SEER-MCache: A Prefetchable Memory Object Caching System for IoT Real-Time Data Processing

SEER-MCache: A Prefetchable Memory Object Caching System for IoT Real-time Data Processing

Dingding Li, Mianxiong Dong, Yanting Yuan, Jiaxin Chen, Kaoru Ota, and Yong Tang

***Abstract*—Memory object caching systems, such as Memcached and Redis, have been proved to be a simple and high-efficient middleware for improving the performance of IoT (Internet of Things) devices querying the database in cloud. However, its performance guarantee is built on the fact that the target data, queried by the IoT device, will be accessed many times and hit in the caching system. Therefore, when database system is handling the unrepeated IoT queries, it usually presents the sub-optimal performance, which greatly impairs the efficiency of real-time data processing on IoT devices. To improve this issue, we propose Seer-MCache, the memory object caching system with a smart prefetching (read-ahead) function, to fill up the caching system with the desired data before the intensive IoT queries arriving. Seer-MCache includes a set of rules to launch the specific behaviors of read-head. These rules are able to be customized according to the workload characteristics and system load. We implement a prototype system in Redis (caching layer) and MySQL server (database system). Extensive experiments are conducted to verify the effectiveness of Seer-MCache, the results show that Seer-MCache can improve the performance of read-intensive workload up to 61% (39.5% in average). Meanwhile, the cost of the read-ahead behavior is moderate and controllable.**

***Index Terms*—Internet of things (IoT), database, memory object caching system, read-ahead.**

I. Introduction

SInce always-on IoT (Internet of Things) devices or sensors proliferate, their data are required to be fetched and actionable in the moment [1] [2]. Mainly for improving their energy consumptions and lifetimes, usually a data infrastructure behind IoT applications, such as database in cloud, is deployed to decouple the intensive computing and massive storage from IoT devices [3] [4]. However, in the era of big data, the IoT application produces data streams previously unimaginable, both in variety and quantity [5]. As a result, the main components in a typical database, such as query parser (CPU-intensive, mainly for parsing the SQL statements) and storage engine (I/O intensive, mainly for issuing the specific read/write requests), are incurring the unprecedented load pressure [6], consequently shows a performance bottleneck either on CPU or I/O subsystem [7]. This problem greatly affects the quality of real-time data processing on IoT devices [8].

D. Li, Y. Yuan, J. Chen and Y. Tang are with the School of Computer Science, South China Normal University, Guangzhou 510613, China. E-mail: {dingdingli, sherryyuan, jiaxinchan, ytang}@m.scnu.edu.cn.

M. Dong and K. Ota are with the Department of Information and Electronic Engineering, Muroran Institute of Technology, Muroran 050-8585, Japan. E-mail: {mx.dong, ota}@csse.muroran-it.ac.jp

Manuscript received January 11, 2018; revised April 4, 2018.

An improvement on such issue is *horizontal scalability*, namely forking the extra database nodes and then distributing the IoT data streams uniformly among them [9]. However, it is not a cheap method, because: (1) a complex protocol is required to run on different nodes collaboratively [10]; (2) the multiplied maintenance cost on many database nodes is expensive [11]. To relieve this problem, memory object caching system, such as Redis [12] [13], is deployed alongside with the database in cloud, for decoupling a portion of work from the traditional database system [14]. For example, Redis acts as a *front-end* cache layer for the *back-end* database system, storing a portion of SQL results for satisfying the following IoT queries [15]. When the same or similar IoT query arrives, it can directly return the results to devices, instead of handling on the back-end. Not like a traditional database, such memory object caching system is simple since it is no need to maintain the ACID features (Atomicity, Consistency, Isolation and Durability) [16]. Therefore, it is much lightweight to be distributed, which allows it to be a cost-effective method to scale the database performance as the number of IoT devices increased. Currently, memory object caching system is wildly used by many companies and in countless IoT production environments[1].

However, as a cache role, the modern memory object caching systems lack the prefetch or read-ahead function, mainly for keeping their own simplicities. But in fact read-ahead is an important feature for a typical high-performance database or file system, such as `Superfetch` service in Windows 7 and linear/random read-ahead in InnoDB storage engine of MySQL server [17]. By leveraging the principle of locality, read-ahead function in a system can get the data, which will be accessed in the near future, into the caching layer in advance. If the used read-ahead policy is reasonable, a portion of new IoT queries will be completed successfully in the front-end cache, instead of incurring a process to find the data in back-end, which involves both costs of communication and disk I/O [18]. Therefore, the deficiency of read-ahead function on a Redis-like caching layer may bring the following downsides:

- Warming a cache needs an uncertain time, denoted by T_w, to fill up the desired or correct data in front-end. But T_w may be delayed in absence of the read-ahead function, thus punishing the performance of IoT query which tries to find the target items during T_w;

[1]https://www.infoworld.com/article/3063161/nosql/why-redis-beats-memcached-for-caching.html

- During T_w, the back-end database system is almost exposed to the external IoT queries directly. If the number of IoT queries are massive, the database system is easily congested and even off-line;
- IoT devices are less likely to always issue different queries for the same item because the external physical environment is complex. As a result, the issued IoT queries do not benefit from the front-end caching layer.

To improve above issues, we propose Seer-MCache, a dedicated and lightweight read-ahead function for the memory object caching system of IoT devices. The idea behind Seer-MCache is straightforward: when an item I, has been fetched from the back-end database system to respond a IoT query, not only I is cached in the front-end caching layer, but also $\{J\}$, a set of items which have correlations with I, are cached in front-end. Seer-MCache defines these correlations as $\{C_i\}$, which are designed to allow $\{J\}$ to be accessed later with high probability. All elements in $\{C_i\}$ are rules built from the *principle of locality* [19], at the layer of database semantic.

However, many rules used in Seer-MCache trigger many read-ahead behaviors in the back-end, which involve a lot of disk I/O and networking, thus incurring a large resource consumption on the original system. This side-effect will degrade the response time of IoT query in a loaded data infrastructure. Therefore, Seer-MCache provides a modular design on $\{C_i\}$ deployment. Namely, rules in $\{C_i\}$ can be applied into system as any combination and be featured in the *hot-plug* manner. In this way, Seer-MCache can orchestrate $\{C_i\}$ carefully depending on the system load of data infrastructure and workload characteristics of IoT application.

We implement Seer-MCache in a real system with Redis (front-end) and MySQL server (back-end). Extensive experiments, including micro-benchmarks and realistic workloads, are conducted to demonstrate the effectiveness of Seer-MCache. The results show that Seer-MCache can: (1) reduce the latencies of IoT queries at the period of cold cache; (2) bring the controllable cost for running read-ahead function and (3) present better IoT querying performance on an original system.

In summary, we have made the following contributions in this paper:

1) For the caching system of IoT applications, we devise a full read-ahead framework in a typical memory object caching system;
2) We propose two read-ahead policies to drive the specific read-ahead procedure: *passive* and *proactive*;
3) We implement the practical read-ahead mechanisms in a real memory object caching system of IoT devices;
4) Finally, We not only use both the quantitative and qualitative experiments to show the performance benefits of Seer-MCache, but also reveal its penitential cost when dealing with the read-intensive, read-write mixture and write-intensive IoT workloads.

The rest of this paper is organized as follows. In section II, we provide a motivating and real example to exemplify the necessity of read-ahead function in a memory object caching system of IoT. Then, we describe the design and implementation of Seer-MCache in section III and IV respectively. Section V describes the experimental methodology and discusses the results. Then we discuss the related work in section VI. Finally, section VII concludes the paper.

II. A Motivating Example

Here exemplifies a real system for large-scale real-time face recognition. Generally, there are three stages to recognize a persons face: S_1, capturing the video streaming and selecting several key frames based on the shot content change ratio [20] at the sensor layer[2]; S_2, sending the key frames to the computation layer and extracting the face features of people at the computation layer; S_3, delivering these features to the data layer, for finding the matched records of the captured people in S_2.

Under rudimentary design, we use a variant algorithm of *Eigenface* to identify the faces, in which a serial of features (denoted by $\{F_i\}$) is extracted at S_2. Then we use $\{F_i\}$ to find the matched record from the backing database system at the data layer, in which a query is issued from the computation layer to the data infrastructure to find the best-matched record according to $\{F_i\}$. Each record in the database also has the unique features, denoted by $\{f_i\}$. If $\{F_i\}$=$\{f_i\}$, then we treat the owner of $\{f_i\}$ as the matched record. Finally, the result is returned to the recognition node at the computation layer.

With trial running in several days, we have observed that the intensive networking and disk I/O would appear in the system if too many facial data being sensed within a very short period of time, such as the cameras being deployed in the populated area. The networking I/O is not a problem, because the links, which connect the different layers, are 10 Gbps Ethernet based on the optical fiber. But for maintaining the data consistency simply [21], we only keep a centralized database in the back-end, instead of using the distributed one. This design results in an over-burden database and hurt the S_3 performance.

To relieve the problem, we have deployed memory object caching system, such as Redis, in the computation layer, trying to decouple a portion of work from the busy database. However, the original memory object caching system at the computation layer cannot relieve the load of the back-end database effectively. One main reason for this problem is that the IoT queries from the computation layer are less likely to issue the different queries for the same person during a short time interval. In such a way, too many queries rush into the cold cache at the computation layer, then are sent to the data layer to find their target data, allowing the storage of the data infrastructure to be congested.

Table I describes the current performance of our system, which is further divided into two cases according to the latency of S_3. Namely, this division depends on that the back-end database is congested or not by the IoT queries from the computation layer. The `NORMAL` refers to the case of the number of people in the captured picture is less than 10, while the `INTENSIVE` refers to that the number is greater than or equal to 10. According to the statistics of a random day, 39% frames or pictures fall into the `NORMAL` case (5.4

[2]The format of the key frame is *.jpg and its average size is less than 2MB.

TABLE I
PERFORMANCE OF OUR MOTIVATING EXAMPLE. THE THREE COMPONENTS OF THE TIME FIELD REFER TO THE AVERAGE TIME COST OF A USER QUERY FOR EXECUTING S_1, S_2 AND S_3 RESPECTIVELY. THE UTIL FIELD ALSO INCLUDES THREE PARTS: (1) THE CPU UTILIZATION OF THE COMPUTATION LAYER; (2) THE DISK UTILIZATION OF THE DATA LAYER AND (3) THE NETWORK UTILIZATION BETWEEN THE TWO LAYERS.

Cases	Time (ms)	Util (%)	Cache Hit Rate (%)
`NORMAL`	2014/0574/0873	34/47/25	27.1
`INTENSIVE`	2310/0617/4625	62/99/70	18.9

people in average), while the remaining ones fall into the `INTENSIVE` case (22.4 people in average). The field of `Time` refers to the average time overhead of user query (measured by millisecond), which is broken down into the three stages we just described. `Util` indicates the main hardware utilizations in our system during the sample day, which is also divided into three pieces: (1) the average CPU utilization on the computation layer (denoted by CPU_f); (2) the average disk utilization on the data layer (denoted by $Disk_b$) and (3) the average network utilization between the two layers (denoted by Net_{fb})[3]. In the case of `INTENSIVE`, it can be clearly seen that:

- Compared with the `Normal` case, time for executing S_3 is increased from 873ms to 4,625ms, incurring about 5.3 times delay;
- $Disk_b$ is increased about 2.1 times, due to the busy data layer, which also directly contributes to the multiplied S_3;
- CPU_f is increased from 34% to 62%, but still leaves large room (38%) for being fully saturated;
- The cache hit rate of Redis is even lower, from 27.1% to 18.9%, because the queries for a single face data are hardly repeated many times in a short time window, thereby hurting the cache hit rate of Redis while diminishing the role of caching layer we just deployed.

Therefore, we consider to devise a read-ahead function (termed Seer-MCache) for not only improving the original cache hit rate of Redis, but also utilizing the remaining CPU_f enough in the computation layer. A challenge is that the newly devised function of Redis should keep lightweight or customizable, because it cannot hurt the performance of S_2, which is a computing-intensive process. Meanwhile, we also highly desire that the read-ahead function can be applied to different IoT scenarios, not only in our system.

III. SYSTEM DESIGN

In this section we first present the overview structure of our work. Then, we describe the specific design in detail.

A. Overview

Figure 1 shows the overall design of our work, namely Seer-MCache, on a typical memory object caching system. A split/modularity design is employed by Seer-MCache to add the read-ahead function. Without any intrusive modifications

[3]Currently, we deploy the computation layer and data layer in the different physical machines. Besides, both layers of caching and computation are located in the same host.

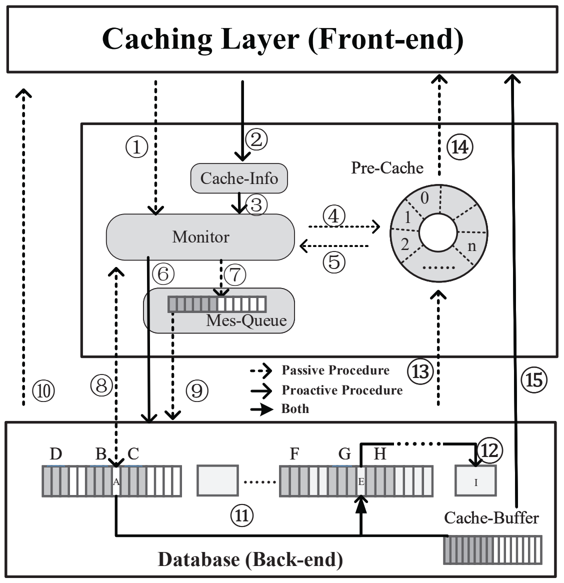


Fig. 1. Overview of Seer-MCache design, which follows the split/modularity style. The light arrows with dot/solid lines refer the passive/proactive read-ahead procedure. The heavy arrow indicates the path that both procedures of passive and proactive need to walk.

on the original systems, Seer-MCache provides a controllable trigger to enable/disable the read-ahead function expediently depending on the specific system load. Moreover, this kind of design allows Seer-MCache to collect the information, which are necessary for running the read-ahead function, from both sides of the front-end and back-end easily.

Seer-MCache mainly includes two policies to trigger the read-ahead behavior: (1) *passive*, which is caused by a cache miss of user query in the caching layer; (2) *proactive*, which is started by the light load of system. To achieve them, Seer-MCache consists of the following five components:

- `Cache-Info` is used to monitor the data, which are accessed frequently in the original memory object caching system;
- `Monitor` is used to analyze the user request (or IoT query) and hotspot data, and then generates a read-ahead request, which will be placed in the `Mes-Queue`;
- `Mes-Queue` is used to store the read-ahead requests generated by the `Monitor`, in which all requests constitute a read-head process;
- `Pre-Cache` is a key-value queue for storing the result of a passive read-ahead procedure, and a data loading process is responsible for populating the read-ahead result into the front-end;
- `Cache-Buffer` is used to store the result of a proactive read-ahead.

B. Passive Read-ahead

A typical passive read-head process of Seer-MCache is described as follows:

(1) The user or IoT application sends a query request, denoted by Req_Q, to the system.

(2) Req_Q first reaches the memory object caching system in front-end. If Req_Q is hit in caching layer, the memory object

caching system will complete this request directly and produce an acknowledge, denoted by ACK, to user or IoT application. Otherwise, goto step (3).

(3) Req_Q is perceived by Monitor, which is referred by ① in figure 1. During this process, Monitor first searches the target data of Req_Q, denoted by A, in Pre-Cache (④ in figure 1). If A exists, Req_Q is satisfied and A will be pushed from Pre-Cache to the original caching layer (⑭). Then, Monitor records the related information about Req_Q to prepare for launching a passive read-ahead behavior (⑤), goto step (5). If A is not in Pre-Cache, Req_Q is transferred to the database in back-end, goto step (4).

(4) Database receives Req_Q and searches A in back-end (⑧). After A was found, ACK is sent to the corresponding user. Subsequently, A is also read to the original caching layer (⑩). Meanwhile, Monitor records the associated information of Req_Q (⑧), in order to start a passive read-head behavior.

(5) A *passive* read-head process P_p, triggered by Req_Q, is started. According to the related passive read-head rule (will be described in section III-B1), Monitor generates a serial of query requests $\{Req_p\}$, which are firstly registered in Mes-Queue (⑦).

(6) Mes-Queue scans the queue from the top down and delivers the requests in $\{Req_p\}$ in turn to the database in back-end (⑨). Then, the target data, denoted by $\{A\}$, to feed $\{Req_p\}$, are read into Pre-Cache (⑬). If any old data in Pre-Cache share the same meta-data (such as the identical storage position in database) with the ones in $\{A\}$, they are substituted by the newer ones.

(7) A daemon process in Pre-Cache continually pushes the data to the original caching layer in front-end (⑭).

During both procedures of (3) and (4), Monitor is required to extract several information from Req_Q, in order to prepare a passive read-ahead procedure in the underlying database. In detail, the information comprises of four fields: 1) the database name which A belongs to; 2) the table names which are involved by A; 3) the primary and foreign keys contained by these tables; 4) the specific row numbers of A involving. By the help of these information, Seer-MCache can define the accurate read-ahead rules for a single read query. The specific descriptions of the rules are presented as follows.

1) Rules for Passive Read-ahead: Seer-MCache mainly utilizes the semantic locality of database to build the passive read-ahead behavior, which is constituted by three specific rules:

- Spatial-Associated (SA): When the ith row of data in a database table D was read, the rows, from the $(i-N)$th row to the $(i+N)$th one, are treated as the read-ahead candidates, in which $(i-N) \geq 0$ and $(i-N) \leq (L-1)$. L is the length of D and N is the window size of SA;
- Temporal-Associated (TA): When a row was read at time t, the rows across all tables, which were just modified or inserted, from time $t-T$ to t, are treated as the read-ahead candidates, where T is the window size of TA;
- Foreign-Associated (FA): When a row of data was read from the secondary table D, the associated primary tables $\{M\}$, which are traversed by referring the foreign keys of D, select a portion of their own data as the read-ahead candidates. The window size of FA has two dimensions denoted by $\{X, Y\}$, where X refers to the number of tables contained in $\{M\}$ while Y refers to the row number of a certain table of $\{M\}$. FA will determine the value of $\{X, Y\}$ randomly from 0 to X and 0 to Y respectively when a passive read-ahead activity is aroused.

It should be noted that Seer-MCache uses the principle of locality only at the level of database semantic (logical level), because modern databases usually establish a strict corresponding relationship between the views of logical and physical for performance [22].

The aforesaid three rules can be applied to Seer-MCache as any combinations. Empty rule means the passive read-ahead is disabled completely, which often happens when the system is very busy. In addition, because each IoT query arriving in database causes a passive read-ahead behavior, intensive query flow not only results in the heavy resource consumption on database, but also may produce many redundant read-ahead data in Pre-Cache, degrading the space utilization. To relieve such issue, Seer-MCache sets a time window W to prevent the passive read-ahead being started frequently: only after W has passed away, Seer-MCache permits a new passive read-ahead behavior to be started.

2) Customization: In current Seer-MCache design, Only DBAs (DataBase Administrators) have the privilege of customizing read-ahead rules. One consideration of this regulation is that the inappropriate read-ahead rules may overburden the server and thus hurt the response times of IoT queries. If the external entities, such as the upper applications, can devise and apply any read-ahead rules, the real-time system would take the risk of being disturbed maliciously.

According to the specific scenario, basically the read-ahead rules leverage the RDBMS-level semantic locality during the process of database logical design. If a new rule is needed to be added into the running Seer-MCache, the system must halt and then apply it. However, each applied rule in Seer-MCache has a flag to indicate its status: disabled or active. A privileged process can change the status during system running. Specifically, applying new read-ahead rule into the system requires the source code to be recompiled and restarted manually. However, triggering the added but disabled rules is automatic. In our system, when the system load is lightweight, one privileged process injects a signal to the specific read-ahead procedure, telling it to use the pre-defined and added rules or part of them. In contrast, when the system is overburden, the privileged process also injects a signal to the specific read-ahead procedure, allowing it to disable one or more applied read-ahead rules. This privileged process is invoked by the DBAs.

To devise the specific rules: (1) the DBAs should learn the read/write ratio of the upper workloads which are talking with the back-end database. The metric can be sampled by using several common tools in the modern operating system and database, such as `iostat`[4] or `innotop`[5]. (2) The DBAs should learn the method to activate or disable one or more rules which have applied into Seer-MCache. Currently, we implement this by a privilege process, which uses the `pipe`[6] to transfer the signal to the main process of Seer-MCache. The DBAs simply need to execute the process to activate or disable the related rules. It should be noted the above two requirements can be devised to an automatic method without the obvious manual labor. Since it is another complicated work, we leave it for future work. (3) The DBAs should be familiar with the logical database design of the upper IoT application or scenario. In doing so, based on the principle of locality at the level of database semantic, the suitable read-ahead rule can be customized and then be applied into Seer-MCache via the former two requirements.

C. Proactive Read-ahead

A consideration on the design of passive read-ahead is constraining its cost on the original system. Therefore, Seer-MCache should control the window sizes of read-ahead rules carefully, even disables one or more to keep its lightweight. However, in doing so the read-ahead coverage is narrowed, which indirectly degrades the cache hit rates of IoT queries in the caching layer. On the other hand, if system is running with the light load, a portion of memory spaces in the front-end are also under-utilized.

To improve these problems, Seer-MCache devises the *proactive* read-ahead behavior. It uses an more aggressive method, but with weak locality, to select the read-ahead candidates and then read them to fill up the caching layer. Due to its coarse-grain, Seer-MCache should run the proactive procedure when system load is somewhat light. To do this, three metrics are taken into account for preparing a proactive read-ahead behavior, namely CPU_f, $Disk_b$ and Net_{fb}. They are already defined in section II. Obviously, when the values of the above three metrics satisfy a set of conditions, denoted by $\{C\}$, Seer-MCache is qualified to launch the proactive read-ahead.

After that, Seer-MCache runs the proactive read-head process on the basis of *Hot-Spot*, which is a technology to monitor the hot data in the memory object caching system (front-end). Specifically, Hot-Spot uses `Cache-Info` in figure 1 to trace the related information of the potential hot data. A table called *Key-Counter* is maintained for this task, where `Key` refers to the key data type corresponding to an item in the original memory object caching system and `Counter` refers to the number of this item being accessed. Whenever an item in the caching layer is read, the associated `Counter` is increased, and then `Cache-Info` will extend the life time of the data. In contrast, if a `Key` is not active for a time, `Cache-Info` will treat it as the eviction candidate, decrease its life time and seek opportunity for replacing it with a new memory page.

Hot-Spot always takes the most recently accessed item in *Key-Counter* as the initialization reference to start the process of proactive read-ahead. A time period T_h is set to enable the timeliness of *Key-Counter*. Namely, the value of `Counter` is legitimated only in a T_h. If a T_h has passed way, `Counter` must be reset. Meanwhile, T_h is also the time interval of Hot-Spot to periodically check if $\{C\}$ are satisfied, where $\{C\}$ are the conditions to decide whether the system is in a lightweight load.

Algorithm 1: Process of proactive read-ahead in Seer-MCache.

```
Input: $A$, which is the most hot item in
       Key-Counter during a $T_h$.
Output: $Flag$, which indicates the process of
        proactive read-ahead is completed or not.
1  ITEM_ARRAY $\{A_p\} = NULL$; /*$\{A_p\}$ is a set of
     items, which are read by a read-ahead procedure.*/
2  ITEM_ARRAY $IA = NULL$; /*IA is a temporary
     array to store a set of items.*/
3  ITEM $temp = NULL$;
4  int $i = 0$;
5  Bool $Flag$ = False;
6  CleanCacheBuffer();/*Emptying Cache-Buffer.*/
7  IA = Readahead($A$);/*Starting a read-ahead procedure
     according to A, the used rules are described in
     section III.A.*/
8  CacheBuffer.Add(IA);/*Putting IA into Cache-Buffer.*/
9  while $i \leq (CacheBuffer.Length)$ do
10     temp = CacheBuffer.Get(i);
11     $\{A_p\}$ = Readahead($temp$);
12     PushingToFrontend($\{A_p\}$);/*Pushing $\{A_p\}$ into the
         original caching layer.*/
13     CacheBuffer.Delete(i);
14     i++;
15 end
16 $Flag$ = True;
17 Return $Flag$;
```

In summary, a typical procedure of proactive read-ahead in Seer-MCache is described as follows:

1) T_h kicks off. Data A, which locates in the front-end, is accessed frequently during T_h, which allows it to become the most hot item in *Key-Counter*, as ② in figure 1 showing;
2) When T_h is up, *Hot-Spot* in `Cache-Info` is invoked to check $\{C\}$. If conditions are satisfied, goto step 3, or else, go back to 1;
3) Seer-MCache first disables both the trigger of passive read-ahead and T_h counting, and then passes the meta-data of A from `Cache-Info` to `Monitor` (③ in figure 1;
4) `Monitor` starts the specific procedure of proactive read-ahead (⑥ in figure 1), which is described in algorithm 1 in detail;
5) The procedure of proactive read-ahead is finished. Seer-MCache re-enables the trigger of passive read-ahead and then go back to step 1.

[4]https://linux.die.net/man/1/iostat
[5]https://github.com/innotop/innotop
[6]https://linux.die.net/man/7/pipe

The `readahead()` function in algorithm 1 corresponds to one invocation of the passive read-ahead procedure. Therefore, a single proactive read-ahead process is composed of a group of passive read-ahead procedures, which are launched simultaneously. To allow the concurrent read-head procedures, we need to clear the time window W at the beginning of each proactive read-ahead behavior. After finished, we set it again. Due to the light load of the current system, multiple read-ahead procedures did not overburden the system.

Intuitively, multiple concurrent read-ahead procedures may conflict each other. In our proposed system, the conflicts between the applied rules mainly lead to two results: (1) a portion of data are prefetched by a rule, and then they are read again by another rules, namely, the redundant data occurs during the read-ahead procedures; (2) correspondingly the read overhead is amplified, since several read operations in the back-end are unnecessary. To maintain the procedure of read-ahead simple, currently, we did not tailor-made a technique to solve this issue. Instead, Seer-MCache has a module called `Pre-Cache`, which stores the data have been prefetched. When the specific read procedure put data into this module, the redundant item(s) will be checked and then be eliminated automatically.

On the other hand, a proactive read-ahead process may be terminated abnormally, such as power outage. In this case, there are several items stayed in `Cache-Buffer`, which have lost the locality to their parent item such as A in algorithm 1. Therefore, the line 6 in algorithm 1 shows a clean procedure in `Cache-Buffer` to delete all old items before beginning the new proactive read-ahead process.

IV. Implementation

We implement Seer-MCache in a `Redis` (front-end) and `MySQL` (back-end) server system. We split Seer-MCache into two parts, placing them on the front-end and back-end respectively.

The front-end of Seer-MCache includes only `Cache-Info` component, which is mainly for tracing the hottest item of the original caching layer by maintaining the `Key-Counter` table. `Cache-Info` uses the logical-level semantic of database to name the read-ahead items in Redis. For example, a row of data R will be named T_i, where T refers the table name of R and i indicates the identifier (ID) of R in T. In such a way, a IoT user query can find its target data in the front-end by parsing T_i.

Besides, in order to monitor the access frequency of items in the caching layer, we leverage `Jedis`[7], a client library in Java for `Redis`, to acquire the related information inside Redis.

Finally, `Cache-Info` is required to collect the system information of front-end, such as CPU_f which is described in section III-C, for preparing a proactive read-ahead process. To implement this, we employ `Sigar`[8], which is a library providing interface to low-level information on computer hardware and operating system activity.

[7]https://github.com/xetorthio/jedis
[8]https://github.com/hyperic/sigar

We deploy `Monitor`, `Mes-Queue`, `Pre-Cache` and `Cache-Buffer` in the same physical machine with the back-end database system, for reducing the unnecessary data across the network thus providing a better interactive performance among them.

Except for intercepting the missed user query from front-end, an additional function of `Monitor` is to collect the low-level system information of back-end by using `Sigar`, namely $Disk_b$ described in section III-C. On the other hand, since a database can deploy one or more caching layers in multiple front-ends, Seer-MCache also allows `Monitor` to collect the global network utilization between front-ends and back-end, namely Net_{fb} described in section III-C.

V. Evaluation

A. Experimental Setup

Two separated physical machines act as the roles of front-end and back-end respectively. Both of them have dual Intel Xeon CPU E5-2620 2.1GHz with hyper-threading enabled, 96GB DDR4 RAM, 2TB TOSHIBA hard disk drive and Intel Ethernet connection I217-LM (Gbit/s). Two machines are connected via a 10Gb switch.

CentOS 7 (1511) with Linux kernel 4.1.25 are used as their operating systems. On the front-end side, `Redis`-3.2.11 is installed while `MySQL`-5.6.24 is deployed on the back-end side. Because many parameters in Seer-MCache are required to be determined in actual operation, Table II summaries their values which are used through our experiments. It should be noted that most of them are given according to our practical experience, however, it does not follow that they are the optimal ones in all cases. We will investigate this issue in our future work.

We use three kinds of evaluations to test Seer-MCache: (1) synthetic benchmark; (2) realistic benchmark and (3) trace replay. The first one further includes three kinds of workloads: (1) write-intensive (read operations account for about 20%); (2) read-write mixture (read accounts for 50%) and (3) read-intensive (read accounts for 80%). The second one, realistic workload, uses `OLTPBench` to simulate three realistic workloads, namely `TPCC`, `TATP` and `SmallBank`. Finally, a trace, which is collected from table I in section II, is replayed to verify the effectiveness of Seer-MCache in a real scenario. Before each test, we reboot both front-tend and back-end to exclude the interference from the warm cache.

As we will describe in section VI, the prefetchers can be implemented at hardware-level, block-level and application-level. Because to the best of our knowledges there is no suitable counterpart at the application-level, in this paper we compare our work with the original system, which deploys the memory object caching system without the application-level prefetcher. But we activated the existed prefetchers of the underlying file system (block-level), database storage engine (block-level) and hardware during all experimental configurations.

B. Synthetic Workloads

Initially, we create one million records in total at the back-end side, which are scattered across 22 tables uniformly. Each

TABLE II
DESCRIPTIONS OF SEER-MCACHE PARAMETERS, WHICH ARE USED IN OUR EXPERIMENTS.

Name	Description
N	Used in the window size of spatial-associated rule (SA).The value is set to 50 rows.
T	Used in the time interval of temporal-associated rule (TA). The value is set to 3 seconds.
X	Used in the foreign-Associated (FA) rule, indicating the number of primary tables which are associated with a secondary table. The value is set to 5 tables.
Y	Used in the foreign-Associated (FA) rule, indicating the row number of a certain primary table which will be read-ahead. The value is set to 10 rows.
W	A time window permitting only one passive read-ahead process to be launched. The value is set to 500ms.
CPU_f	CPU utilization of the Redis process in the front-end. If $CPU_f \geq 50\%$, the trigger of proactive read-head process is disabled.
$Disk_b$	**Idle** disk utilization of the back-end. If $CPU_b \leq 50\%$, the trigger of proactive read-head process is disabled.
Net_{fb}	Network utilization between the font-end and back-end. If $Net_{fb} \geq 75\%$, the trigger of proactive read-head process is disabled. We sample this metric in the back-end side.
T_h	Expiry time of the counter value of `Key-Counter` table in `Cache-Info`. This parameter also indicates the time window that allows only one proactive read-ahead process to be launched. The value is set to 3 minutes.

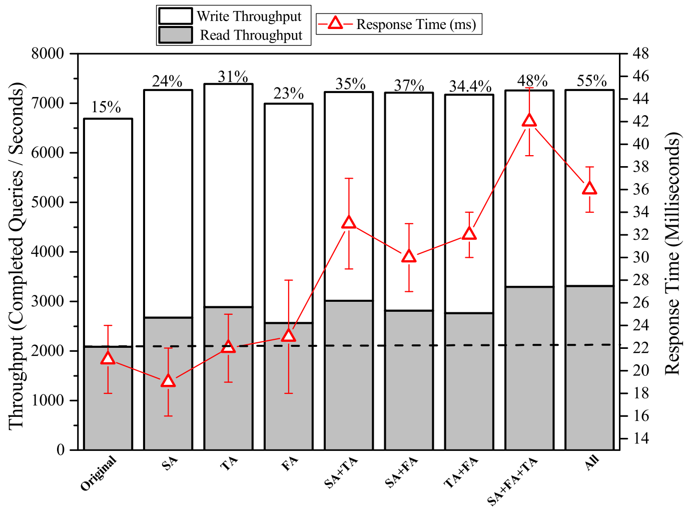


Fig. 2. Results under write-intensive workload (the read user query accounts for about 20%). The numeric label on top of each bar is the cache hit rate of user read query on the font-end. "Original" refers to the original system, "SA" means the passive read-ahead process with spatial associated rule, "TA" refers to the temporal associated rule, "FA" refers to the foreign associated rule, "All" means the combination of three passive read-ahead rules and the proactive read-head process. Each value is the mean of 5 trials.

table at least has a foreign key associated with the other table(s). Then we write a program to deliver three different workloads to the back-end. The program executes an infinite loop to issue the random user queries across all 22 tables, but with the relatively high priority ($w = 0.3$) to issue them into the same table, which is just accessed by the prior user query, or into its primary tables ($w = 0.3$). The remainder weights ($w = 0.4$) are distributed uniformly throughout the other tables. During this procedure, the testing program issues the intensive queries as the uninterrupted and continuous flow. Each test procedure maintains at least 30 minutes and then we terminate it manually.

1) Write-intensive Workload: Figure 2 shows the result of write-intensive workload. It can be seen that the single read-ahead rule, namely `SA`, `TA` and `FA`, can improve the cache hit rate of front-end from 15% to 24%, 31% and 23% respectively. `TA` performs better because it reads-ahead the rows which have just be modified or created under this write-intensive workload.

As for performance, `SA`, `TA` and `FA` outperform the original system about 28%, 38.4% and 22.9% respectively on read throughput. To the write, they surprisingly incur 0.2%, 2% and 4% overhead individually versus the original one. We conjecture the reason is that the database system in back-end is required to serve more disk I/O requests brought by the additional read-head processes, which inevitably enforce the resource contention on the shared disk device to the write queries.

Rules applied in pair, namely `SA+TA`, `SA+FA` and `TA+FA`, achieve about 133%, 147% and 129% improvements respectively on the cache hit rate of front-end. Correspondingly, their read throughputs obtain 44.3%, 34.9% and 32.5% performance benefit respectively. But the write throughputs respectively show 8.4%, 4.4% and 4.2% performance penalty against the original one due to the amplified read-ahead activity. On the other hand, the response time of user query, is increased against the single read-ahead rule, from the average 21.3ms to 31.7ms. We attribute this cost to the write portion of this workload, where most of user queries (write operations) are delayed due to the additional read-ahead process on the shared disk device.

The triple rule, namely `SA+FA+TA`, reaps the roughly 220% benefit on the cache hit rate of front-end versus the original one, from 15% to 48%, alongside with the 57.8% improvement of read throughput, from 2,087 queries/second to 3,294 queries/second, but it comes at a relatively large cost of write throughput, from 4,601 to 3,965, incurring about 13.8% performance penalty, because the data amount of read-ahead on `SA+FA+TA` is further increased. For the response time of user query, `SA+FA+TA` shows 42ms on average, which is increased about 33% than the rules in pair.

Finally, the proactive read-ahead process plus the aforesaid triple rules of passive read-ahead, namely `All` in figure 2, shows the highest cache hit rate in the font-end. But compared with `SA+FA+TA`, the performance benefit on throughput is trivial because the number of read user query is limited by the write-intensive workload. However, `All` still improves the response time of user query about 14.3% against `SA+FA+TA`, due to the higher cache hit rate of front-end.

2) Write-Read Mixture Workload: Figure 3 shows the result of write-read mixture workload. Due to the proportion of read query increasing, from 20% to 50%, the overall performances across all cases are improved versus the write-intensive workload.

`SA`, `TA` and `FA` improve the cache hit rate of front-end from 19% to 32%, 35% and 31% respectively against the original

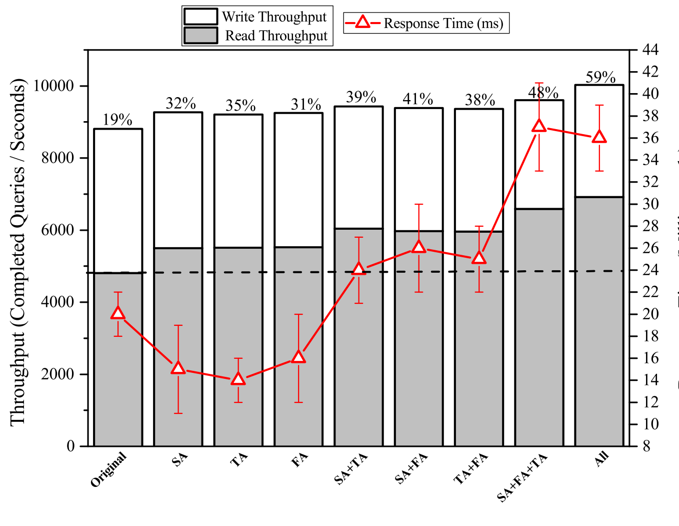


Fig. 3. Results under write-read mixture workload (the read user query accounts for about 50%). The numeric label on top of each bar is the cache hit rate of user read query on the font-end. Each value is the mean of 5 trials.

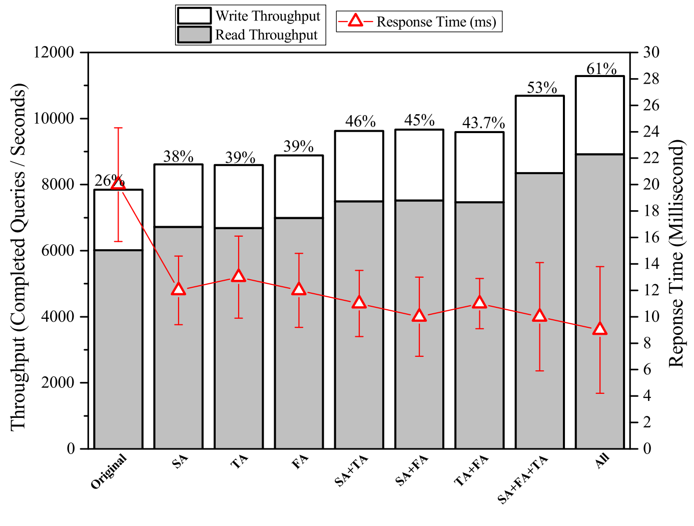


Fig. 4. Results under read-intensive workload (the read user query accounts for about 80%). The numeric label on top of each bar is the cache hit rate of user read query on the font-end. Each value is the mean of 5 trials.

system. The associated read throughputs are also increased about 14.2%, 14.6% and 14.8% respectively. In contrast, the write throughput incurs about 6.63% performance penalty on average in `SA`, `TA` and `FA`. One main reason contributes this cost: read-ahead requests contend the resource of shared disk device with the write ones in the back-end. All three single rules improve the response time of original system about 6.6% on average. The benefit mainly derives from the fact that more read queries are hit in the front-end due to the passive read-ahead behaviors, instead of finding the data in back-end.

`SA+TA`, `SA+FA` and `TA+FA` obtain the better cache hit rate of front-end for their twice passive read-ahead rules, from the original 19% to 39.3% on average, but trade with the write throughput about 14.8% on average versus the original one. Moreover, the average response times of all user queries are delayed from 20ms to 25ms on average. The performance penalty is caused by the write queries, which must touch the disk device of back-end (for maintain the integrity of database transaction) and thus be involved with the concurrent read-ahead disk I/O.

Similarly, compared with the original system, `SA+FA+TA` achieves 152.6% improvement over the cache hit rate of front-end. As a result, the read throughput is also enhanced about 36.9%. However, the amplified read-ahead process in back-end punishes the write throughput about 24.5%, from 3,995 to 3016. Meanwhile, the mean response times of all user queries are affected negatively from 20ms to 37ms.

With the proactive read-ahead process, `SA+FA+TA` reaps the largest cache hit rate of front-end and the read throughput. It should be noted that these benefits almost come at no cost of write throughput by comparing with the single `SA+FA+TA`, even improve its response times of user queries lightly, from 37ms to 36ms. The reason is that 59% read queries are served in the front-end. They do not trigger the corresponding passive read-ahead processes in the back-end, thus reducing the concurrent disk I/O with the write ones. This result justifies that we only start the proactive read-ahead process when system is in a light load while demonstrating the rationality of interleaving the passive read-ahead and the proactive one.

3) Read-intensive Workload: Figure 4 shows the results of read-intensive workload. In general, the overall performances across all cases are improved due to the read queries accounting for a large proportion in this workload. On the cache hit rate of front-end, `SA`, `TA` and `FA` respectively outperform the original system about 46.2%, 50% and 50%. As a result, the read throughputs are improved about 13.1% on average. Meanwhile, the response times of all user queries are also reduced versus the original one, from 20ms to 12.3ms on average, mainly due to the large proportion of read queries. Write throughput in this workload is unaffected, even be improved from 1,833 to 1,895. The reason is that the performance benefit brought by the read-ahead process offsets the associated overhead under the read-intensive workload.

`SA+TA`, `SA+FA` and `TA+FA` further increase the cache hit rate of front-end to 46%, 45% and 43.7% respectively. Both read throughputs and response times of user queries are improved about 24.7% and 46.7% on average. Write throughputs are also benefited from the relatively high cache hit rate of front-end, because write queries are handled more quickly with the read ones being hit in the front-end, instead of forming the concurrent disk I/Os in the back-end.

`SA+FA+TA` shows 53% cache hit rate of font-end, with about 100% improvement over the original system. Its read/write throughput is improved about 38.9%/27.9%. The response times of all user queries are also reduced by approximately 50%. By combining the proactive read-ahead process, `SA+FA+TA` improves the cache hit rate from 53% to 61%. Both throughputs of read and write are increased about 48.4% and 29.2 % respectively, with a 55% reduction on response time of user query.

4) Summary: It should be noted that the average response time of each user query is delayed by Seer-MCache in figure 2 and 3, in which the cases of stacked read-ahead rules are more obvious. Instead, in the case of read-intensive workload,

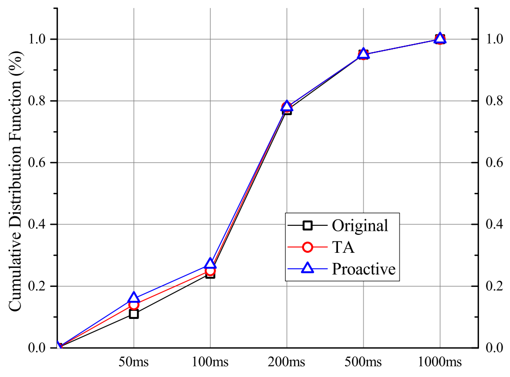


Fig. 5. CDF plot of query latencies on the `TPC-C` workload. ”Original” refers to the original system, ”TA” means the temporal-associated rule and ”Proactive” refers to the temporal-associated rule plus the proactive read-ahead process.

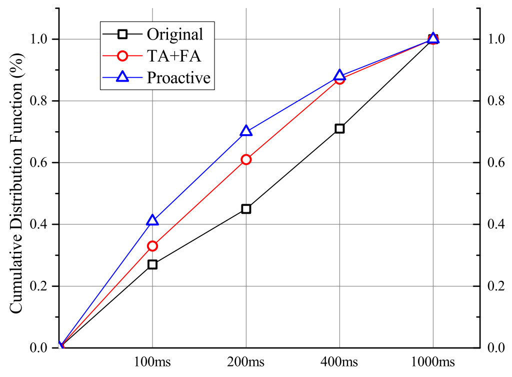


Fig. 6. CDF plot of query latencies on the `TATP` workload. ”Original” refers to the original system, ”TA+FA” means the combination rule of temporal-associated and foreign-associated while ”Proactive” refers to `TA+FA` plus the proactive read-ahead process.

Seer-MCache provides the original system with the improved response time of user query, as figure 4 showing. The reason is that activating the read-ahead technologies in all systems bring the associated overhead inevitably, since in a server the extra resources are required to be allocated to serve the extra read requests (*read amplification*). If the server is in busy, such read-ahead operations further burden the server load, thus leads to a resource contention among these requests (including the original read/write request and the associated read-head requests). Generally, write operations are more inclined to incur this kind of performance interference, because most of them are synchronous and thus need to be persisted synchronously in the slow disk device. It is not like the read ones, which have the chance to be finished in advance in the memory, such as hitting in the page cache or the caching layer (Seer-MCache) of the front-end in our system.

In summary, based on the above observations and analyses, we make the following inferences about Seer-MCache:

1) In the case of read-intensive workload, `SA`, `TA` and `FA` are effective to improve the cache hit rate of front-end, as well as the read throughput.
2) `TA` is more suitable for serving the *read-after-write* workload, due to focusing on the data just be modified or created.
3) `SA`, `TA` and `FA` have cost itself on delivering the corresponding read-ahead process. Specifically, as the proportion of write query increased, this kind of cost may interference the performance of write queries in the back-end.
4) Higher cache hit rate of front-end leads to fewer passive read-ahead process, because more read user queries are satisfied directly in the caching layer of front-end, rather than going to the database system of back-end.
5) Proactive read-ahead process is orthogonal to the passive read-ahead process during system running and it can effectively benefit the cache hit rate of front-end.

C. Realistic Workloads

In this subsection, we test Seer-MCache with three realistic workloads. According to the inferences we made in section V-B4, we always activate the trigger of proactive read-ahead. In addition, to provide the boot opportunity of proactive read-ahead process, we random inject five pause events in the client side during the testing procedure, lowering the system load. Each experiment maintains about 1 hour and then we end it manually.

1) TPC-C: `TPCC-MySQL`[9] is used in client to issue the `TPC-C` workload[10]. We first use `tpcc_load` command to create one test database in MySQL sever of the back-end system, which includes 8 warehouses. Then `tpcc_start` simulates 32 concurrent user connections to the `InnoDB` engine of MySQL server. This procedure contains 9 tables and 92% transactions are user write queries. Due to its write-intensive and the inferences in section V-B4, we only select `TA` as the single passive read-ahead rule. The unit of measurement is tpmC (completed transactions per minute). The results show that `TA` improves the original system only about 3%. Benefit brought by the proactive read-ahead process is also minor, only achieving 6.7% improvement. We use `SA+FA+TA` in this workload for checking the amplified cost of excessive passive read-ahead rules, the score is 169 (tpmC), incurring about 19.5% performance penalty. This verifies the third inference which is described in section V-B4.

Figure 5 shows the CDF plot of query latencies with `TPC-C` workload. `TA` increases the ratio of user queries, which have the lower latency ($\leq$50ms), from 11% to 14%. Meanwhile, `TA+Proactive` increases this ratio from 11% to 16%.

2) TATP: `TATP` (Telecommunication Application Transaction Processing)[11] workload consists of four tables and the read-only user query accounts for approximately 40%. Since three tables have the foreign keys, we apply `TA+FA` to the passive read-ahead rule. We set the test duration to 1 hour with 10 terminals. The unit of measurement is the completed transactions per second. The results show that `TA+FA` outperforms the original system about 38% (from 92 to 127), and the proactive read-ahead process further improve

[9]https://github.com/Percona-Lab/tpcc-mysql
[10]http://www.tpc.org/tpcc/
[11]http://tatpbenchmark.sourceforge.net/

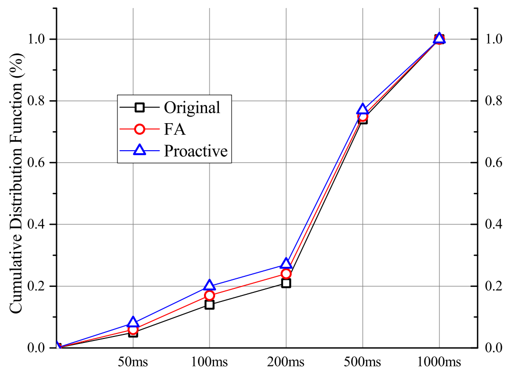


Fig. 7. CDF plot of query latencies on the `SmallBank` workload. "Original" refers to the original system, "FA" means the foreign-associated rule while "Proactive" refers to `FA` plus the proactive read-ahead process.

it from 127 to 148, obtaining 61% improvement versus the original one.

Figure 6 shows the CDF plot of query latencies under `TATP` workload. `TA+FA` increase the ratio of small latency (≤100ms) from 27% to 33% versus the original system. By combining the proactive read-ahead process, this ratio is improved up to 41%. On the other hand, the ratio of median latency (≤200ms) is enlarged from 45% to 61%. The proactive read-ahead process further improves this ratio up to 70%. Mover over, the ratio of large latency (≥400ms) is controlled in about 12.5% with Seer-MCache, against the 29% in original system.

3) SmallBank: `SmallBank` workload issues the write operations and simple read on the accounts of customers, consisting of 3 tables and the ratio of read user queries is about 15%. Because of write-intensive and two tables creating the foreign keys, we only apply `FA` to the passive read-ahead rule. The testing time is set to 1 hour and database is connected by 5 terminals. The measurement unit is the completed user queries per second. The results show that `FA` improves the original system about 7.8%. By triggering the proactive read-ahead process, Seer-MCache reaps 16.7% performance benefit.

Figure 7 shows the CDF plot of query latencies under `SmallBank` workload. It can be seen that `FA` has 17% ratio of user queries with the small latency (≤100ms), versus the 14% in case of original system. An additional triggering of the proactive read-ahead process can further improve this ratio up to 20%. For the ratio with the median latency (≤200ms), the original system, `FA` and `FA` plus the proactive read-ahead process show 21%, 24% and 27% respectively.

Not like the experiments in section V-B, in which the testing program issues the synthetic user queries without any intentional intervals, there is a random delay (produced by `Poisson` random timer) between each of two consecutive queries to simulate the think time during the above `TPC-C`, `TATP` and `SmallBank` tests. It is indicated that these realistic workloads give the database a short and random time window to handle with the pending user requests, thus relieving the response times of user queries. As a result, although TPC-C and SmallBank are write-intensive, the average response time of each user query is still improved about 17% (TPC-C), 25.4% (TATP) and 21.9% (SmallBank) respectively.

D. Trace Replay

We collect the trace from table I of section II to verify the effectiveness of Seer-MCache in our large-scale real-time face recognition system. The system issues the read-intensive query to the backing database. Each query from the computation layer is strictly replayed according to its creation time, query type and target items in the data layer. Because the database logical design of our large-scale real-time face recognition system organizes the personal documents by following their residential addresses, for example, the people living a community share the same table in the backing database system, here we first apply `SA` into the read-head rules. In addition, a person's record contains a group of foreign keys, which refer to the people (records) have certain relationships with him, so we also apply `FA` into our system.

The results show that the cache hit rate of the caching layer is increased from 27.1% to 42.2% in the case of `NORMAL`, from 18.9% to 30.8% in the case of `INTENSIVE`, obtaining about 51.3% and 63% improvement respectively. Meanwhile, the time overhead for running S_3 is relieved about 9% under the `NORMAL` case, from 873ms to 794ms, and about 63.9%, from 4,625ms to 1,671ms under the `INTENSIVE` case. Correspondingly, the hardware utilizations, namely CPU_f, $Disk_b$, Net_{fb}, are varied from 34/47/25 to 38/61/39 in the case of `NORMAL`, and varied from 62/99/70 to 81/79/50 in the case of `INTENSIVE`. In summary, Seer-MCache improves the issue effectively which we presented in section II. The utilization of CPU resource is also improved.

VI. Related Work

From the hierarchy perspective of computer architecture, the related work about prefetching mechanisms of system caching can be divided into three layers: (1) application-level; (2) block-level and (3) hardware-level.

Application-level rules of read-ahead are often specific to a kind of applications, in which the rules are built around the high-level "semantic" locality. For example, in Seer-MCache a group of logically-adjacent rows in the table of a typical RDBMS can be read together into the memory after a user query targeted on one of them, although their corresponding underlying physical positions may be scattered to each other. The concept of "rows" here depends on the application, namely RDBMS, irrespective of the host operating system and hardware. Since this kind of rules are more closer to applications, the prefetchers can bring more precise data while shielding the physical limitations. However, the rules may be different in various upper applications, which indicates one application-level prefetcher cannot be directly used by another.

Block-level rules usually locate under the file system, therefore, the object of prefetching is logical block number (LBN) of the disk. Since a group of adjacent LBNs is not strictly corresponding to a set of adjacent PBNs (physical block number), like the application-level methods, this kind of prefetcher can also shield the underlying hardware. For example, in 2008 Wu et al. present a read-ahead framework for the block layer of Linux 2.6 kernel [23]. This work not only works on the DAS (Direct-attached storage), but also has a role on the NFS (Network File System). Meanwhile, different applications, which run on the same operating system, can be all benefited from the single block-level prefetcher.

Hardware-level rules are directly applied into the physical device, such as the CPU or memory controller. Due to its low-level, the object of prefetching usually is memory address, which refers the data or instructions. For example, in 2017 Bhattacharjee proposes TEMPO [24], a hardware improvement on the page table path and memory controller, by prefetching the hot references from memory into the row buffer and the LLC (last level cache) of CPU. This kind of rules are required to modify the hardware, but its effectiveness covers all upper operating systems and applications.

All rules of read-ahead, regardless of its level, bring the overhead. It not only consumes the CPU cycles and memory bandwidth, but also pushes the "wrong" data into the caching layer with a certain probability. Generally, with the high level, the rules present high cache hit rate, but trade-off the large cost. This is often caused by layers of abstraction that software introduces [25]. Therefore, the application-level prefetching mechanisms should carefully verify their associated costs, which is what we do in this paper. Rules in different levels can be overlapped for each other, but the overhead is also increased. During all evaluations of this paper, Seer-MCache is deployed on the modern hardware and mainstream operating system. All prefetchers of the block-level and hardware-level are activated by default.

In 2011, Tang proposed FVD [26], an improved virtual machine (VM) image format for QEMU[12]. It includes an adaptive prefetching strategy at the block-level, which uses resource idle time to copy from NAS (Network-Attached Storage) to DAS (Direct-Attached Storage) the image data that have not been read or written by the VM. The proactive read-ahead process of Seer-MCache shares the core idea with FVD, but the differences between them are: (1) FVD uses the read throughput of VM as metric to detect resource contention while Seer-MCache takes the disk utilization of back-end into account; (2) FVD uses a complex leaky bucket algorithm to decide if a resource contention has real occurred but Seer-MCache just compares the disk utilization directly with a threshold for simplicity; (3) Seer-MCache has a serial of passive read-ahead rules while FVD only has the proactive read-ahead one.

In 2013, Saad et al. proposed a data prefetcher to improve the performance of BoT (Bag of Tasks) and DAG (computational work-flow) applications [27]. This work very focuses on the Desktop Grid environment and is application-level, in which the data are often obtained from the remote nodes through wide area networking. Due to the collaboration between worker node and master node (the data source), the data prefetcher is required to deploy a daemon in each involved data, finally shows a peer-to-peer networking. Compared with Seer-MCache, this work is devised for the desktop grid, without a centralized backing database system. Therefore, it cannot be migrated directly to our scenario. Moreover, the associated overhead brought by the prefetcher is also not given.

[12]https://github.com/qemu/qemu

Jiang et al. in 2013 proposed DiskSeen [19], a block-level prefetch policy, to improve the sequentiality of disk accesses and overall prefetching performance. DiskSeen has a mechanism to correct the mis-prefetching behavior, on the view of per-workload, to reduce the associated performance loss. The main difference between DiskSeen and Seer-MCache is that: Seer-MCache is an application-level read-ahead system while DiskSeen is a block-level one. Therefore, both of them can be complementary to each other.

In 2015, Zhu et al. proposed several practical prefetching techniques in the memory controller for improving the performance of in-memory key-value store [28]. The main difference between this work and Seer-MCache is: Zhu's work focuses on the hardware-level while Seer-MCache works in the application-level. It indicates that the object of prefetching in [28] is CPU instruction. Instead, Seer-MCache only targets on the high-level semantic object in the database, such as one row in a data table. Moreover, Zhu's work only verifies its effectiveness in a simulated environment, but Seer-MCache has experienced a test procedure under the realistic workloads.

In 2017, Li et al proposed a layered caching architecture [29] to improve the performance of real time big data management. This work belongs to the boundary between hardware-level and block-level. At the block-level, it applies a K-means clustering algorithm to support the prefetching mechanism. At the hardware-level, it uses a small portion of the DRAM and the NAND-Flash memory space to support its layered caching architecture. Not like Seer-MCache, this work requires the hardware to provide the hybrid storage, allowing its prefetching procedure to be run efficiently. In contrast, Seer-MCache can run on all modern hardware, presenting better generality.

Almost of all works on the IoT prefetcher are improving the read performance of the upper IoT devices. However, several works are proposed to improve the write one. A typical example is Triple-L [30], an improved data infrastructure for IoT devices writing. It leverages the local storage to cache the updated replicas, thus avoiding the ordinary network path between clients and servers. However, Triple-L only improves the write direction of IoT queries. In contrast, Seer-MCache presented in this paper is a good complement for Triple-L, since it mainly improves the read performance of IoT queries.

VII. Conclusion

In the era of big data, IoT devices perceive massive data from the complex physical world and then often push them to the data infrastructure of cyberspace, to meet the processing requirements of a specific scenario. During this procedure, the database usually incurs excessive load due to the massive user queries from IoT devices. As a result, the querying performance of IoT applications may be hurt, such as the large-scale real-time face recognition system we referred in section II. To improve this problem, we propose Seer-MCache for the ordinary memory object caching system, devising a smart read-ahead (prefetch) function to enhance the cache hit rate of IoT queries at the caching (computation) layer. In such a way, a portion of IoT queries can be served outside the data infrastructure, thus relieving the system load of back-end. The read-ahead function includes several rules, which are built from the principle of locality. They are highly customizable. Therefore, Seer-MCache can apply them into the real system as any combinations according to the characteristics of IoT applications and the load of data infrastructure. We implement Seer-MCache in the Redis and MySQL systems. Extensive experiments are conducted to verify the advantages of Seer-MCache. We find that Seer-MCache is effective to improve the performance of IoT applications while brings the mild and controllable cost.

Acknowledgment

We would like to thank Mr.Yijie Zhong for his wonderful work to improve this paper. This work was funded by the National Natural Science Foundation of China under grant number 61502180 and 61772211, by the Natural Science Foundation of Guangdong Province, China under grant number 2017A030303074 and 2016A030313441, by the Pearl River S&T Nova Program of Guangzhou under grant number 201710010189, by JSPS KAKENHI Grant Number JP16K00117, JP15K15976, and KDDI Foundation. Mianxiong Dong is the corresponding author.

References

[1] K. Ueta, X. Xue, Y. Nakamoto, and S. Murakami, "A Distributed Graph Database for the Data Management of IoT Systems," in *2016 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, Dec 2016, pp. 299–304.

[2] L. Jiang, L. D. Xu, H. Cai, Z. Jiang, F. Bu, and B. Xu, "An IoT-Oriented Data Storage Framework in Cloud Computing Platform," *IEEE Transactions on Industrial Informatics*, vol. 10, no. 2, pp. 1443–1451, May 2014.

[3] T. Yu, X. Wang, and A. Shami, "Recursive Principal Component Analysis-Based Data Outlier Detection and Sensor Data Aggregation in IoT Systems," *IEEE Internet of Things Journal*, vol. 4, no. 6, pp. 2207–2216, Dec 2017.

[4] T. Kumrai, K. Ota, M. Dong, J. Kishigami, and D. K. Sung, "Multiobjective Optimization in Cloud Brokering Systems for Connected Internet of Things," *IEEE Internet of Things Journal*, vol. 4, no. 2, pp. 404–413, April 2017.

[5] S. Rautmare and D. M. Bhalerao, "MySQL and NoSQL database comparison for IoT application," in *2016 IEEE International Conference on Advances in Computer Applications (ICACA)*, Oct 2016, pp. 235–238.

[6] Z. Liu, C. Zhang, M. Dong, B. Gu, Y. Ji, and Y. Tanaka, "Markov-Decision-Process-Assisted Consumer Scheduling in a Networked Smart Grid," *IEEE Access*, vol. 5, pp. 2448–2458, 2017.

[7] M. K. Aguilera, J. B. Leners, and M. Walfish, "Yesquel: Scalable SQL Storage for Web Applications," in *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP 2015)*. New York, NY, USA: ACM, 2015, pp. 245–262.

[8] Y. Nakamura, H. Suwa, Y. Arakawa, H. Yamaguchi, and K. Yasumoto, "Middleware for Proximity Distributed Real-Time Processing of IoT Data Flows," in *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS 2016)*, June 2016, pp. 771–772.

[9] E. Zamanian, C. Binnig, T. Harris, and T. Kraska, "The End of a Myth: Distributed Transactions Can Scale," *Proc. VLDB Endow.*, vol. 10, no. 6, pp. 685–696, Feb. 2017.

[10] R. Ramakrishnan, B. Sridharan, J. R. Douceur, P. Kasturi, B. Krishnamachari-Sampath, K. Krishnamoorthy, P. Li, M. Manu, S. Michaylov, R. Ramos, N. Sharman, Z. Xu, Y. Barakat, C. Douglas, R. Draves, S. S. Naidu, S. Shastry, A. Sikaria, S. Sun, and R. Venkatesan, "Azure Data Lake Store: A Hyperscale Distributed File Service for Big Data Analytics," in *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD 2017)*. New York, NY, USA: ACM, 2017, pp. 51–63.

[11] I. Abdelaziz, R. Harbi, Z. Khayyat, and P. Kalnis, "A Survey and Experimental Comparison of Distributed SPARQL Engines for Very Large RDF Data," *Proc. VLDB Endow.*, vol. 10, no. 13, pp. 2049–2060, Sep. 2017.

[12] Y. Cheng, A. Gupta, and A. R. Butt, "An In-memory Object Caching Framework with Adaptive Load Balancing," in *Proceedings of the Tenth European Conference on Computer Systems (Eurosys 2015)*. New York, NY, USA: ACM, 2015, pp. 4:1–4:16.

[13] M. Pilman, K. Bocksrocker, L. Braun, R. Marroquín, and D. Kossmann, "Fast Scans on Key-value Stores," *Proc. VLDB Endow.*, vol. 10, no. 11, pp. 1526–1537, Aug. 2017.

[14] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang, "KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC," in *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP 2017)*. New York, NY, USA: ACM, 2017, pp. 137–152.

[15] W. C. Chen, Y. H. Chen, C. L. Wu, and L. C. Fu, "An Efficient Data Storage Method of NoSQL Database for HEM Mobile Applications in IoT," in *2014 IEEE International Conference on Internet of Things (iThings), and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom)*, Sept 2014, pp. 336–339.

[16] A. Mahgoub, P. Wood, S. Ganesh, S. Mitra, W. Gerlach, T. Harrison, F. Meyer, A. Grama, S. Bagchi, and S. Chaterji, "Rafiki: A Middleware for Parameter Tuning of NoSQL Datastores for Dynamic Metagenomics Workloads," in *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference (Middleware 2017)*. New York, NY, USA: ACM, 2017, pp. 28–40.

[17] J. Lee, H. Kim, and R. Vuduc, "When Prefetching Works, When It Doesn&Rsquo;T, and Why," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 1, pp. 2:1–2:29, Mar. 2012.

[18] V. Seshadri, S. Yedkar, H. Xin, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "Mitigating Prefetcher-Caused Pollution Using Informed Caching Policies for Prefetched Blocks," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 4, pp. 51:1–51:22, Jan. 2015.

[19] S. Jiang, X. Ding, Y. Xu, and K. Davis, "A Prefetching Scheme Exploiting Both Data Layout and Access History on Disk," *Trans. Storage*, vol. 9, no. 3, pp. 10:1–10:23, Aug. 2013.

[20] J. Calic and E. Izuierdo, "Efficient Key-Frame Extraction and Video Analysis," in *Proceedings. International Conference on Information Technology: Coding and Computing*, April 2002, pp. 28–33.

[21] C. Li, N. Preguiça, and R. Rodrigues, "Fine-grained Consistency for Geo-Replicated Systems," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, 2018, pp. 359–372.

[22] I. Elghandour, A. Aboulnaga, D. C. Zilio, and C. Zuzarte, "Recommending XML Physical Designs for XML Databases," *The VLDB Journal*, vol. 22, no. 4, pp. 447–470, Aug. 2013.

[23] W. Fengguang, X. Hongsheng, and X. Chenfeng, "On the Design of a New Linux Readahead Framework," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 5, pp. 75–84, Jul. 2008.

[24] A. Bhattacharjee, "Translation-Triggered Prefetching," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2017)*. New York, NY, USA: ACM, 2017, pp. 63–76.

[25] D. Le, H. Huang, and H. Wang, "Understanding Performance Implications of Nested File Systems in a Virtualized Environment," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST 2012)*. Berkeley, CA, USA: USENIX Association, 2012.

[26] C. Tang, "FVD: A High-Performance Virtual Machine Image Format for Cloud," in *Proceedings of the 2011 USENIX conference on USENIX Annual Technical Conference (USENIX ATC 2011)*. Berkeley, CA, USA: USENIX Association, 2011.

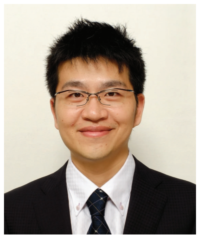
[27] W. Saad, H. Abbes, C. Crin, and M. Jemni, "A Data Prefetching Model for Desktop Grids and the Condor Use Case," in *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, July 2013, pp. 1065–1072.

[28] P. Zhu, G. Sun, P. Wang, and M. Chen, "Improving Memory Access Performance of In-Memory Key-Value Store Using Data Prefetching Techniques," in *Advanced Parallel Processing Technologies*, Cham, 2015, pp. 1–17.

[29] X.-S. Li, S.-K. Yoon, J.-G. Kim, and S.-D. Kim, "A Self-learning Pattern Adaptive Prefetching Method for Big Data Applications," *Sustainable Computing: Informatics and Systems*, 2017.

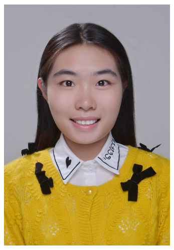
[30] D. Li, M. Dong, Y. Tang, L. T. Yang, K. Ota, and G. Zhao, "Triple-L: Improving CPS Disk I/O Performance in a Virtualized NAS Environment," *IEEE Systems Journal*, vol. 11, no. 1, pp. 152–162, March 2017.

Dingding Li received his Ph.D on Computer Science from the Huazhong University of Science and Technology at Wuhan, China. He is now a lecture of Computer Science at South China Normal University (SCNU). His research is focused around I/O virtualization and cloud computing.

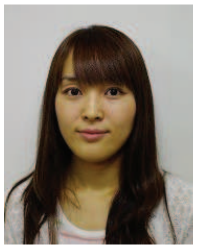


Mianxiong Dong received B.S., M.S. and Ph.D. in Computer Science and Engineering from The University of Aizu, Japan. He is currently an Associate Professor in the Department of Information and Electronic Engineering at the Muroran Institute of Technology, Japan. He was a JSPS Research Fellow with School of Computer Science and Engineering, The University of Aizu, Japan and was a visiting scholar with BBCR group at University of Waterloo, Canada supported by JSPS Excellent Young Researcher Overseas Visit Program from April 2010 to August 2011. Dr. Dong was selected as a Foreigner Research Fellow (a total of 3 recipients all over Japan) by NEC C&C Foundation in 2011. His research interests include Wireless Networks, Cloud Computing, and Cyber-physical Systems. He has received best paper awards from IEEE HPCC 2008, IEEE ICESS 2008, ICA3PP 2014, GPC 2015, IEEE DASC 2015, IEEE VTC 2016-Fall, FCST 2017 and 2017 IET Communications Premium Award. Dr. Dong serves as an Editor for IEEE Transactions on Green Communications and Networking (TGCN), IEEE Communications Surveys and Tutorials, IEEE Network, IEEE Wireless Communications Letters, IEEE Cloud Computing, IEEE Access, as well as a leading guest editor for ACM Transactions on Multimedia Computing, Communications and Applications (TOMM), IEEE Transactions on Emerging Topics in Computing (TETC), IEEE Transactions on Computational Social Systems (TCSS). He has been serving as the Vice Chair of IEEE Communications Society Asia/Pacific Region Meetings and Conference Committee, Leading Symposium Chair of IEEE ICC 2019, Student Travel Grants Chair of IEEE GLOBECOM 2019, and Symposium Chair of IEEE GLOBECOM 2016, 2017. He is the recipient of IEEE TCSC Early Career Award 2016, IEEE SCSTC Outstanding Young Researcher Award 2017 and The 12th IEEE ComSoc Asia-Pacific Young Researcher Award 2017.

Yanting Yuan is an undergraduate at South China Normal University. Her research interests include machine learning and database.



Jiaxin Chen is an undergraduate at South China Normal University. Her research interests include computer architecture and operating system.



Kaoru Ota was born in Aizu-Wakamatsu, Japan. She received M.S. degree in Computer Science from Oklahoma State University, USA in 2008, B.S. and Ph.D. degrees in Computer Science and Engineering from The University of Aizu, Japan in 2006, 2012, respectively. She is currently an Assistant Professor with Department of Information and Electronic Engineering, Muroran Institute of Technology, Japan. From March 2010 to March 2011, she was a visiting scholar at University of Waterloo, Canada. Also she was a Japan Society of the Promotion of Science (JSPS) research fellow with Kato-Nishiyama Lab at Graduate School of Information Sciences at Tohoku University, Japan from April 2012 to April 2013. Her research interests include Wireless Networks, Cloud Computing, and Cyber-physical Systems. Dr. Ota has received best paper awards from ICA3PP 2014, GPC 2015, IEEE DASC 2015, IEEE VTC 2016-Fall, FCST 2017 and IET Communications 2017. She is an editor of IEEE Transactions on Vehicular Technology (TVT), IEEE Communications Letters, Peer-to-Peer Networking and Applications (Springer), Ad Hoc & Sensor Wireless Networks, International Journal of Embedded Systems (Inderscience) and Smart Technologies for Emergency Response & Disaster Management (IGI Global), as well as a guest editor of ACM Transactions on Multimedia Computing, Communications and Applications (leading), IEEE Communications Magazine, IEEE Network, etc. Also she was a guest editor of IEEE Wireless Communications (2015), IEICE Transactions on Information and Systems (2014), and Ad Hoc & Sensor Wireless Networks (Old City Publishing) (2014). She was a research scientist with A3 Foresight Program (2011-2016) funded by Japan Society for the Promotion of Sciences (JSPS), NSFC of China, and NRF of Korea. She is the recipient of IEEE TCSC Early Career Award 2017.

Yong Tang got his BS and MSc degrees from Wuhan University in 1985 and 1990, respectively, and PhD degree from University of Science and Technology of China in 2001, all in computer science. Dr. Tang is now a Professor and Dean of School of Computer Science, South China Normal University (SCNU). He serves also as the Director of Services Computing Engineering Research Center of Guangdong Province. He was vice Dean of School of Information of Science and Technology, Sun Yat-sen University, before joined SCNU in 2009. His current interests include temporal database, cooperative computing, cloud computing and social network services. He has published more than 200 papers in various journals, conferences and books.