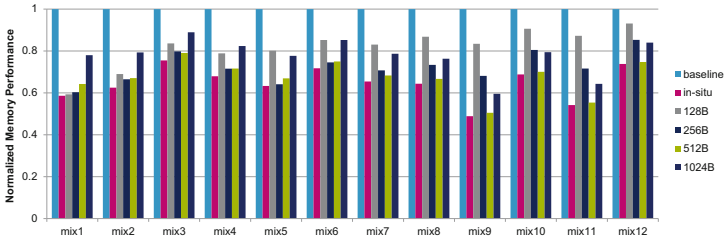

Fig. 6. Effect of prefetching size with real workloads.

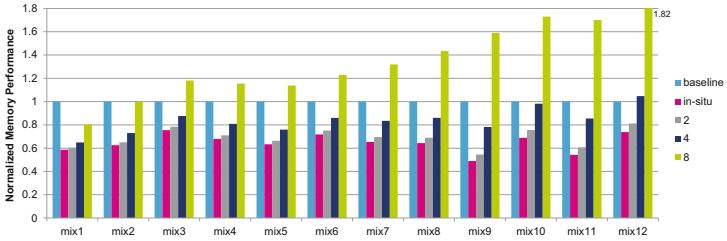

Fig. 7. Efficiency of naive exhausted prefetching with real workloads.

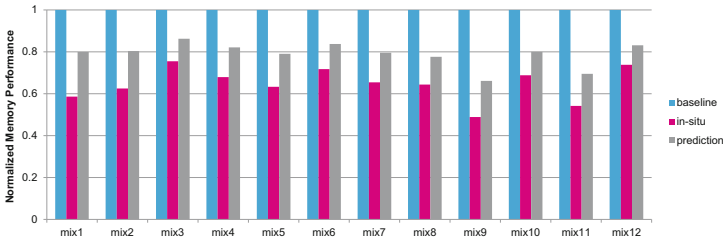

Fig. 8. In-situ processing vs. prediction.

to the average value sizes calculated in Table 3. It proves average size profiling method can achieve improvements on performance.

In Fig. 7, the naive exhausted prefetching with in-situ method is applied with real workloads. We can draw the similar conclusion that it only works when the bucket utilization is lower than 50 %, and its offsets impact on performance increases with the average value size. Basically, it is hard for hash table with full utilization of buckets to gain benefits through naive exhausted prfetching.

In Fig. 8, we compare the in-situ case with best performance to case with lowest design overhead. As mentioned before, the in-situ case is to use in-situ processing for both prefetching address and size with most accuracy. The case with lowest overhead is to use dynamic profiling and the prefetching address prediction (e.g. hit table method). Experiment shows hit table method can achieve about 50 % accuracy of address predication on average. We find that the prefetch-

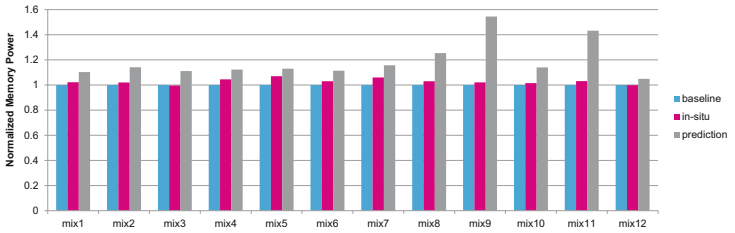

Fig. 9. Extra power consumption.

ing based on simple profiling and prediction can also improve performance. On average, performance is improved by 35.4 % in in-situ case and is improved by 21 % in the latter case.

Although prefetching can help improve performance of IMKV, it also induces extra power consumption. In Fig. 9, we demonstrate the normalized power results for different workloads. We can find that the power overhead is trivial for in-situ prefetching case. It is because prefetching only changes the sequence of load data value without inducing extra memory requests. However, with the prediction based prefetching, the power consumption is not always negligible due to incorrect prediction and dynamic profiling.

5 Related Work

Most in-memory stores using hash table as indexing structure: Memcached [2, 18], Redis [3], RAMCloud [22], MemC3 [8], and MICA [15] all exploit hashing to achieve low latency and high performance. Standard Memcached uses a classical hash table design to index the key-value entries, with linked-list-based chaining to handle collisions. Its cache replacement algorithm is strict LRU, also based on linked lists. RAMCloud [22] employs a cache-optimized hash table layout to minimize the memory cache misses. MemC3 [8] is optimized for read-mostly workload by applying CLOCK-based eviction algorithm and concurrent optimistic cuckoo hashing. MICA [15] enables parallel access to partitioned data, utilizes lossy concurrent hash indexes, and bulk chaining techniques to handle both read- and write-intensive workloads. These mechanisms could be layered on top of our data prefetching schemes to achieve the same goals.

Prefetching is a commonly used method to hide the increasing latency of accesses to main memory. Various studies have been conducted to investigate the benefits of prefetching. These techniques can be classified as software-controlled or hardware-controlled. Software-controlled prefetching techniques [19,26] use special prefetch instructions to asynchronously pre-load cache blocks. Additional instructions must be inserted and executed in the applications. Unlike software-controlled method, hardware-controlled prefetching techniques [12] construct pre-fetcher triggered by dedicated conditions to retrieve data in advance. PADC [12] estimates the usefulness of prefetch requests, adaptively prioritize between demand and prefetch requests, and drop useless prefetches. Lee

et al. [13] study the DRAM bank-level parallelism issues in the presence of prefetching. Based on commercial microprocessors equipped with data prefetch engines [23], [9] evaluate accurate measurement of performance metrics through adaptive prefetching scheme depending workloads natures. According to location of prefetching initiator, memory-side prefetching [27], in tandem with processor-side prefetcher to leverage knowledge of DRAM state, answers what/when/where to prefetch. In contracts, our work focuses on hash table prefetching in IMKVS and therefore has the application-specific knowledge of regular access pattern in workloads to improve memory access performance.

Co-processor has been widely used for acceleration of specific applications. Recently, for hashing index based IMKV, Babak et al. propose Widx [10], an on-chip accelerator for database hash index lookups. Widx uses a custom RISC core to achieve high-performance hashing computation. Widx walks multiple hash buckets concurrently to exploit the inter-key parallelism. We extend Widx in memory controller as in-situ method to cover not only linked-list style, but also bucket style hash index structure, to issue accurate prefetching requests to improve data retrieving based on hash lookup result.

6 Conclusion

In-memory KV stores have been extensively employed for modern applications for high performance data retrieval. Since hashing based index is widely adopted in these KV stores, the memory access patterns for data retrieval are regular. Thus, data prefetching technique can be employed to improve performance of memory access. In this work, with detailed analysis of data access pattern in real IMKVs, we propose several practical prefetching techniques. The in-situ processing based prefetching can achieve the best performance but also induces most overhead. The prediction and profiling based prefetching can also improve performance with moderate design overhead. However, it may induce non-trivial power overhead. Considering the trade-off, proper prefetching should be adopted in real cases for different design goals.

Acknowledgements. This work was partially supported by National High-tech R&D Program of China (2013AA013201) and in part by National Natural Science Foundation of China (61202072, 61272132, 61221062).

References

1. Intel 64 and IA-32 Architectures Software Developers Manuals. www.intel.com/products/processor/manuals
2. Memcached. http://memcached.org/
3. Redis. http://redis.io/
4. Cattell, R.: Scalable SQLA and NoSQL data stores. SIGMOD Rec. **39**(4), 12–27 (2011)

5. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: a distributed storage system for structured data. ACM Trans. Comput. Syst. **26**(2), 4:1–4:26 (2008)
6. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: Proceedings of the 1st ACM Symposium on Cloud Computing, pp. 143–154. ACM (2010)
7. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: Amazon's highly available key-value store. In: Proceedings of the 21st ACM Symposium on Operating Systems Principles, SOSP 2007, pp. 205–220. ACM, New York (2007)
8. Fan, B., Andersen, D.G., Kaminsky, M.: MemC3: compact and concurrent memcache with dumber caching and smarter hashing. In: Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation, NSDI 2013, pp. 371–384. USENIX Association, Berkeley (2013)
9. Jiménez, V., Cazorla, F.J., Gioiosa, R., Buyuktosunoglu, A., Bose, P., O'Connell, F.P., Mealey, B.G.: Adaptive prefetching on POWER7: improving performance and power consumption. ACM Trans. Parallel Comput. **1**(1), 4:1–4:25 (2014)
10. Kocberber, O., Grot, B., Picorel, J., Falsafi, B., Lim, K., Ranganathan, P.: Meet the walkers: accelerating index traversals for in-memory databases. In: Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 46, pp. 468–479. ACM, New York (2013)
11. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. SIGOPS Oper. Syst. Rev. **44**(2), 35–40 (2010)
12. Lee, C.J., Mutlu, O., Narasiman, V., Patt, Y.N.: Prefetch-aware DRAM controllers. In: Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 41, pp. 200–209. IEEE Computer Society, Washington, DC (2008)
13. Lee, C.J., Narasiman, V., Mutlu, O., Patt, Y.N.: Improving memory bank-level parallelism in the presence of prefetching. In: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 42, pp. 327–336. ACM, New York (2009)
14. Liao, S.w., Hung, T.H., Nguyen, D., Chou, C., Tu, C., Zhou, H.: Machine learning-based prefetch optimization for data center applications. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC 2009, pp. 56:1–56:10. ACM, New York (2009)
15. Lim, H., Han, D., Andersen, D.G., Kaminsky, M.: MICA: a holistic approach to fast in-memory key-value storage. In: 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, pp. 429–444. USENIX Association, Seattle (2014)
16. Loos, P.D.P., Lechtenbrger, J., Vossen, G., Zeier, A., Krger, J., Mller, J., Lehner, W., Kossmann, D., Fabian, B., Gnther, O., Winter, R.: In-memory databases in business information systems. Bus. Inf. Syst. Eng. **3**(6), 389–395 (2011)
17. Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V.J., Hazelwood, K.: Pin: building customized program analysis tools with dynamic instrumentation. In: ACM Sigplan Notices, vol. 40, pp. 190–200. ACM (2005)
18. Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H.C., McElroy, R., Paleczny, M., Peek, D., Saab, P., Stafford, D., Tung, T., Venkataramani, V.: Scaling memcache at Facebook. In: Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013, pp. 385–398. USENIX, Lombard (2013)

19. Ortega, D., Ayguadé, E., Baer, J.L., Valero, M.: Cost-effective compiler directed memory prefetching and bypassing. In: Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques, PACT 2002, pp. 189–198. IEEE Computer Society, Washington, DC (2002)
20. Plattner, H., Zeier, A.: In-memory Data Management: Technology and Applications. Springer Science & Business Media, Heidelberg (2012)
21. Rosenfeld, P., Cooper-Balis, E., Jacob, B.: DRAMSim2: a cycle accurate memory system simulator. Comput. Archit. Lett. **10**(1), 16–19 (2011)
22. Rumble, S.M., Kejriwal, A., Ousterhout, J.K.: Log-structured memory for DRAM-based storage. In: Schroeder, B., Thereska, E. (eds.) Proceedings of the 12th USENIX Conference on File and Storage Technologies, FAST 2014, Santa Clara, CA, USA, 17–20 February 2014. pp. 1–16. USENIX (2014)
23. Sinharoy, B., Kalla, R., Starke, W.J., Le, H.Q., Cargnoni, R., Van Norstrand, J.A., Ronchetti, B.J., Stuecheli, J., Leenstra, J., Guthrie, G.L., Nguyen, D.Q., Blaner, B., Marino, C.F., Retter, E., Williams, P.: IBM POWER7 multicore server processor. IBM J. Res. Dev. **55**(3), 1:1–1:29 (2011)
24. Stuecheli, J.: Next Generation POWER microprocessor. http://www.hotchips.org/archives/2010s/hc25/
25. Wu, C.J., Jaleel, A., Martonosi, M., Steely, Jr., S.C., Emer, J.: PACMan: prefetch-aware cache management for high performance caching. In: Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 44, pp. 442–453. ACM, New York (2011)
26. Wu, Y.: Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching. In: Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation, PLDI 2002, pp. 210–221. ACM, New York (2002)
27. Yedlapalli, P., Kotra, J., Kultursay, E., Kandemir, M., Das, C.R., Sivasubramaniam, A.: Meeting midway: improving CMP performance with memory-side prefetching. In: Proceedings of the 22nd International Conference on Parallel Architectures and Compilation Techniques, PACT 2013, pp. 289–298. IEEE Press, Piscataway (2013)
28. Zhao, C., Mei, K., Zheng, N.: Design of write merging and read prefetching buffer in DRAM controller for embedded processor. Microprocess. Microsyst. **38**(5), 451–457 (2014)