An In-Kernel NOSQL Cache for Range Queries Using FPGA NIC

Korechika Tamura
Dept. of ICS, Keio University,
3-14-1 Hiyoshi, Kohoku-ku, Yokohama, Japan
Email: tamura@arc.ics.keio.ac.jp

Hiroki Matsutani
Dept. of ICS, Keio University,
3-14-1 Hiyoshi, Kohoku-ku, Yokohama, Japan
Email: matutani@arc.ics.keio.ac.jp

***Abstract*—To make use of big data, various NOSQL data stores have been deployed, such as key-value stores and column-oriented stores. NOSQL data stores typically achieve a high degree of scalability, while specialized for some specific purposes; thus, Polyglot persistence that employs multiple NOSQL data stores complementally is a practical choice toward a high diversity of application demands. We assume various NOSQL data stores running on database servers are accessed by clients via a network. This paper aims to improve performance of the Polyglot persistence by introducing an FPGA-based 10GbE network interface (NIC) and In-Kernel NOSQL Cache (IKNC) implemented in the NIC device driver. IKNC stores query results as a key-value form in a host memory, and the requested data can be returned to clients immediately if the query result has been cached. In the IKNC key-value pair, the key is a hashed value of a given search query and the value is a query result of the search query. Existing works have focused only on key-value stores, while that for column-oriented stores that support range queries (e.g., scan operation) has not been addressed. In this paper, we also propose two cache strategies of IKNC for column-oriented stores. In our experiments, Apache HBase is running on an application layer, while our IKNC with the proposed cache strategies is implemented on an FPGA-based NIC and its device driver. A significant performance improvement is achieved by the proposed IKNC and pros and cons of the proposed two cache strategies are demonstrated.**

I. Introduction

To manage large data sets generated in various network services, structured storages (NOSQLs) [1] have been deployed as scalable data stores for big data. Various types of structured storages are available and they can be classified into four types based on their data structures: key-value store, column-oriented store, document-oriented store, and graph databases. Key-value type stores the data as pairs of key and value [2]. Column-oriented storage stores the data as sorted rows each of which consists of multiple columns (i.e., key-value pairs) [3][4]. Document-oriented type is a scheme-less data store where data are stored as documents. Graph database represents the data as a graph that consists of nodes, their properties, and relationships. Structured storages are suitable for managing big data, because they employ simpler data structures and are specialized to store and retrieve data so that they can achieve high scalability as compared to the RDBMS. Thus, Polyglot persistence that employs multiple structured storages complementally is a practical choice toward a high diversity of application demands while managing large data sets.

Structured storages are running on database servers and often accessed by clients via network, such as Memcached. As their queries are simple, network processing time (e.g., TCP/IP network protocol stack and Socket APIs) is long compared to the computation time at the server; so the network processing is a bottleneck as reported in [5]. To address this issue, various studies have been reported and they can be classified into three approaches: 1) Key-value store appliance using FPGA-based network interface [5], 2) Bypassing network protocol stack [6], and 3) In-kernel key-value store. This paper focuses on the third approach, where a large software cache is implemented in the Linux kernel space as a loadable kernel module for the network interface. We call this approach In-Kernel NOSQL Cache (IKNC). When a received query hits in IKNC, the response is immediately returned to the client without accessing kernel network protocol stack and structured storage server software. We implemented IKNC as a loadable driver of NetFPGA-10G network interface and use the FPGA to offload a hash computation and CRC checksum computation to reduce the query response time. Only a small modification is needed on each structured storage. Since our target is Polyglot persistence, in this paper, we propose an IKNC design and apply it to key-value store and column-oriented store.

IKNC stores query results as a key-value form in a host memory, and the requested data can be returned to clients immediately if the query result has been cached. In the IKNC key-value pair, the key is a hashed value of a given search query and the value is a query result of the search query. In this paper, we employ an FPGA-based network interface to offload a hash computation function and checksum computation and implement IKNC on top of the network interface. Existing works have focused only on key-value stores, while that for column-oriented stores that support range queries (e.g., scan operation) has not been addressed. In this paper, we also propose two cache strategies of IKNC for column-oriented stores. In our experiments, Apache HBase is running on an application layer of a server machine, and our IKNC with the proposed cache strategies is implemented on an FPGA-based NIC and its device driver. A client machine connected to the server via 10GbE transmits HBase queries using maximum network bandwidth, and we measure the number of operations processed by the server per a certain time (i.e., throughput). As a result, a significant performance improvement is achieved by the proposed IKNC. We also discuss pros and cons of the proposed two cache strategies.

The rest of this paper is organized as follows. Section II

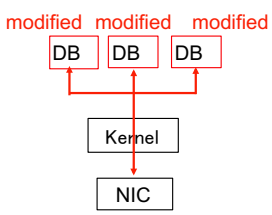

Fig. 1. Network protocol bypassing approach

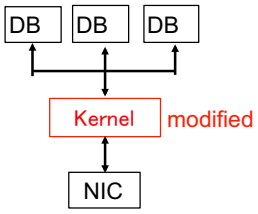

Fig. 2. In-kernel data store approach

overviews related work and Section III introduces structured storages. Section IV proposes IKNC and Section V shows the design and implementation. Section VI shows experimental results and Section VII concludes this paper.

II. Related Work

This section briefly introduces accelerations of relational databases and structured storages.

To improve query performance of relational databases, FPGA-based hardware accelerators have been reported since 2009. A query compiler for FPGA-based database accelerator, called Glacier, is proposed in [7]. In response to an input relational database query, the compiler generates a dedicated query accelerator for FPGAs. In [8], different queries can be accelerated without reconfiguring the FPGA for each query. In [9] and [10], graph database and document-oriented data store are accelerated by GPUs, respectively.

Memcached is a distributed caching system that employs a key-value data structure and it has been widely used in data centers. Memcached appliances implemented on FPGA boards with fast network interfaces have been reported since 2013 [5]. As reported in [5], 64usec and 30usec are consumed at the network interface and Linux kernel (network protocol stack) respectively, while only 30usec is consumed for the Memcached processing at the server. Since network processing consumes a significant time compared to the Memcached computation, an FPGA-based Memcached appliance with 1GbE interface is proposed in [5].

FPGA-based Memcached appliances with 10GbE interface are proposed in [11], [12], and [13]. In these designs, Memcached operation is divided into packet decomposition, hash computation, memory access, and response formatter, and these steps are performed in a pipeline manner. They are designed as standalone FPGA-based Memcached appliances. In [14], a dedicated SoC (System-on-Chip) for Memcached queries is proposed. A software and hardware co-design for the Memcached appliance is also discussed in [14].

The above-mentioned FPGA-based approach is very efficient in terms of performance per Watt, but the storage capacity is limited by the on-board DRAMs implemented on FPGA board.

As a software-based approach, in [6], a key-value store server software running on an application layer directly accesses the network interface to process key-value store queries by using Intel DPDK [15], as shown in Figure 1. As the kernel network protocol stack is bypassed, this approach can eliminate the overheads of network protocol stack. However, we need to modify the structured storage server software to directly access the network interface. In Polyglot persistence, different structured storages complementally coexist and thus we need to modify all the structured storages required for application demands.

Another software-based approach is to process key-value store queries inside an operating system kernel. In [16], a key-value store is implemented in Linux kernel using Netfilter framework. Memcached queries received in the kernel are hooked so that a customized handler is called to process the key-value store queries in the kernel context. As the key-value store is implemented inside the kernel, it can eliminate overheads for network protocol processing and socket APIs.

In this paper, we extend the idea of in-kernel key-value store [16] to In-Kernel NOSQL Cache (IKNC) for Polyglot persistence. As shown in Figure 2, various structured storage servers are running on an application layer and IKNC is implemented as a loadable network interface driver. When a received query hits in IKNC, the response is immediately returned to the client without accessing kernel network protocol stack and structured storage server software. Otherwise, the query is transferred to the application layer and processed by the structured storage server software as usual. In our design, only a small modification is needed on each structured storage, as shown in Figure 2. Since our target is Polyglot persistence, in this paper, we propose an IKNC design and apply it to key-value store and column-oriented store. Actually, most prior works have focused only on a Memcached-style key-value store and that for column-oriented stores have not been addressed, though column-oriented store is an important class of structured storage [3][4] as introduced in the next section.

III. Column-Oriented Stores

In this paper, the proposed IKNC is applied to column-oriented stores in addition to key-value stores. It is a distributed three-dimensional sorted map in which data are indexed by row key, column name, and timestamp. This concept first appears in Google BigTable which has been used as data storage systems. A similar data structure can be seen in Apache HBase and Apache Cassandra. We assume application to Apache HBase while it can be applied to the other implementations.

In this section, the original data structure called “Flat-Wide” is introduced and then the transformed version called “Tall-Narrow” suitable for caching is introduced. Their typical queries are also shown.

A. Flat-Wide Data Structure

Figure 3 illustrates the data structure of a column-oriented store. The data can be represented as rows sorted by their row keys. In this example, the leftmost field in each row is row key (e.g., Row#101) and the data set consists of six rows (i.e., Row#101 to Row#119) sorted by their row keys. Each row consists of multiple columns, each of which is a pair of name and value. Arbitrary columns can be added dynamically for each row. For example, Row#101 consists of

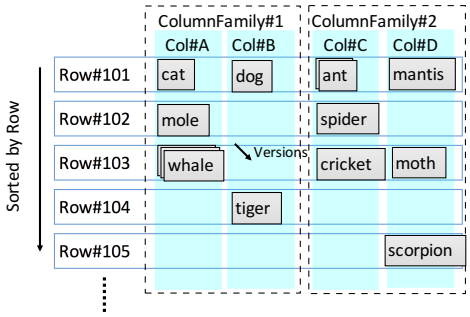

Fig. 3. Flat-wide structure in Column-oriented store

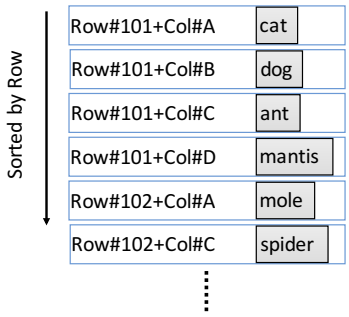

Fig. 4. Tall-narrow structure in Column-oriented store

four columns whose names are Col#A, Col#B, Col#C, and Col#D, while Row#104 consists of only two columns. Please note that corresponding values are not shown in this figure for simplicity. These columns can be grouped by column family. In this example, columns are grouped as ColumnFamily#1 and ColumnFamily#2. Column families are defined beforehand and cannot be added dynamically. In addition, multiple revisions with timestamps can be stored for each value; thus the past revisions can be retrieved by specifying a time.

The above-mentioned data structure is called "Flat-Wide" since each row has multiple columns and looks flat and wide.

B. Tall-Narrow Data Structure

The Flat-Wide data structure can be transformed into key-value pairs which are simpler and sutaible for caching. Figure 4 illustrates the Tall-Narrow data structure. A row that consists of multiple column names are transformed into multiple rows whose row key is a longer string that concatenate the original row key and each column name. In this case, each row has only a single column and thus it has only a single value. In this example, a single row Row#101 that consists of Col#A, Col#B, Col#C, and Col#D in Figure 3 is transformed into four key-value pairs Row#101+Col#A to Row#101+Col#D in Figure 4.

The above-mentioned data structure is called "Tall-Narrow" since each row has only a single column (i.e., narrow) and the number of rows is longer than that of Flat-Wide.

C. Queries

The following query reads a value field specified by Row#101 and Col#A in Figure 3 (Get operation).

```
Get "Row#101", "ColumnFamily#1:Col#A"
```

The following query writes a new value to a value field specified by Row#101 and Col#B (Set operation).

```
Put "Row#101", "ColumnFamily#2:Col#C",
    "newValue"
```

The following query reads all the rows between Row#102 and Row#104 (Scan operation).

```
Scan startRow="Row#102", stopRow="Row#104"
```

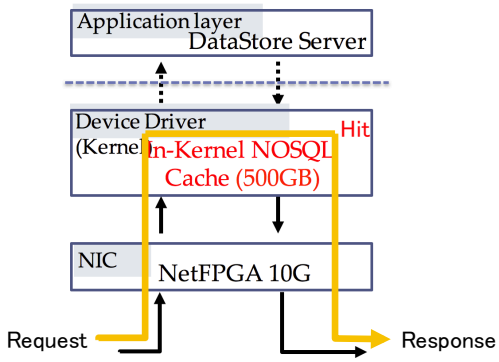

Fig. 5. Overview of IKNC and flow of query packet

IV. In-Kernel NOSQL Cache

In this paper, we implemented IKNC as a cache in Linux kernel in order to accelerate Get/Scan queries (i.e., read queries) on Polyglot persistence. Figure 5 illustrates how read query packets are processed with IKNC. We assume that a column-oriented store server is running on a user space and IKNC allocates a large memory space (e.g., 500GB) in the kernel space. Query results of the column-oriented store are cached as key-value pairs in IKNC. Then, the query result can be returned to clients immediately from IKNC if the requested rows have been cached in IKNC. In the IKNC key-value pair, the key is a hashed value of a given search query, such as Get, Set, and Scan operations of the column-oriented store. We offload the hash computation to FPGA. The value is a query result of the search query given by the column-oriented store server when the query does not hit in the IKNC and thus processed by the column-oriented store server. After IKNC makes the packet, which includes the response, IKNC sends it back to client via a network. To reduce query response time, IKNC offloads the CRC computation to FPGA.

The above-mentioned IKNC approach works well for key-value stores or Get/Set operations of the column-oriented stores. To support Scan operations (i.e., range queries) used in the column-oriented stores for the IKNC, the following subsections propose two cache strategies: All-row caching and Each-row caching for column-oriented stores.

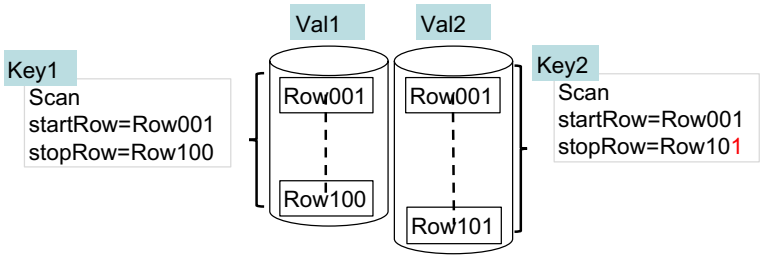

Fig. 6. All-row caching policy

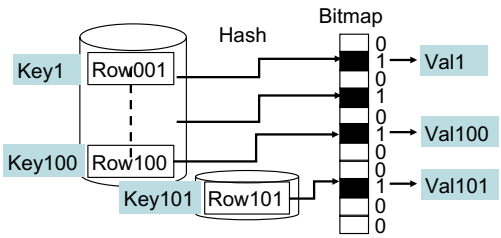

Fig. 7. Each-row caching policy

A. All-Row Caching

A straightforward caching strategy for Scan operations is All-row caching, in which a pair of startRow and stopRow in a given query is hashed and used as a key of a key-value pair in IKNC. Selected rows between StartRow and StopRow are cached as the value of the corresponding key-value pair.

Figure 6 illustrates two query results cached in IKNC with All-row caching strategy. Key1 is a hashed value of the first query (i.e., scan startRow=row001 stopRow=row100) and the result is cached as Val1. Key2 is a hashed value of the second query (i.e., scan startRow=row001 stopRow=row101) and the result is cached as Val2. If IKNC receives the first or second query again, the result is returned to the client without accessing the network protocol stack and column-oriented storage running on an application layer.

One issue of All-row caching strategy is inefficiency of memory usage. As shown in the figure, Val1 is completely overlapped with Val2 since row001 to row100 are also cached in Val2. Another problem is that All-row caching strategy may introduce more cache invalidations as the number of rows (i.e., range) associated with a key increases. For example, if row050 is updated later, then both Val1 and Val2 will be invalidated and removed from IKNC.

IKNC with All-row caching strategy is simple and fast, because complete query results for the column-oriented store can be cached and immediately returned to clients only with a single IKNC memory access. However, due to the above-mentioned issues, it is useful only when the identical query is repeated without data modifications.

B. Each-Row Caching

In the case of IKNC with Each-row caching strategy, each row retrieved by a Scan query is cached as a single key-value pair.

It employs a bitmap table in which rows currently cached are marked as "cached." Figure 7 illustrates two query results cached in IKNC with Each-row caching strategy. The first query is "scan startRow=row001 stopRow=row100" while the second query is "get row101." Each row retrieved by these queries are individually cached in IKNC. The bitmap (0 or 1) indicates which rows are currently cached in IKNC. A hashed value of a row key is used as an index of that key in the bitmap. When a row currently cached is updated, it is invalidated in IKNC by simply storing 0 in the corresponding bit in the bitmap.

When IKNC with Each-row caching strategy receives a Scan query, it parses every row and checks the bitmap respectively if all the rows requested in the query are cached in IKNC. A hash computation is needed for each row to find its key. Because this operation is complicated, the hash computation is left to CPU in the case of Each-Row caching strategy. If all the rows are cached, IKNC retrieves all the rows from the cache and return the result to the client. Otherwise, the query is simply transferred to the column-oriented store via network protocol stack.

Memory efficiency of Each-row caching strategy is better than that of All-row caching, because each row is cached by at most a single key and never cached by multiple keys. One issue with Each-row caching is that it requires to compute a hashed value (key) and read the row data (value) from IKNC for each row; thus computation cost increases as the number of rows requested in the range query increases.

V. Implementation

A. Target Platform

Client and server machines directly connected via a 10GbE direct attached cable are used in our experiment. HBase is running on the server machine as a column-oriented store software and all data are stored on HDD. A client application generates HBase queries (e.g., Get, Set, and Scan) and sends them to the server. A prototype of IKNC shown in Figure 5 is implemented in the server side in order to accelerate the column-oriented store queries.

NetFPGA-10G board [17] is mounted in the server machine via a PCI-Express Gen2 x8 interface as a 10GbE network interface. CRC checksum computations and hash computations needed for INKC with All-row caching are offloaded to the FPGA-based network interface (FPGA NIC for short) [1]. Our IKNC with both All-row and Each-row caching strategies is implemented as a loadable device driver of the FPGA NIC.

In the following subsections, we will explain the server-side software implementation in Section V-B and a hash table design of IKNC in Section V-C.

B. NOSQL Daemon Server (NDS)

We assume that multiple structured storages are running on the server, and a column-oriented store such as HBase is one

[1]Hash computations needed for Each-row caching are not offloaded to the FPGA NIC. It is our future work.

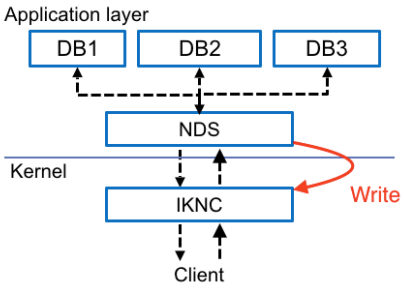

Fig. 8. Structured storage software and IKNC at server side

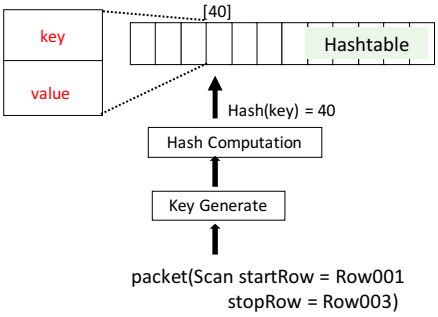

Fig. 9. Hashtable in IKNC

of the structured storages running on the Polyglot persistence environment. To support multiple structured storages, here we introduce an NOSQL daemon server (NDS) that receives all the queries from clients and passes them to proper structured storage servers.

a) From Client to NDS: Figure 8 illustrates NDS and IKNC in the server machine. A client generates HBase compatible queries (e.g., Get, Set, and Scan) to NDS running as an application at the server machine. When a read query from a client does not hit in INKC, it is transferred to NDS via a kernel network protocol stack and then passed to HBase at the server. In our prototype implementation, we employ UDP as a transport layer protocol for the communications between clients and NDS in order to demonstrate raw performance of IKNC.

b) From NDS to HBase: We implemented communications between NDS and HBase by using Apache Thrift APIs [18]. When NDS receives a Get/Scan query from a client, it forwards the query to HBase using the Thrift remote procedure calls.

c) From HBase to NDS: When NDS receives the response from HBase, it forwards the response to the client. The response sent from NDS to the client are peeked and cached in IKNC, as shown in a red line in Figure 8. Thus, if IKNC receives the same query again (or queries that request the rows already cached), it can return the query result to a requestor immediately.

d) Invalidation of IKNC: When NDS receives a Set operation from a client to add or modify data in HBase, it checks whether the rows to be updated are cached in IKNC. If the cached rows will be updated by the query, NDS invalidates them in IKNC.

TABLE I
SERVER MACHINE AND CLIENT MACHINE

	Server	Client
Processor	Intel Xeon E5-2637 v3 @ 3.50GHz	Intel Core i5-3470S @ 2.90GHz
Memory	512 GByte	4 GByte
OS	CentOS Linux 6.6	Ubuntu Linux 13.04
NIC	NetFPGA-10G (PCIe Gen2 x8)	Mellanox 10GbE NIC

C. In-Kernel NOSQL Cache (IKNC)

We implemented IKNC as a loadable driver of NetFPGA-10G network interface. More specifically, we added necessary functions of IKNC to the Linux device driver of Reference NIC provided by NetFPGA project [17]. A large memory block up to 500GB is statically allocated by the device driver. It is called Hashtable and used for the caching at IKNC.

Pairs of HBase query and the result are cached in Hashtable implemented in IKNC. A hashed value of the query is used as an index in Hashtable, as shown in Figure 9. That is, the index is used as memory address where the query and the result are stored in Hashtable. Figure 9 illustrates how the cached value is retrieved from Hashtable in IKNC. When IKNC receives a packet that contains a Get/Scan query from a client, it first extracts the query and then computes a hashed value of the query as an index. The index is used as a memory address where the corresponding value is cached in Hashtable. After reading the cached value from Hashtable, IKNC returns the result to the client immediately.

VI. EVALUATIONS

A. Evaluation Environment

Tables I show the server and client machines used in experiment. IKNC is implemented in the server machine. We measured the throughput of Scan operations using these machines. We used UDP as a transport layer protocol. For the Scan query, the payload consists of an operation type (i.e., Scan), startRow, and stopRow. Size of rows is limited by up to 32Byte.

B. Evaluation Results

In this experiment, the throughput indicates the number of Scan operations processed by the server per a second (Ops/sec). A real processing throughput of Scan queries depends on cache hit rate of IKNC. We thus measured the throughput when all the Scan queries are hit in INKC with All-row caching and Each-row caching strategies, respectively. We also measured the throughput of the original HBase, which is corresponding to throughput when all the Scan queries are missed in IKNC and processed by HBase.

Figure 10 shows throughput of HBase only, All-row caching, and Each-row caching. Scan queries that request a hundred rows are generated by the client. The client machine sends over one million request packets and receives the response from the server. As shown, the throughput of HBase only is 1.35k Ops/sec. Although it is smaller than 2.5k Ops/sec which is a reported value in [19], performance improvement by IKNC is significant.

When we compare All-row caching and Each-row caching strategies, All-row caching outperforms Each-row caching by

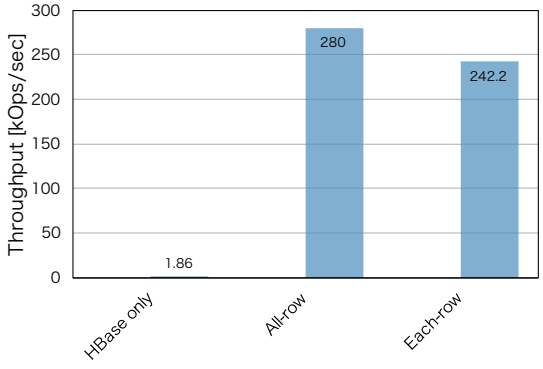

Fig. 10. Throughput of HBase only, All-row caching, and Each-row caching

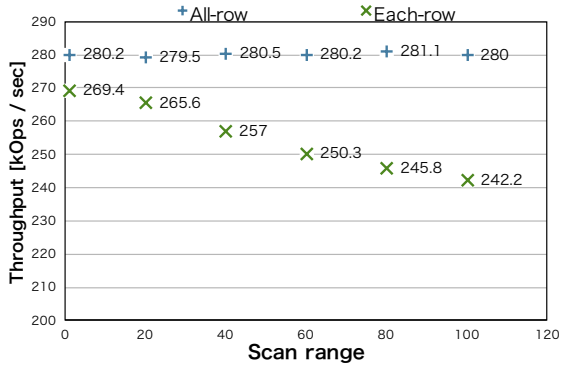

Fig. 11. Throughput vs. Scan range

15.6%. As every Scan query requests a hundred rows, INKC with Each-row caching computes a hash and reads the memory a hundred times, while that with All-row caching does the same only once. Figure 11 shows the throughputs of All-row caching and Each-row caching when the number of rows requested by a Scan query is changed from 1 to 100. As the Scan range is reduced, the throughput difference between All-row caching and Each-row caching is decreased. All-row caching slightly outperforms Each-row caching because the hash computation and cache search using bitmap are offloaded to FPGA NIC.

Please note that although All-row caching outperforms Each-row caching in terms of performance, memory efficiency of Each-row will be better than that of All-row caching as discussed in Section IV. As All-row caching and Each-row caching strategies can coexist in the same IKNC, we can select a proper caching strategy for each query, depending on the query access pattern as a future work.

VII. Summary and Future Work

In this paper, we implemented IKNC as a loadable device driver of an FPGA NIC for Polyglot persistence including column-oriented stores. IKNC is suitable for Polyglot persistence compared to the kernel bypassing approach since all the structured storage servers are required to be modified in the case of the kernel bypassing approach. As column-oriented store supports range queries, we proposed two caching strategies: All-row caching and Each-row caching strategies. CRC checksum computation and a part of hash computation are offloaded to the FPGA NIC, while we focused on IKNC design as a software approach in this paper. On top of INKC, we implemented NDS which is a daemon server for Polyglot persistence that receives structured storages' queries and passes them to proper structured storages using Thrift APIs.

Experimental results showed that HBase with IKNC significantly improves the Scan query performance compared to the original HBase. IKNC with All-row caching strategy outperforms that with Each-row caching strategy by up to 15.6%. Their performance gap increases as the number of rows requested in Scan queries increases, because Each-row approach repeats hash computation and memory read for each row. However, memory efficiency of Each-row strategy is better than that of All-row strategy since the same rows may be cached more than twice in All-row caching. As a future work, we are planning to combine both the caching strategies by considering the size and access pattern of each query. We are also planning to propose caching strategies for document-oriented stores and graph databases toward future Polyglot persistence.

Acknowledgements This work was supported by SECOM Science and Technology Foundation and JST PRESTO.

References

[1] Pramod J. Sadalage and Martin Fowler, *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley, 2012.
[2] "Memcached - A Distributed Memory Object Caching System," http://memcached.org.
[3] "The Apache HBase Project," http://hbase.apache.org.
[4] "The Apache Cassandra Project," http://cassandra.apache.org.
[5] S. R. Chalamalasetti *et al.*, "An FPGA Memcached Appliance," in *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA'13)*, Feb. 2013, pp. 245–254.
[6] H. Lim *et al.*, "MICA: A Holistic Approach to Fast In-Memory Key-Value Storage," in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI'14)*, Apr. 2014, pp. 429–444.
[7] R. Mueller, J. Teubner, and G. Alonso, "Streams on Wires: A Query Compiler for FPGAs," in *Proceedings of the International Conference on Very Large Data Bases (VLDB'09)*, Aug. 2009, pp. 229–240.
[8] B. Sukhwani *et al.*, "Database Analytics Acceleration Using FPGAs," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'12)*, Sep. 2012, pp. 411–420.
[9] S. Morishima *et al.*, "Performance Evaluations of Graph Database using CUDA and OpenMP-Compatible Libraries," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 4, pp. 75–80, Sep. 2014.
[10] ——, "Performance Evaluations of Document-Oriented Databases using GPU and Cache Structure," in *Proceedings of the International Symposium on Parallel and Distributed Processing with Applications (ISPA'15)*, Aug. 2015, pp. 108–115.
[11] M. Blott *et al.*, "Achieving 10Gbps Line-rate Key-value Stores with FPGAs," in *Proceedings of the USENIX Workshop on Hot Topics in Cloud Computing (HotCloud'13)*, Jun. 2013.
[12] M. Blott and K. Vissers, "Dataflow Architectures for 10Gbps Line-rate Key-value-Stores," in *Proceedings of the IEEE Symposium on High Performance Chips (HotChips'13)*, Aug. 2013.
[13] M. Blott, L. Liu, K. Karras, and K. Vissers, "Scaling out to a Single-Node 80Gbps Memcached Server with 40Terabytes of Memory," in *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'15)*, Jul. 2015.
[14] K. Lim *et al.*, "Thin Servers with Smart Pipes: Designing SoC Accelerators for Memcached," in *Proceedings of the International Symposium on Computer Architecture (ISCA'13)*, Jun. 2013, pp. 36–47.
[15] "Intel Data Plane Development Kit (Intel DPDK) Overview," Dec. 2012.
[16] Y. Xua, E. Frachtenbergb, and S. Jiang, "Building a High-Performance Key-Vluec Cache as an Energy-Efficient Appliance," *Performance Evaluation*, vol. 79, pp. 24–37, Sep. 2014.
[17] "NetFPGA," http://netfpga.org.
[18] "Apache Thrift," http://thrift.apache.org.
[19] T. Rabl *et al.*, "Solving Big Data Challenges for Enterprise Application Performance Management," in *Proceedings of the International Conference on Very Large Databases (VLDB'12)*, Aug. 2012, pp. 1724–1735.