Hardware-based Hash Functions for Network Applications

Fumito Yamaguchi
Graduate School of Science and Technology
Keio University
Yokohama, Japan
yamaguchi@west.sd.keio.ac.jp

Hiroaki Nishi
Faculty of Science and Technology
Keio University
Yokohama, Japan
west@sd.keio.ac.jp

***Abstract*— For rich network services, it is indispensable to reconstruct TCP stream in the middle of the network. In this reconstruction function, managing a large number of streams is a crucial as an embedded hardware for achieving high-throughput processing. In this case, hashing is a well-used solution as an ID key of TCP streams. Since CRC hashes have simple tree structure of XOR logics and are perfectly composed of combination logics, they can easily be hardware implemented and enables low-latency calculation of the hash values. Therefore, CRC hash has been used in network applications where low-latency and high speed operations are necessary. Some of their usages include data identification, load balancing, encryption, and checksum. Although CRC hashes may be well suited for hardware implementation, it is not the top choice when collision rates and distribution of hash values are considered. For some network workloads, CRC hashing shows very uneven distribution of hash values and can lead to decrease in the overall performance of network systems. In this study, two hashes, Jenkins hash and MurmurHash, are implemented on a FPGA as TCP connection manager and evaluated for their aptitude for hashing used in hardware-based network applications.**

Keywords-Hashing; FPGA; Hardware Implementation; TCP Connection Manager

I. INTRODUCTION

Many services are provided on the Internet, and the services become diverse and complex according to the demand of rich network services. Big data is an idea to utilize massive data as services for providing the rich services. Network traffic is a typical resource of big data resource and several studies are conducted to mine new network services. In these services, TCP stream reconstruction in network devices on the middle of the Internet is a challenging but anticipated technology.

The study of Cisco fog computing [1] pointed out that cloud services have a problem for latency-sensitive applications, which require nodes in the vicinity to meet their delay requirements. An emerging wave of Internet deployments, most notably the Internet of Things (IoTs), requires mobility support and geo-distribution in addition to location awareness and low latency. It is preferable for the nodes of fog computing to handle the network streams and contents directly in the middle of the Internet to meet the requirements because the contents of stream can define the distribution and routing by reflecting dynamically changing needs of the contents for enabling low-latency applications.

In the idea of Internet Open Innovation Platform (IOIP) [2], we implemented Service-oriented Router (SoR) as a testbed of IOIP core device. SoR [3][4] is a router architecture that integrates a database and packet stream analysis for providing rich services from the network traffic. The SoR observes the internet traffic, analyzes the packets for payloads according to user-defined queries and stores the results into a database. The search queries are described by using SQL like query language given by users to capture packets. The ability of analyzing packet streams up to layer 7, namely the capability of deep packet inspection (DPI), makes the SoR unique in the field. However, for this upper-layer capability, the SoR has to manage TCP connections, which is not necessary in a standard router.

For achieving a function of TCP stream handling in the middle of the Internet as fog computing and IOIP, TCP connection manager is an indispensable hardware for wire-rate TCP stream processing of DPI, which uses the contents in the packet stream. In this reconstruction process, managing a large number of streams is a crucial as an embedded hardware for achieving high-throughput discarding free processing. In this case, hashing function is a well-used solution as an ID key of TCP streams. According to our prior evaluation of the network traffic captured in the edge router connected to the Japan's largest academic Internet backbone, a link interface transmits about 70,000 TCP connections simultaneously in average if the expiration time of TCP stream is set to 30 seconds. TCP stream manager has to manage this TCP stream table and support 1.5 million/s accesses in 10-Gbps network. This means effective and low-cost hashing hardware are required for managing TCP streams.

TCP stream reconstruction, especially focusing on hash function, is not well discussed. In software design, Snort [5], a network intrusion detection system (NIDS), and TCP analysis application such as Wireshark [6] does not support stream reconstruction in the middle of the network. As hardware-based TCP stream reconstruction, Sugawara, et. al. [7] and Mythili et. al. [8] discussed the hashed key of TCP stream segments. However, these discussions did not mention what kinds of hash function were used. IBM PowerEN is a network processor and has a hardware acceleration co-processor of XML, string

978-1-4799-2084-6/13/$31.00 ©2013 IEEE

matching, encryption/decryption, zip encoding/decoding for network streams [9]. However, the user program owes to handle the key of streams by designing a hash function on their own. This study is the first approach to compare hash functions in the application of network devices.

II. HASH FUNCTIONS

This section briefly surveys the current research on hashing used in network application and summarizes the additional requirements for hardware-based hashing used in TCP connection management on SoRs.

Hashing is a method of creating a fixed length value from the input data. Since hash values are shorter in length than the input data, hash values are usually considered as a "digest" of the input data. Some of their usages include data storage, load balancing, encryption, data identification, and error detection checksum in data transactions. Most hashes are fixed length values, and are generated from the input data by a series of calculations.

The characteristics of hash functions largely differ among functions. One of the most important factors that characterize hash functions is the evenness of hash value distribution. An ideal hash would distribute hash in a completely random manner, and this would result in an even distribution of hash values along the codomain of possible hash values. This characteristic is the outcome of how calculations are done in a hash function.

Hash function usually consists of a combination of very simple calculations. Some of the commonly used include simple arithmetic calculation, logic operation, and bit shift. There has been intensive research on crypto hashes optimized for software implementation and calculation on a general-purpose CPU, but little on hardware non-crypto hash [10][11][12]. The study by Jiang et al. analyzed the hashing method used for network application, but gave no considerations for hardware-implemented applications [13]. For hashes to be implemented on hardware and achieve satisfactory performance, the following criteria need consideration.

A. Low Latency

For hashes implemented on hardware, throughput of the hash depends on the delay caused by the circuitry. Although hash function that require multiple clock cycles to complete its calculation can sustain throughput when calculations can be pipelined, application that require immediate results, such as in case of a table lookup, must use hashes that complete calculation within a single clock cycle. Even for hash function that finishes its calculation within a single clock cycle, circuitry and logic gates pose some delay. To maximize the throughput of hash functions, hashes must have calculation schemes that can be achieved with smaller circuit delay. For example, since a multiplication circuit is far more complex than a full-adder circuit, it is better for the hashes to exclude any multiplication in its function. To minimize the delay caused by hash functions implemented on FPGAs and ASICs, two basic fundamental properties must be attained.

- Single clock calculation.
- Small delay on critical path.

In this study, hash functions that satisfy these requirements are chosen for evaluation.

B. Circuit Size

Having a small circuit size is important for hardware implemented circuits. Larger circuits lead to increased fabrication costs and power consumption due to leak currents. Smaller circuits are beneficial when multiple hashes need to be implemented, such as in a parallel processing engine. Therefore, the circuit size of hash functions on FPGAs and ASICs are worth considering.

C. Even Distribution

An ideal hash function would generate hash values that are evenly distributed across codomain. For this to be accomplished, the hash function must generate a completely different hash even if there is only a slight change in the input data. To determine if a hash has even distribution of hash values, avalanche testing, which will be described in detail in Section III, is performed.

D. Low Collision Rate

A hash collision occurs when two or more different input keys generate a same hash value. Since hashes are simply a "digest" of input keys, hash collisions are unavoidable and occur in a probabilistic manner. Hashes that give uneven distribution of hash values are, in most cases, prone to hash collisions; the number of input keys that are represented by hash values in the "congested" area of the uneven distribution would be larger and have a larger chance of colliding. Hash collision causes decrease in the performance of application that uses hashing, since the application needs to take measures to detect and avoid hash collisions.

These four criteria must carefully be considered for a low-latency hash in hardware-based network applications, and will be given consideration throughout this study. Given these requirements, the following three hash functions are chosen for evaluation.

- CRC Hash
- Jenkins Hash (Lookup3)
- MurmurHash

The three hashes are chosen because their calculations are very simple and conclude within a single clock cycle. Jenkins hash and MurmurHash also have good reputation for its low collision rates and even distribution in software implementation. CRC hash is also analyzed for comparison with the two hashes.

1) CRC Hash: Cycle Redundancy Check, or CRC, was originally designed to detect continuous errors in data transitions and is commonly used as checksums [14]. For example, the FCS field of the Ethernet frame is a 32bit CRC hash that is used for detecting bit errors. The values are well agitated and it is suited for not only checksums but also keys for data lookup. CRC hash can be generated by using XOR gates and flip-flops; therefore, hardware implementation is

easily accomplished. Despite having these advantages, CRC hash has an un-even distribution of hash values in some workloads, which can be disadvantageous in some applications. In this study, a CRC-32 hash was used to analyze the behavior of CRC hash using real-life internet traffic.

2) Jenkins Hash: Jenkins hash is a series of hash function proposed by Bob Jenkins [15][16]. In 2009, the Lookup3 hash function was introduced, followed by the SpookyHash in 2011. The Lookup3 function is used for this study, for its good avalanche beheivior. The Lookup3 hash generates hashes by using a combination of bit shift, addition, OR operation, and XOR operation.

3) MurmurHash: MurmurHash was introduced by Austin Appleby and consists of multiplication, XOR operation, and bit shift [17]. Among the several variations of MurmurHash, MurmurHash2 was used for this study. The hash calculation is very simple, consisting of only a mix and a finalization process. Along with its simple structure, MurmurHash has other benefits, such as even distribution, low collision rates, and achieving avalanche.

III. Evaluation Methods

The evaluation of hashes is done in two approaches for this study. The first approach is to test the characteristics of the hash functions itself. The avalanche behavior is measured to show how well the hash values are distributed, and circuit delay for how fast the hash functions are.

The second approach is to embed the hash functions into a network application to see how the overall performance of the changes when different hashes are utilized. Each hash is implemented into the TCP connection manager and compared on how different hash functions can affect the manager's performance. Real-life internet traffic is used for the evaluation.

The focus of this paper is the implementation and evaluation of low-latency non-cryptographic hash for network applications, hence the hash's tolerance to security threats need not be considered.

A. Avalanche Effect

To test whether a hash function has an even distribution of hash values, the most accurate way would be to calculate hash values of all possible input keys and analyze the distribution of the hash values. However, since the total number of the input keys is too large, this ideal analysis is not realistic in most cases. A popular analysis to test the distribution of hash values is the avalanche testing.

The avalanche effect shows how each bit of the input data affects the output hash value. This analysis shows the independence of each of the output bits. If all the output bits of the hash function show complete independence from the input bits, this indirectly shows the evenness of the distribution of hash values; output bits that are correlated to a particular bit result in uneven distribution.

The avalanche behavior is obtained by calculating the probability of each of the output bits changing its state when the input bit changes. This calculation is done for all the combination of the input and output bits. For hash function that achieve avalanche, each of the output bits would change its state with a 50% chance. A mathematical representation of the probability of bit flip P, when achieving avalanche is given as follows.

$$P(\textit{input bit } i \textit{ change}/\textit{output bit } j \textit{ change}) = 0.5, \quad \textit{for all } i, j$$

The evenness of the avalanche behavior can be determined by the largest bias of the state-change probability. The bias is the difference of the state-change probability from the ideal 50%. Although less bias does not directly relate to the evenness of hash value distribution, it can be used as an index of how well hash values change amid a slight bit change in the input key.

B. Circuit Delay

The throughput of a hardware-implemented device is dependent on how fast the circuitry can operate, or in other words, how fast the maximum clock frequency is. The path with the largest delay inside a circuit, usually called the critical path, determines the maximum operating frequency. Hardware circuits must operate at clock frequencies less than that of the inversed delay of the critical path.

The critical path delay of each hardware-implemented hash is examined to see how fast each hash can perform. For this analysis, all three hashes are designed for implementation into a FPGA device. Verilog HDL is used for coding the hash functions. Simulations are done with Cadence NC-Verilog LDV and Simvision 06.20. Circuit timing analyses and synthesis are done on ISE Design Suite 14.2. A FPGA evaluation device, LogicBench with a Xilinx Vertex5 XC5VLX330T, is used as the target device.

C. TCP Connection Manager

TCP connection management is chosen as an example of a network application. The TCP connection manager manages the connection state of TCP connections. The connection management is growing in necessity from recent trend in the upper-layer analysis of network traffic. The architecture of the TCP connection manager is shown in Figure 1.

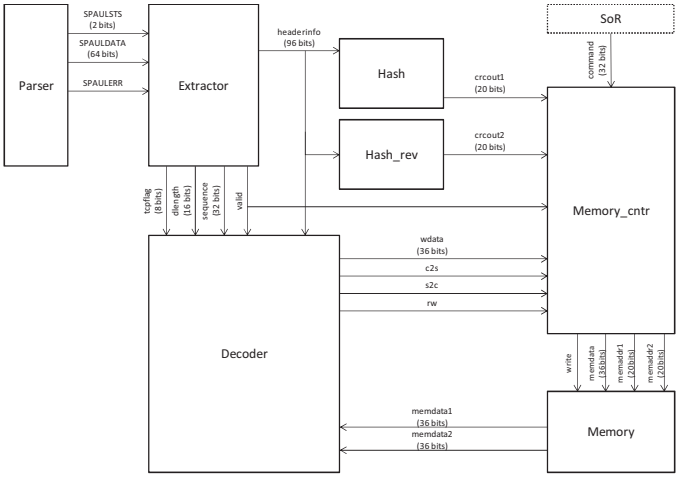

Figure 1. TCP Connection Manager Architecture

The TCP connection management operation is performed in the following manner.

1) Header Info Extraction: The header information from the TCP/IP header is extracted by the parser and the extractor. The parser function formats the header information and payload data. This parser function is provided by the LogicBench ethernet board, and has a maximum thourghput of 1Gbps. The extractor extracts the 4 tuples (Src IP, Dst IP, Src Port, Dst Port), and the sequence number, which are required for TCP connection management.

2) Hashing: Hashing is done on the 4 tuples to distinguish TCP connections. Packets with the same hased value are considerd as belonging to the same TCP connection. The hash values are used as the address of the memory, which is used to register TCP connection states.

3) Decoding: Decoding is done for every incomming packets. The decoder determines the TCP connection state based on information from the memory and the extractor. The decoder also updates the connection state according to the received packets. The flowchart of the decoder is shown in Figure 2.

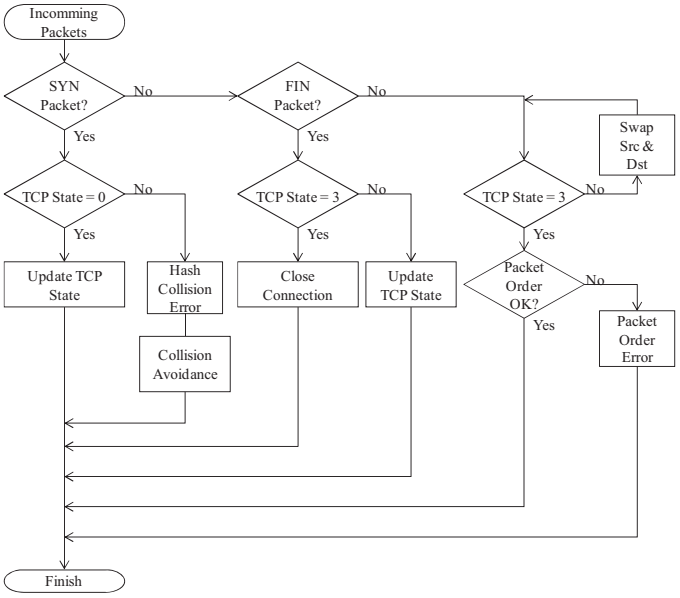

Figure 2. Flowchart of Decoder

A software simulator written in the C language is used to obtain results, since there are challenges in collecting and analyzing result data from the FPGA device. The simulator measures the total number of memory access. Lower memory access shows even distribution and less collision ratio of hash functions; the memory controller performs linear probing in an event of a hash collision and therefore increases the memory access.

The analysis for this study is conducted using real-life Internet traffic obtained from the WIDE network. Each workload contains internet traces over a span of 15 minutes. The TCP connection manager and the hash functions are implemented in the LogicBench FPGA device, and the outputs are observed with an Agilent 16823A logic analyzer.

Although not used in the TCP connection manager, the hashes are also tested as the hashing used for a BloomFilter. A BloomFiter [18] is a data structure to check whether a data is a member of a dataset, and is widely used in network applications. It can also be beneficial for the TCP stream manager. To test the collision rate of the hashes used for the BloomFilter, the amount of unique entries remaining after hashing is compared among the three hashes. 40,000 IP addresses obtained from the WIDE network are used for evaluation. The hash values are fixed in length at 20bits.

IV. Results

A. Avalanche Effect

The avalanche diagram of CRC hash, Jenkins hash, and MurmurHash is shown in Figure 3. The y-axis is the output bit, and the x-axis is the input bit. Each point in the diagram shows the probability of the corresponding output bit to change its state when the input bit changes. The probability is shown in colors, with green being 50%, red 0%, and white 100% for the CRC hash, and green 50%, red 30%, and white 70% for the Jenkins hash and MurmurHash to emphasize the results.

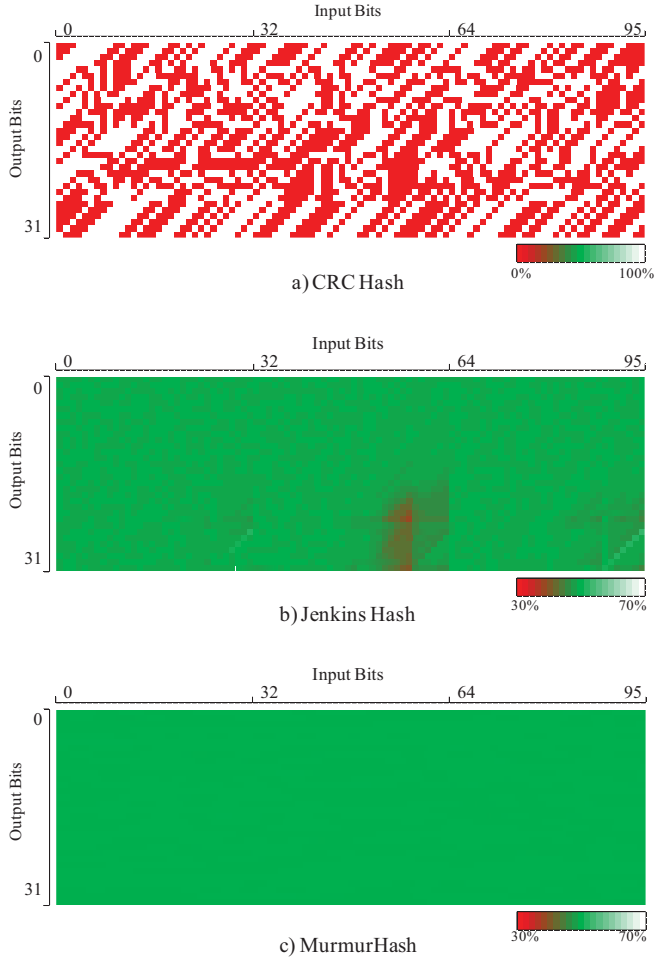

Figure 3. Avalanche Diagram

The avalanche diagram of CRC hash shows that the state-change probability is composed of only 0% and 100%. This characteristic implies that there is 100% chance that at least one bit of the output hash would change its state when a single input bit changes. This is the reason why CRC hashing is used in checksums. Although this is suited for use in checksums, CRC hash is weak to the resistance of hash collisions among input keys that are only slightly different.

The avalanche behavior of Jenkins hash and MurmurHash are identical with most of the bits having a flip probability of around 50%. TABLE I shows the maximum bias from the 50% flip probability.

TABLE I. MAXIMUM BIAS OF HASH FUNCTIONS

Hash	CRC	Jenkins	Murmur
Bias	-50.0% +50.0%	-12% +3%	-0.21% +0.16%

The bias of hashes are small enough to say that both Jenkins hash and MurmurHash achieve avalanche. These results show that, Jenkins hash and MurmurHash have the ability of generating completely different hashes regardless of the given workload. This contributes to an even distribution of hash values under any circumstances. On the other hand, CRC hash may have problem of creating an even distribution of hashes for workloads having many similar keys. This may lead to a very uneven distribution of hash values and increase the hash collision, which is evaluated in the latter half of this section.

B. *Circuit Delay*

The circuit delay of each hash function when implemented on a FPGA device is given in TABLE II.

TABLE II. CIRCUIT DELAY OF HASH FUNCTIONS

Hash	Maximum Delay
CRC	7.1ns (3.4ns logic, 3.7ns route)
Jenkins	22.4ns (12.9ns logic, 9.5ns route)
Murmur	35.6ns (31.5ns logic, 4.1ns route)

The results show that CRC hash have the smallest circuit delay and is around 1/3rd and 1/5th of that of Jenkins hash and MurmurHash respectively. When hashes are used alone, CRC hash can operate at higher clock frequencies, which gives it an advantage in high-speed applications.

When used with other modules, however, the delay of hash functions itself may be small enough that it is cancelled off by the delay of the entire system. This is the case where there are many modules in parallel that consume more time than the hash function. In the case of the TCP connection manager, the hash functions are in serial of other process, and therefore the increase in the delay of hash function directly affected the overall delay of the system. The overall delay was 97.7ns and 120.3ns for CRC hash and Jenkins hash respectively. Likewise, for application that requires a large number of hash calculations in series, the small delay of CRC hash may be beneficial enough to be chosen as the selective.

C. *TCP Connection Management*

Figure 4 shows the outputs of the TCP connection manager captured on the logic analyzer. Crcout1 and 2 show the hash outoputs and the valid signal shows the TCP connection state.

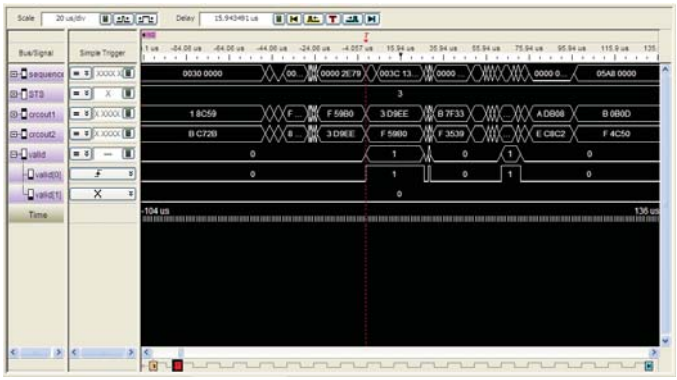

Figure 4. Output Data of TCP Connection Manager

The total number of memory access for the TCP connection manager is shown in Table III. Darker shading shows better results.

TABLE III. TOTAL MEMORY ACCESS

Workload	CRC	Jenkins	Murmur
200904020400	31694	31689	31387
200904020215	2256101	391449	391905
200904020500	694543	267523	267760
200904020730	91229	99130	98928
200904020200	48647	48388	48089
200904021230	109547	95828	96319
200904022215	129020	113313	113182
200904021245	63310	63280	63708
200904020145	42555	42597	42771
200904022345	180958	142337	141133
200904020130	1004644	283942	285021
200904021215	390852	199243	198386

Jenkins hash and MurmurHash show similar results with the smallest number of memory accesses in five of twelve workloads. The beneficial difference of the two hashes is close to zero; the average relative difference of reduction in memory access is a mere 0.45%.

These results also show how infrequent hash collisions are. The excess memory access are only caused by the hash collision avoidance, therefore smaller number of memory access implies that there are less hash collisions.

Although CRC hash performs the best in two workloads, their reduction in the memory access is only 8.9% at the most. On the other hand, workloads *200904020215* and

200904020130 have a significant reduction when Jenkins hash and MurmurHash is utilized. Both hashes achieve a memory access reduction of around 83% and 72% for the two workloads. The average reduction of memory access among all of the workloads is 25.2% for the Jenkins hash and 25.3% for the MurmurHash. This reduction is far more than the performance improvements that CRC hash has in two workloads. Therefore, it is fair to say that Jenkins hash and MurmurHash bring sufficient performance improvements to the TCP connection manager.

The remaining unique entries of the BloomFilter after hashing are shown in TABLE IV. TABLE V shows the loss rates of hash entries.

TABLE IV. REMAINING CACHE ENTRIES

Workload	Original	CRC	Jenkins	Murmur
l2res	1084	973	1074	1078
l5res	649	608	642	647
l10res	557	521	554	555

TABLE V. LOSS RATE

Workload	CRC	Jenkins	Murmur
l2res	10.2%	0.9%	0.6%
l5res	6.3%	1.1%	0.3%
l10res	6.5%	0.5%	0.4%

Jenkins hash and MurmurHash have a loss rate of less than 1% in most cases. These results show that the hash functions with even distribution is crucial in application where the retention of entries after hashing is important, such as BloomFilters. In the case of Jenkins hash and MurmurHash, the two hashes maintain 99% of the unique cache entries even after hashing, and therefore is suited for hash functions to be used in caches that use BloomFilters.

The four evaluation tests showed that Jenkins hash and MurmurHash have better results in most of TCP connection management and BloomFilter performance evaluations. If this performance benefit does not get cancelled off from the reduced maximum operating speed of the Jenkins and MurmurHash, the hashes that are used in network applications can effectively be replaced to achieve better performance.

V. CONCLUSION

CRC hash, Jenkins hash, and MurmurHash were evaluated for its aptitude in hardware-implemented TCP connection manager for the SoR. The hashes were implemented on a FPGA device, and evaluated for its circuit delay and performance in TCP connection management and BloomFilter. CRC hash had the smallest circuit delay compared with the other two hashes. Jenkins hash and MurmurHash showed better results than CRC hash for most of the TCP connection management and BloomFilter performance. From the evaluation results, it can be concluded that the replacement of CRC hash with Jenkins hash and MurmurHash is effective for the network application, such as a TCP stream manager.

ACKNOWLEDGMENT

This work was partially supported by Funds for integrated promotion of social system reform and research and development, MEXT, Japan, by Low Carbon Technology Research and Development Program for "Practical Study on Energy Management to Reduce CO2 emissions from University Campuses" from Ministry of the Environment, Japan and by MEXT/JSPS KAKENHI Grant (B) Number 24360230 and 25280033.

REFERENCES

[1] F. Bonomi, R. Milito, J. Zhu, S. Addepalli, “Fog computing and its role in the internet of things,” In Proceedings of the first edition of the MCC workshop on Mobile cloud computing (MCC '12). ACM, New York, NY, USA, pp. 13-16.

[2] www.openinter.net

[3] K. Inoue, D. Akashi, M. Koibuchi, H. Kawashima, H. Nishi, “Semantic Router using Data Stream to Enrich Services,” 3rd International Conference on Future Internet Technologies, pp. 20-23, June 2008.

[4] K. Takagiwa, R. Kubo, S. Ishida, K. Inoue, H. Nishi, “Feasibility Study of Service-oriented Architecture for Smart Grid Communications”, 22nd IEEE International Symposium on Industrial Electronics, TF-006505, 2013 (to be appeared)

[5] http://www.snort.org/

[6] http://www.wireshark.org/

[7] Y. Sugawara, M. Inaba, and K. Hiraki, “High-speed and Memory Efficient TCP Stream Scanning Using FPGA,” in Proc. International Conference on Field Programmable Logic and Applications, Aug 2005.

[8] Mythili Vutukuru, Hari Balakrishnan, Vern Paxson, “Efficient and Robust TCP Stream Normalization,” In Proceeding of the 2008 IEEE Symposium on Security and Privacy, pp. 96-110, 2008”

[9] A. Krishna, T. Heil, N. Lindberg, F. Toussi, S. VanderWiel. “Hardware acceleration in the IBM PowerEN processor: architecture and performance,” In Proceedings of the 21st international conference on Parallel architectures and compilation techniques (PACT '12). ACM, New York, NY, USA, pp. 389-400.

[10] S. Bakhtiari, R. Safavi-Naini, J. Pieprzyk, “Cryptographic Hash Functions: A Survey,” Technical Report pp. 95-09, Department of Computer Science, University of Wollongong, July 1995

[11] S. Al-Kuwari, J.H. Davenport, R.J. Bradford, “Cryptographic Hash Functions: Recent Design Trends and Security Notions,” IACR Cryptology ePrint Archive, 2011, pp. 565-565.

[12] R. Purohit, U. Mishra, A. Bnasal, “A Survey on Recent Cryptographic Hash Function Designs”, International Journal of Emerging Treands & Technology in Computer Science, vol. 2, issue 1, pp. 117-122, January - February 2013

[13] P. Jiang, J. Liu; Z. Qin, “On hashing techniques in networking systems,” Information Networking and Automation (ICINA), 2010 International Conference on , vol.2, no., pp. V2-444,V2-449, 18-19 October 2010

[14] W.W. Peterson, D.T. Brown, “Cyclic Codes for Error Detection,” Proceedings of the IRE , vol.49, no.1, pp. 228-235, January 1961.

[15] B. Jenkins, “Algorithm alley: Hash functions,” Dr. Dobb's Journal of Software Tools, 22(9), September 1997.

[16] B. Jenkins, “A Hash Function for Hash Table Lookup”, http://www.burtleburtle.net/bob/hash/doobs.html

[17] A. Appleby, “MurmurHash,” https://sites.google.com/site/murmurhash/

[18] B. Bloom, “Space/time trade-offs in hash coding with allowable errors,” Communications of the ACM 13, July 1970, pp. 422-426.