Design and Implementation of Hardware Cache Mechanism and NIC for Column-Oriented Databases

Akihiko Hamada
Dept. of ICS, Keio University,
3-14-1 Hiyoshi, Kohoku-ku, Yokohama, Japan
Email: hamada@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*—Recently some researches to utilize big data efficiently have been made vigorously. To store and process big data, structured storages (NOSQLs) that have high degree of horizontal scalability have attracted a lot of attention. Key-value stores and column-oriented stores are known as famous examples of structured storages. Especially, column-oriented stores can store variable numbers of columns for each row while maintaining high scalability. Moreover, range queries (scan operations) are supported in column-oriented stores. This paper proposes hardware cache mechanism using FPGA NIC to accelerate column-oriented databases. In this paper, it is assumed that column-oriented stores running on database servers are accessed by clients via a network. This paper aims to improve performance and power efficiency of column-oriented stores by introducing an FPGA-based 10GbE network interface (NIC) and a hardware cache mechanism (HBC) implemented on the NIC. HBC stores query results (sorted rows) as a key-value form in the DRAM implemented on the FPGA NIC, and the requested data can be returned to clients immediately if the query result has been cached. Existing work that aims to accelerate structured storages by hardware have focused only on key-value stores while column-oriented stores that support range queries (scan operations) have not been addressed. HBC deploys methods that address data mappings and range queries of caches using specific data structures that can be represented in binary-tree forms and this paper shows HBC can accelerate range queries by hardware. In experiments of this paper, HBase is running on an application layer, while HBC is implemented on an FPGA-based NIC. This paper shows that improvement of power efficiency and significant performance improvement can be achieved by the proposed HBC and also pros and cons of the proposed HBC are discussed.**

I. Introduction

There have recently been a lot of researches into the efficient utilization of big data. Some structured storages (NOSQLs) [1] that have high degree of horizontal scalability attracted a lot of attention in addition to traditional RDBMS for storing and utilizing big data. As famous examples of structured storages, key-value stores [2] that store the data as pairs of key and value and column-oriented stores [3][4] that store the data as sorted rows that have row keys and multiple columns (key-value pairs) are widely known. Especially, column-oriented stores can store variable numbers of columns for each row while maintaining high scalability. Moreover, some range queries (scan operations) between given startRow and stopRow can be performed because stored data are sorted by row keys. Column-oriented stores can manage data more flexibly compared to key-value stores. These structured storages shown above are considered to have characteristics that accessing memories and data transfer (I/O) cost much time compared to database processing, thus it is considered that higher performance can be achieved by connecting I/O and database processing closely.

This paper proposes methods for acceleration of column-oriented stores by a hardware cache mechanism that deploys FPGA NIC. More specifically, a hardware cache mechanism of column-oriented stores that works on an FPGA NIC that has 10GbE interfaces is introduced. This hardware cache mechanism proposed in this paper is called HBC (HBase Cache). In this paper, HBC is designed and a prototype of HBC is implemented.

978-1-5090-3707-0/16/$31.00 ©2016 IEEE

HBC stores results of range queries (scan operations) performed by software deploying the DRAM implemented on the FPGA NIC. When a received query hits in HBC, the requested data that HBC caches are returned to the client directly from the NIC without software processing, thus access to the databases can be accelerated. When a received query is missed in HBC, HBC transfers the query to a software layer and software can generate the result and return the requested data to the client.

To perform returning scanned results of column-oriented stores, HBC deploys specific data structures represented in binary-tree forms to set tag information and performs search of cached data using the tag information. When a scan query is received in HBC, it searches the range of requested data and the decision of hit or miss is made. In other words, HBC performs hardware-based scan operations on FPGA NIC. Existing work has focused only on key-value stores and work that accelerates scan operations of column-oriented stores by hardware has not been reported as far as we know. This paper introduces proposed HBC and HBCMS (HBC management system) that is software cooperating with HBC and column-oriented stores. HBCMS cooperates with HBC and column-oriented stores, however, modification to column-oriented stores is not needed as HBCMS is independent from column-oriented stores themselves. In the proposed methods, when a received query hits in HBC and HBC returns the result, column-oriented stores do not detect the returned result and when a received query is missed in HBC and software returns the result, HBC does not detect the returned result. Thus, they compose a system where the databases and HBC are transparent each other.

In the experiments of this paper, HBase and HBCMS are running on an application layer on a server machine and HBC is implemented on the FPGA NIC mounted on the server machine. Clients transfer HBC queries to the server to request data and when they are missed in HBC, the queries are processed by software. The client machine is connected to the server machine via 10GbE and the client sends queries for HBC using maximum network bandwidth by a hardware-based packet injector, and we measured the number of operations processed by the server per a certain time (i.e., throughput). As a result, orders of magnitude higher performance is achieved by proposed HBC compared to that of software. Pros and cons of the proposed methods are also discussed.

The rest of this paper is organized as follows. Section II overviews related work and Section III introduces column-oriented stores. Section IV proposes HBC 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 overviews accelerations of relational databases and structured storages.

To improve query performance of relational databases, researches of FPGA-based hardware acceleration have been reported since 2009. A query compiler for FPGA-based database accelerator that is called Glacier, is proposed in [5]. This compiler receives RDB queries as inputs and it realizes dedicated hardware

to process the queries on the FPGA. In [6], different queries can be processed without reconfiguring the FPGA.

Researches to accelerate Memcached that is a key-value store by dedicated hardware have been reported since 2013. [7] reports that 64usec and 30usec are consumed at the network interface and Linux kernel (network protocol stack) respectively, while only 30usec is consumed for the software processing of Memcached at the server. Since network processing consumes much longer time compared to the Memcached computation, a Memcached appliance that is implemented on an FPGA board with 1GbE interface is proposed in [7]. Memcached appliances implemented on FPGA boards with 10GbE interfaces are proposed in [8] and [9]. In these designs, Memcached operations are composed of packet decomposition, hash computation, memory access, and response formatter, and these steps are processed in a pipeline. They are designed as standalone Memcached appliances implemented on FPGA boards. [10] proposes accelerators as SoC (System-on-Chip) for Memcached processing. Co-design of software and hardware for the Memcached processing is also discussed in [10].

As mentioned above, Memcached that has the simplest data structures in structured storages is considered to have characteristics that accessing memories and data transfer (I/O) cost much time compared to database processing. For such kind of uses, devices that can connect I/O and database processing closely such as FPGAs can be deployed effectively. Also, existing work of hardware-based approach on structured storages is mainly targeting Memcached that is one of key-value stores whose storages are volatile (persistence of stored data using hard disks is not supported). On the other hand, column-oriented stores have been utilized in various fields since Google proposed BigTable as a storage to store indexes of the web and some open source databases such as HBase [3] and Cassandra [4] are widely known. However, researches of hardware-based acceleration of column-oriented stores have not been reported as far as we know. This paper proposes methods to accelerate column-oriented stores using FPGA NIC.

III. Column-oriented Databases

In this paper, proposed HBC is applied to column-oriented stores. To accelerate column-oriented stores is considered to be effective as they have more flexible functions compared to key-value stores. This work is targeting HBase as a column-oriented store while it can be applied to other column-oriented stores.

This section shows basic data structures and queries examples of column-oriented stores.

a) Data Structures of Column-oriented Stores: The data structure of column-oriented stores is an advanced version of that of key-value stores and it is composed of row keys and columns that belong to rows. The data structure of a column-oriented store is shown in Figure 1. Column-oriented stores can store

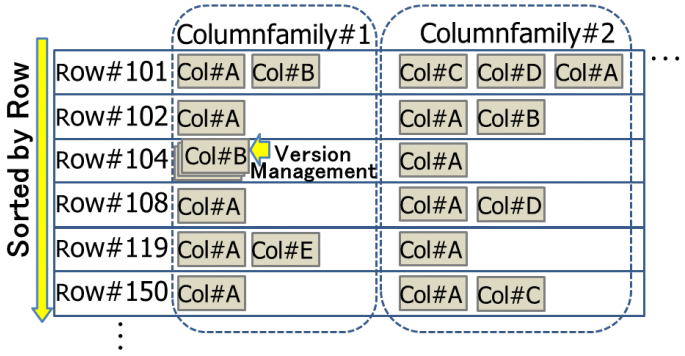

Fig. 1. Data structure of Column-oriented store

data as rows that have row keys (e.g., Row#101) and columns (key-value pairs) that are assigned to row keys. The values are not shown in this figure for simplicity. As rows are sorted by their row keys and stored, range queries (scan operations) that designate the range of row keys (startRow and stopRow) can be performed. Columns (e.g., Col#A) are also categorized by column families (e.g., ColumnFamily#1) and stored. Columns can be added dynamically, however, column families can not be added dynamically and need to be configured before the system starts. Columns can retain multiple versions of data by their timestamps. To read old revisions of data by designating some specific versions of data can be performed.

b) Queries of Column-Oriented Stores: As typical queries of column-oriented stores, Get, Set and Scan operations can be performed.

The query shown below reads a value specified by Row#101, ColumnFamily#1, and Col#A (Get operation).

Get “Row#101”, “ColumnFamily#1:Col#A”

The query shown below writes a new value (Val1) to a value field specified by Row#100, ColumnFamily#2, and Col#B (Set operation).

Put “Row#100”, “ColumnFamily#2:Col#B”, “Val1”

The query shown below reads all the rows between Row#100 and Row#105 (Scan operation).

Scan startRow=“Row#100”, stopRow=“Row#105”

IV. Hardware Cache Mechanism

In this paper, HBC is implemented on FPGA NIC as a hardware cache to accelerate Get/Scan queries (Read queries) of column-oriented databases. Figure 2 shows how packets of read queries are processed by HBC. In this paper, a column-

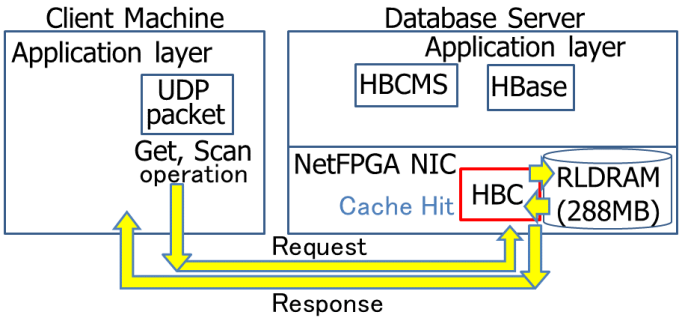

Fig. 2. Overview of HBC and data flow of query packets

oriented store and HBCMS are running on an application layer of the server machine and HBC running on the FPGA NIC mounted on the server machine uses the DRAM (we are currently using NetFPGA-10G board that has only 288MB DRAM, while NetFPGA-SUME board that will be used in the future work has 8GB DRAM [11]) on the FPGA NIC to store data in a hardware cache. The query results of column-oriented stores are cached in HBC as key-value pairs. If requested rows are cached in HBC, the query results can be returned from HBC to clients rapidly. In a key-value pair of HBC, the key is the string composed of a row key, a column family, and a column of each row in a data mapping manner of HBC shown in Section IV-A. The values in HBC are the scanned results (sorted rows) of requested read queries generated by the column-oriented database server when requested queries do not hit in HBC. HBC returns packets that includes requested data after it generates them via a network to clients. In this case, HBC can return requested data from FPGA NIC to clients without processing by software, thus it can accelerate the data access.

The methods of HBC can be applied to Get/Set/Scan operations of column-oriented databases. In the following subsections, the caching manner of HBC and HBCMS that cooperates with HBC and column-oriented databases are proposed. Deploying these methods, Get/Set/Scan operations used in column-oriented stores are supported in HBC.

A. Caching Manner of HBC

HBC deploys the DRAM (288MB) on the FPGA NIC board and thus it can uses 2^{23}addresses while each address can store data up to 36Byte.

Some parameters used in the data mapping manner of HBC are introduced below. The maximum numbers of different column families that can be used in HBC is represented as $N_{cf} = 2^l$. The maximum numbers of different columns that can be used in HBC is represented as $N_c = 2^m$. Thus the maximum numbers of different pairs of a column family and a column that can be used in HBC is represented as $N_{col} = 2^{l+m}$. The maximum numbers of rows that can be cached for each pair of a column family and a column is represented as $N_{row} = 2^n$. The maximum size of a value stored for each row is represented by the following equation.

$$S_{value} = 36 \times 2^a Byte \tag{1}$$

Also the following equation is introduced.

$$l + m + n + a = 23 \tag{2}$$

Thus, the parameters l, m, n, and a are used to configure the data mapping deploying cache of 2^{23}addresses (288MB).

Figure 3 shows the structure of an address used in the data mapping of HBC. As shown in this figure, HBC uses

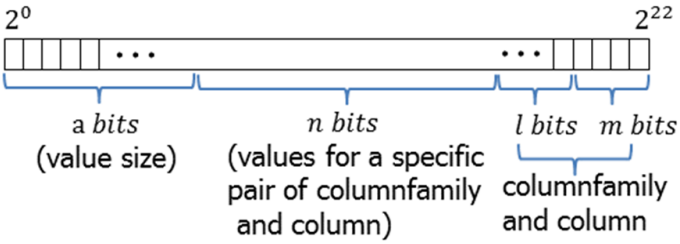

Fig. 3. Structure of address data mapping of HBC

23bits addresses of the DRAM when it accesses the DRAM. In the 23bits, $l$bits are used to identify the variations of column families and $m$bits are used to identify the variations of columns. Therefore $(l + m)$bits are used to identify the variations of the pairs of a column family and a column. Also $n$bits in 23bits are used to represent the maximum numbers of values that can be cached for a specific pair of a column family and a column and $a$bits in 23bits represent the maximum size of a value stored in an address field for a specific row of the DRAM (the size of values of different revisions for a specific address is included in the maximum size).

In this paper, the configuration of 23bits addresses of DRAM is shown above and the parameters of the number of bits and assignment to the DRAM can be adjusted according to the target application.

Figure 4 shows data mapping and the search of binary tree of HBC in the pattern in which the configuration is $l = 1$, $m = 2$, $n = 8$, and $a = 12$. The following subsections show the patterns where software stores data in HBC, HBC processes Set queries, and HBC processes Get/Scan queries.

1) When Software Stores Data in HBC: In this case, HBC needs configuration of column families of N_{cf} patterns and columns of N_c patterns before the system starts. HBC only caches data for the N_{col} patterns of the pairs of configured column families and columns. Software stores values of successive N_{row} patterns that is sorted by rows as a result of a scan operation for a specific pair of a column family and a column performed by software to the corresponding DRAM addresses of HBC whose $n$bits addresses in 23bits addresses are successive. The DRAM addresses are determined by the structure of addresses used in the data mapping of HBC shown in Section IV-A.

Figure 4 illustrates an example of the cache where successive 256 values from Row000 to Row255 are stored in DRAM addresses (23bits) of HBC whose $n$bits addresses are successive (256 addresses) respectively. This operation is performed for each

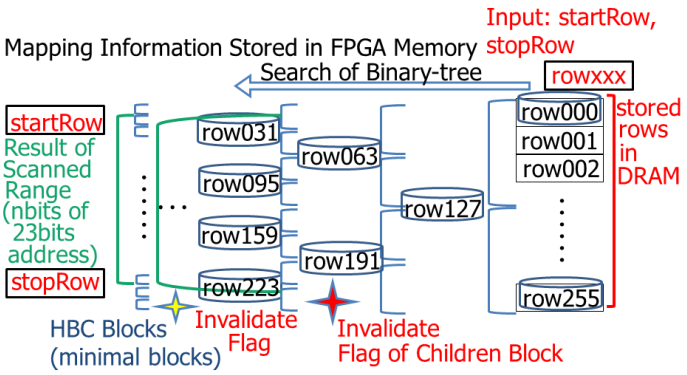

Fig. 4. Data mapping and search of binary tree of HBC

pair of a column family and a column that HBC manages in it respectively. Also to manage the revisions distinguished by timestamps, the values corresponding to a specific row are cached in addresses (2^a addresses) of DRAM that corresponds to the row.

When HBC stores values to its cache, it also stores mapping information of the cache in a data structure represented by blocks distinguished in a binary-tree manner using FPGA memory.

As Figure 4 illustrates, for specific pairs of a column family and a column that HBC supports for caching, 2^n row addresses (256addresses in this figure) are divided into 2 blocks in a step of the search of a binary tree from parents to children recursively. In these steps, only 2 rows that correspond to maximum and minimum addresses respectively in each block are stored as mapping information in FPGA memory. In this manner, the deepest blocks of the data structure represented by a binary tree are defined as minimum blocks which are indivisible. These minimum blocks are called HBC blocks. The number of addresses managed in a HBC block are represented in the following equation.

$$N_{blockadd} = 2^b \tag{3}$$

When updates occurred in the database, software performs updating of HBC cache in the DRAM and mapping information of the cache stored in the data structure represented by a binary tree in FPGA memory. These updates are performed from software to HBC via PCI Express mounted on the FPGA NIC. In the updates, software sends addresses of DRAM that will be updated, updated values, and addresses that are invalidated and HBC performs updating and invalidating according to the information that is received from software.

When HBC invalidates specific cached values, it sets an invalidation flag for each HBC block. Also parents blocks of the invalidated blocks are invalidated from HBC blocks recursively. On the other hand, when the invalidation is released, the invalidation flags are unset.

2) When HBC Receives Set Queries: In this case, to maintain the consistency of the cache, HBC performs search of the mapping information in the data structure represented by a binary tree and checks whether rows that are designated by Set operations are already stored in HBC.

If the rows are stored, it sets invalidation flags to the corresponding HBC blocks. HBC also transfers the Set operations to the software layer for processing regardless of whether the designated rows are stored in HBC.

3) When HBC Receives Get/Scan Queries: In this case, HBC receives Get/Scan operations from clients and searches its cached values for the requested data. HBC processes a Get operation as a Scan operation whose requested number of values is 1.

HBC first checks whether the pair of a column family and a column that corresponds to the given rows is cached in HBC. If it is not cached, HBC processes the query to be cache missed. Otherwise, HBC starts search of the data structure represented by a binary tree. In this case, for given startRow and stopRow respectively, it performs search of the binary tree by blocks from

parents to children to find whether startRow or stopRow are cached in the blocks recursively and finally HBC locates which HBC block includes the given startRow or stopRow respectively.

In the case where the value of startRow is larger compared to that of stopRow, the search reaches the invalidated blocks or it is confirmed that startRow or stopRow is not included in the range of cached blocks, HBC processes the query to be cache missed. If for both startRow and stopRow the search reaches HBC blocks without making the decision of cache miss, HBC acquires the range of addresses represented in n bits of the 23bits addresses of the DRAM, that are represented by the range of addresses between 2 HBC blocks that include the startRow and the stopRow respectively.

According to the acquired range of addresses, HBC generates the range of addresses for cached data in DRAM in the manner of the structure of an address used in the data mapping of HBC shown in Section IV-A and it performs successive read operations to DRAM for the acquired range of addresses by the unit of HBC blocks.

Then the read results are returned to clients from HBC. If the Get/Scan operations are missed in HBC, HBC transfers the Get/Scan operations to the software layer to be processed.

B. Pipeline Processing of HBC

HBC searches the binary tree of mapping information when it receives a Set query and checks whether the designated rows are cached in HBC or when it receives a Get/Scan query and it checks whether the query hits or misses in HBC.

In these cases, if the number of HBC blocks that are searched is considered to be N, the delay is represented by $O(\log_2 N)$ as steps of the binary search.

Also, when the query hits in HBC, the delay to access the DRAM is caused. To hide these delays, the input and output to/from clients of HBC are divided into several input/output stages and HBC forms a pipeline and can keep processing without stopping the following request packets. This method can prevent the declining of the throughput of responses.

C. Queries of HBC

This section shows the format of queries sent by clients and received in HBC.

The examples of queries are shown in Figure 5. The types of

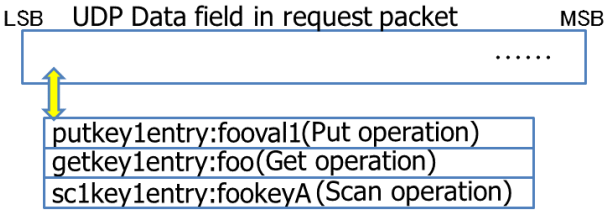

Fig. 5. Queries of HBC

operations to databases are represented by “put”, “get” and “sc1” (these types correspond to Set/Get/Scan operations respectively) and each of them is filled in the first 3Bytes of the UDP data region in a request UDP packet as strings.

If the operation is Set, the designated key name (e.g., “key1”), column family and column (e.g., “entry:foo”) and put value (e.g., “val1”) follow.

If the operation is Get, the designated key name (e.g., “key1”), column family and column (e.g., “entry:foo”) follow.

If the operation is Scan, the designated startRow (e.g., “key1”), column family and column (e.g., “entry:foo”) and stopRow (e.g., “keyA”) follow.

HBCMS interprets HBC queries into HBase queries to perform software operations.

D. HBCMS

In this section, software that cooperates with HBC and column-oriented stores is introduced. This software used in this paper is called HBCMS (HBC Management System). Figure 6 illustrates the data flow and processing of HBCMS. HBCMS

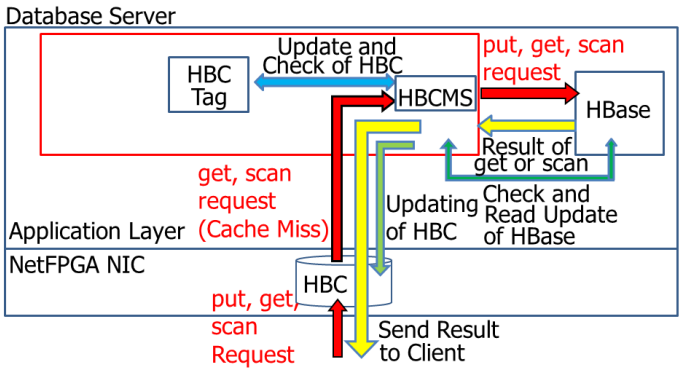

Fig. 6. Data flow and Processing of HBCMS

performs updating of cached data and its mapping information in HBC when clients send Set queries to the column-oriented databases and the data of databases that corresponds to stored data in HBC is updated, sending back the requested data to clients from the databases when the Get/Scan queries from clients are missed in HBC, updating cached data and mapping information in HBC when updates occurred in the databases, and storing scanned results generated by software in HBC when Scan queries from clients are missed in HBC.

HBCMS stores mapping information of HBC using the memory of the server machine (HBC tag shown in Figure 6). HBCMS accesses the mapping information of HBC stored in the server to determine the updates to HBC and then updates the stored data in HBC.

When clients send Set queries to the server, HBCMS sends Set requests to HBase and Set operations are performed by software. HBCMS also checks the mapping information of HBC in the server and if it finds that the data to be updated by Set queries is also stored in HBC, it updates the corresponding data in HBC and also updates the mapping information of HBC in the server. When updating of the databases occurred, HBCMS checks and reads updated data in the databases. HBCMS sends the updated values and mapping information of HBC to HBC and also updates the mapping information of HBC stored in the server.

When a Scan query is missed in HBC, HBCMS accesses column-oriented databases so that it processes the scan query and reads the scanned results. The results are sent back to clients via FPGA NIC. HBCMS keeps the scanned results until updating of HBC is finished. Then HBCMS accesses the mapping information of HBC stored in the server and checks whether the range of the scanned rows overlaps with the range of the rows stored in HBC. If it is not overlapped, HBCMS adds the scanned rows and the mapping information that corresponds to them to HBC. Until the update is completed, it sets invalidation flags to the added rows. It also updates the mapping information of HBC in the server. If the update is completed, the invalidation flags are unset. In this way, it can prevent sending invalid data back to the clients. If the range of the scanned rows is overlapped, the overlapped part of rows and mapping information in HBC are invalidated and the new scanned rows and the corresponding mapping information are added to HBC. Until the update is completed, HBCMS sets invalidation flags to the updated range of rows in HBC. It also updates the mapping information of HBC in the server. If updates of HBC and the mapping information in the server are finished, it unsets the invalidation flags. In this way, it can prevent sending invalid data back to the clients.

V. Implementation

A. Target Platform

In this paper, a client machine and a server machine that are directly connected via a 10GbE direct attached cable are used in the experiments. HBase is running on the server machine as a column-oriented store in a software layer. Software of the client generates HBase queries (Get, Set and Scan) and sends them to the server. A NetFPGA-10G NIC board [11] is mounted on the server via a PCI-Express Gen2 x8 interface as a 10GbE NIC. A prototype of HBC is implemented on the FPGA NIC on the server to accelerate the responses to the queries of column-oriented stores that are sent back to clients.

B. HBC

A prototype of HBC is implemented according to the design of HBC shown in section IV. The target application of HBC is HBase as a column-oriented store running on the server machine. The prototype of HBC performs the binary search to its cache according to Get/Scan queries from clients and can return requested values if sorted results of scan operations and the mapping information that corresponds to them are stored in HBC. Then if the queries hit in HBC, it returns the requested scanned results to the clients from the NIC rapidly. Verilog HDL is used as a hardware description language in the implementation. Some parts of designs released in NetFPGA Project [11] are also used.

The parameters mentioned in Section IV-A are set to $l = 1$, $m = 2$, $n = 8$, $a = 12$ and $b = 3$ and the maximum sizes of a row key, column family and column are set to 4Byte, 5Byte and 3Byte respectively. The number or versions of values in a row that can be cached in HBC for a specific pair of a column family and a column is set to 1.

In this paper, UDP/IP is deployed as a transport layer protocol for the communications between clients and the server to demonstrate raw performance of HBC.

C. HBCMS

A prototype of HBCMS is implemented according to the design of HBCMS shown in the section IV-D. The target application of HBCMS is HBase. The prototype of HBCMS receives queries from the client via HBC on the FPGA NIC sent to the software layer. HBCMS accesses HBase to perform the requested queries and can process Set/Get/Scan operations. If it receives a Set query, it makes HBase to perform a Set operation. If it receives a Get/Scan query, it makes HBase to perform a Get/Scan operation and reads the results from the database and sends them back to the client. Apache Thrift APIs [12] are used in the implementation.

VI. Evaluations

A. Evaluation Environment

Table I shows the server machine and the client machine used in experiments. HBC is implemented on the FPGA NIC on the server machine. The throughputs of Get/Scan operations are measured using these machines.

UDP/IP is used as a transport layer protocol. For the Get/Scan queries, the query format shown in Section IV-C is used. The row size is limited by up to 18Byte. For the Get query, the payload consists of an operation type (i.e., get), a row key, and a pair of a column family and a column. For the Scan query, the payload consists of an operation type (i.e., sc1), startRow, a pair of a column family and a column, and stopRow.

The result of the logic synthesis and place and route of the prototype of HBC is shown in Table II. The target device of the logic synthesis and place and route is Virtex-5 XC5VTX240T FPGA and Xilinx ISE 13.4 is used.

TABLE I. Evaluation Environment

	HBC		Software	
	Server	Client	Server	Client
CPU	Intel Core i5-4460	Intel Core i5-3470S	Intel Core i5-4460	Intel Core i5-2400
RAM	4GB	6GB	4GB	4GB
OS	CentOS 6.7	CentOS 6.7	CentOS 6.7	Ubuntu 14.04.3
NIC	NetFPGA 10G	NetFPGA 10G	NetFPGA 10G	Mellanox 10G

TABLE II. Result of the Logic Synthesis and Place and Route of the Prototype of HBC

Number of Slice LUTS of HBC	44,247
Utilization of Slice LUTS of HBC	29%
Number of Slice LUT-Flip Flop pairs of HBC	62,944
Utilization of Slice LUT-Flip Flop pairs of HBC	42%
Minimum period of HBC	9.092ns
Maximum frequency of HBC	109.987MHz

B. Evaluation Results

In this experiments, the throughput represents the number of Get/Scan operations processed by the server per a second (Ops/sec). A real processing throughput of Get/Scan queries depends on cache hit rate of HBC. Thus we measured the throughput when all the Get/Scan queries are hit in HBC. We also measured the throughput of the software (cooperation of original HBase and HBCMS) that corresponds to throughput when all the Get/Scan queries are missed in HBC and processed by HBCMS and HBase. We also measured the power consumption (W) of the overall hardware of the server machine.

1) Get Operations: We measured throughput when all the Get queries sent by the client to the server are missed in HBC and when all of them are hit in HBC respectively. When the hit rate of HBC is 0, the measured throughput in our proposed system is low. For example, about 2.5Kops/s throughput was reported in [13] for a read intensive workload.

Figure 7 shows the throughput of Get operations when a 18Byte value is returned per operation. Get queries that request a row are generated by the client. The client machine sends over 10 million request packets and receives the response from the server. The throughput (bits/s) is measured as 9.619 Gbits/s when the hit rate of HBC is 1.

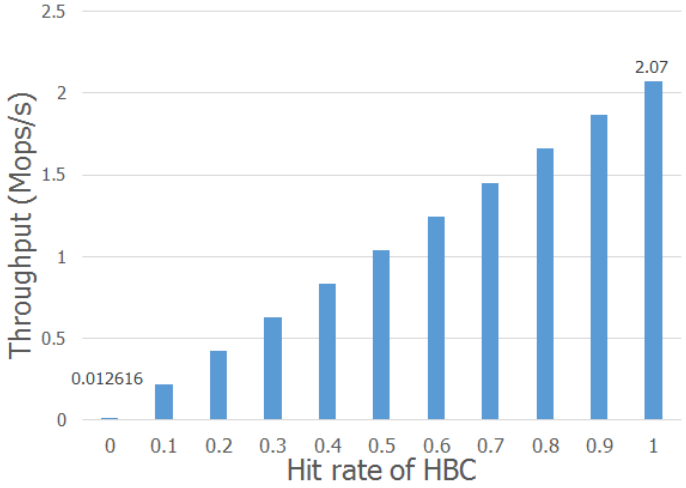

Fig. 7. Throughput of Get operations by HBC or software

2) Scan Operations: We measured throughput when all the Scan queries that the client sends to the server are missed in HBC and when all of them are hit in HBC respectively.

Figure 8 shows the throughput of Scan operations when 8 values (18Byte for each value) are returned per operation. Scan queries that request 8 rows for each query are generated by the client. The client machine sends over 10 million request packets and receives the response from the server. The patterns where the number of scanned rows is other than 8 are omitted since when the search of scanned range is completed and once sequential read of DRAM started, each value of rows is returned at the same

pace (clock cycles of HBC) regardless of the numbers of scanned rows. The network is saturated as the number of scanned rows increases. The throughput (bits/s) is measured as 9.815 Gbits/s when the hit rate of HBC is 1.

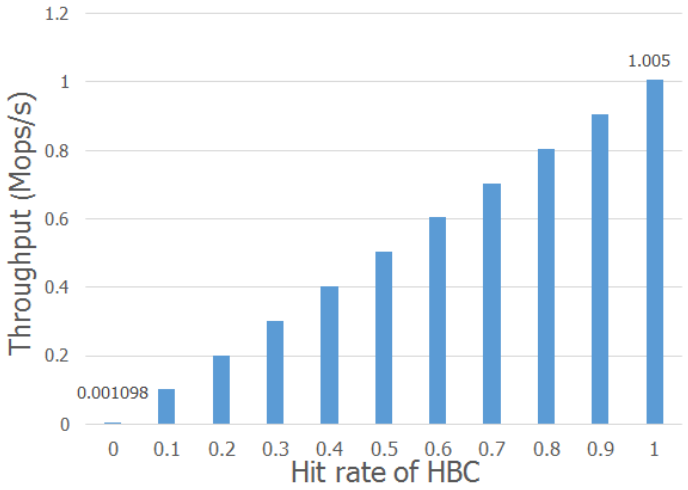

Fig. 8. Throughput of Scan operations by HBC or software

3) Power Consumption: We measured the power consumption of the overall hardware of the server machine when HBC is processing Scan operations and HBCMS and HBase are not working and when HBC is not running and responses to the Scan queries from the client are processed by HBCMS and HBase. Table III shows the power consumption of the overall hardware of the server machine for these 2 patterns.

TABLE III. POWER CONSUMPTION OF OVERALL HARDWARE OF THE SERVER

	HBC	Software
Power consumption(W)	71.6	92.1

C. Discussions

Figure 8 shows the throughput of Scan operations (8 values are read for each query) is 1.005Mops/s when the hit ratio of HBC is 1.

Figure 7 shows the throughput of Get operations is 2.07Mops/s when the hit ratio of HBC is 1. This value is about 2 times higher than that of Scan operations shown in Figure 8 and it is considered that the number of retrieved values in Get operations is lower than that of Scan operations and that caused higher throughput (Mops/s) of Get operations compared to Scan operations. On the contrary, when the hit ratio of HBC is 0 and all the Get/Scan queries from the client are processed by HBCMS and HBase, throughput of Get queries is higher than that of Scan queries. This implies the ratio of performance improvement of Scan operations by HBC compared to software is higher than that of Get operations and the proposed HBC can accelerate Scan operations efficiently.

A lot of works of accelerating Get operations of Key-value stores by software have been reported. Column-oriented stores also support Get operations in a key-value form. A Get operation performed by software is considered to have much higher throughput compared to a Scan operation performed by software when it reads values of the same size and multiple rows are read in the Scan operation. This indicates it is considered that acceleration of Scan operations of column-oriented stores by hardware is effective since there is room for more improvement in performance for Scan operations compared to that for Get operations in comparison with software.

The throughput (bits/s) of Get/Scan operations is near to 10Gbits/s, the maximum bandwidth of 10GbE although the utilization ratio of payload of a packet is not took into consideration. This indicates that HBC processing is not the bottleneck of the network processing of 10GbE and thus the performance of HBC can scale further if FPGA NIC boards whose network interfaces have more broad bandwidth can be deployed. However, the performance of HBC is limited by the size of the on-board DRAM implemented on the FPGA NIC as the hit ratio of HBC tends to be lowered as the cache size decreases.

Section VI-B3 shows HBC can reduce the power consumption up to 20%. This means deploying HBC can improve energy efficiency compared to deploying software without HBC.

VII. SUMMARY AND FUTURE WORK

In this paper, HBC is implemented on the FPGA NIC as a hardware cache to accelerate Get/Scan operations of column-oriented stores. HBC can perform rapid responses to Get/Scan queries from clients by processing scan operations in a hardware-based manner to cached values in DRAM on the FPGA NIC. HBCMS that processes queries by software when queries are missed in HBC and also updates the cache of HBC is proposed.

Experimental results show that HBC can achieve orders of magnitude higher throughput compared to that of software processing when queries are hit in it and the prototype of HBC achieved a significant improvement of performance (Mops/s) when it processed Scan queries and the hit ratio is 1 compared to that of software. The power consumption of the server can also be improved by deploying HBC. In the experiment, deploying the prototype of HBC achieved improvement of the power consumption of the server by about 20% when it performs Scan operations compared to that of software processing. Therefore performance per Watt is also improved. However, the hit ratio of HBC is limited by the cache size of HBC.

As future works, investigation of the relation between cache size and the hit ratio of HBC, applying real workload to the evaluation and evaluation of the Write performance of Set operations to HBC are planned to be addressed.

Acknowledgements This work was supported by JSPS KAKENHI Grant Number JP16H02816.

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] 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.

[6] B. Sukhwani, H. Min, M. Thoennes, P. Dube, B. Iyer, B. Brezzo, D. Dillenberger, and S. Asaad, "Database Analytics Acceleration Using FPGAs," in *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'12)*, Sep. 2012, pp. 411–420.

[7] S. R. Chalamalasetti, K. Lim, M. Wright, A. AuYoung, P. Ranganathan, and M. Margala, "An FPGA Memcached Appliance," in *Proceedings of the International Symposium on Field Programmable Gate Arrays (FPGA'13)*, Feb 2013, pp. 245–254.

[8] M. Blott, K. Karras, L. Liu, K. Vissers, J. Baer, and Z. Istvan, "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.

[9] 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.

[10] K. Lim, D. Meisner, A. G. Saidi, P. Ranganathan, and T. F. Wenisch, "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.

[11] "NetFPGA Project," http://netfpga.org/.

[12] *"Apache Thrift"*, http://thrift.apache.org/.

[13] T. Rabl, M. Sadoghi, H.-A. Jacobsen, S. Gomez-Villamor, V. Muntes-Mulero, and S. Mankovskii, "Solving Big Data Challenges for Enterprise Application Performance Management," in *Proceedings of the International Conference on Very Large Data Bases (VLDB'12)*, Aug. 2012.