2019

In storage process, the next generation of storage system

Dongyang Li
University of Rhode Island, lidongyang@ele.uri.edu

PROCESSING IN STORAGE, THE NEXT GENERATION OF STORAGE SYSTEM

BY

DONGYANG LI

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN

ELECTRICAL ENGINEERING

UNIVERSITY OF RHODE ISLAND

2019

DOCTOR OF PHILOSOPHY DISSERTATION

OF

DONGYANG LI

APPROVED:

Dissertation Committee:

Major Professor Qing Yang

Jien-Chung Lo

Lutz Hamel

Manbir Sodhi

Nasser H. Zawia

DEAN OF THE GRADUATE SCHOOL

UNIVERSITY OF RHODE ISLAND

2019

ABSTRACT

In conventional computer systems, software relies on the CPU to handle the process applications and assign computation tasks to heterogeneous accelerators such as GPU, TPU and FPGA. It requires the CPU to fetch data out of the storage device and move the data to the heterogeneous accelerators. After the accelerators complete computation tasks, the results are flushed to the main memory of the host server for software applications. In this architecture, the heterogeneous accelerators are located far away from the storage device. There are data movements on the system bus (NVM-express/PCI-express), which requires a lot of transmission time and bus bandwidth. When data move back and forth on the storage data bus, it decreases the overall performance of the storage system.

This dissertation presents the in-storage processing (ISP) architecture that offloads the computation tasks into the storage device. The proposed ISP architecture eliminates the back and forth data movements on the system bus. It only delivers the computation results to the host memory, saving the storage bus bandwidth. The ISP uses FPGA as a data processing unit to process computation tasks in real-time. Due to the parallel and pipeline architecture of the FPGA implementation, the ISP architecture processes data in short latency, and it has minimal effects on the data flow of the original storage system.

In this dissertation, we proposed four ISP applications. The first ISP application is the Hardware Object Deserialization in SSD (HODS), which is designed to tailor to the high-speed data conversion inside the storage device. The HODS shows visible differences compared to software object deserialization regarding application execution time while running Matlab, 3D modeling, and other scientific computations. The second ISP application is called the CISC: Coordinating Intelligent SSD and CPU. It speeds up the Minimum Spanning Tree (MST) applications

in graph processing. The CISC coordinates the computing power inside SSD storage with the host CPU cores. It outperforms the traditional software MST by 35%. The third application speeds up the data fingerprint computation inside the storage device. By pipelining multi data computation units, the proposed architecture processes the Rabin fingerprint computation in wire speed of the storage data bus transmission. The scheme is extensible to other types of fingerprint/CRC computations and readily applicable to primary storage and caches in hybrid storage systems. The fourth application is data deduplication. It eliminates duplicate date inside the storage and provides at least six times speedup in throughput over software.

The proposed ISP applications in this dissertation prove the concept of computational storage. In the future, more compute-intensive tasks can be deployed into the storage device instead of processing in the CPU or heterogeneous accelerators (GPU, TPU/FPGA). The ISP is extensible to the primary storage and applicable for the next generation of the storage system.

ACKNOWLEDGMENTS

There are so many people to thank during my Ph.D. study at URI. So many have made my stay here productive and pleasant. I will try to cover all the bases without long-winded words.

Foremost, I would like to thank my advisor and mentor, Dr. Ken (Qing) Yang, for his guidance, encouragement, and inspiration over the past seven years. Dr. Yang has introduced me to the wonderland of research. He has helped me thrive in academia and life. He has taught me crucial skills in writing and presenting and, more importantly, the ways of doing research, which benefits throughout my life. His support has been essential to my success in the Ph.D. program at URI; it paved the way to this dissertation.

I would like to thank Dr. Jien-Chung Lo for providing me with constructive suggestions and feedback during my academic journey. I would like to thank Dr. Lutz Hamel from the Department of Computer Sciences, for introducing me to the machine learning algorithm. Much appreciations go to Dr. Manbir Sodhi, who has advised me on my dissertation, and Dr. Haibo He, the department chair, who has encouraged me and given me insightful suggestions on my research and future career. I sincerely thank them all for their support and feedback to improve this dissertation.

I would like to thank my collaborators, Jing Yang and Shuyi Pei. We have worked on the architecture of memory and storage system. They have introduced me to the FPGA, kernel driver and software developing. They have also advised me on writing and revising my manuscripts. I sincerely thank them for their support and help during my study at URI.

I would also like to thank many professors and staffs in URI ECBE department. I learn from many of them via different courses, which equipped me with

knowledge to tackle problems in this dissertation and my future career. The department staffs were very helpful, especially Meredith Leach Sanders and Lisa Pratt. They helped me to deal with paperwork and showed me concrete guidance of academic affairs.

I would like to express my thanks to my girlfriend Bing Han who is the Ph.D. student of Texas A&M University. She is always there cheering me up, and standing by me through good times and bad.

Finally, none of these would have been possible without the patience and support of my family. I would like to express my heartfelt thanks to my family. My parents raised and educated me with their unconditional love, and supported me to complete this degree. I regret to have less time to accompany with my family during my Ph.D. study. I will cherish everyone who appears in my life.

PREFACE

This dissertation is organized in the manuscript format. Particularly, there are four manuscript chapters. A brief introduction of the manuscripts are as follows:

Manuscript 1: Dongyang Li, Fei Wu, Yang Weng, Qing Yang, Changsheng Xie, "HODS: Hardware Object Deserialization Inside SSD Storage," 2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), Boulder, CO, USA, 2018, pp. 157-164.

Manuscript 2: Dongyang Li, Weijun Li, Qing Yang, "CISC: Coordinating Intelligent SSD and CPU to Speedup Graph Processing," 2018 17th International Symposium on Parallel and Distributed Computing (ISPDC), Geneva, Switzerland, 2018, pp. 149-156.

Manuscript 3: Dongyang Li, Qingbo Wang, Cyril Guyot, Ashwin Narasimha, Dejan Vucinic, Zvonimir Bandic, Qing Yang, "A Parallel and Pipelined Architecture for Accelerating Fingerprint Computation in High Throughput Data Storages," 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), Vancouver, BC, Canada, 2015, pp. 203-206.

Manuscript 4: Dongyang Li, Qingbo Wang, Cyril Guyot, Ashwin Narasimha, Dejan Vucinic, Zvonimir Bandic, Qing Yang, "Hardware accelerator for similarity based data dedupe", 2015 IEEE International Conference on Networking, Architecture and Storage (NAS), Boston, MA, USA, 2015, pp 224-232

TABLE OF CONTENTS

Page

Page

LIST OF FIGURES

Figure **Page**

Figure **Page**

LIST OF TABLES

MANUSCRIPT 1

HODS: Hardware Object Deserialization in side SSD Storage

by

Dongyang Li[1], Fei Wu[2], Yang Weng[3], Qing Yang[4], Changsheng Xie[5]

is published in the 26th Annual International Symposium on

Field-Programmable Custom Computing Machines (FCCM), Boulder, 2018.

[1]Ph.D Candidate, Department of Electrical, Computer and Biomedical Engineering, The University of Rhode Island, Kingston, RI 02881. Email: lidongyang@ele.uri.edu

[2]Associate Professor, School of Computer Science and Technology, Huazhong University of Science and Technology. Email: wufei@hust.edu.cn

[3]Master student, School of Computer Science and Technology, Huazhong University of Science and Technology. Email: wenyang@hust.edu.cn

[4]Distinguish Professor, Department of Electrical, Computer and Biomedical Engineering, The University of Rhode Island, Kingston, RI 02881. Email: qyang@ele.uri.edu

[5]Department Dean, School of Computer Science and Technology, Huazhong University of Science and Technology. Email: csx@hust.edu.cn

1.1 Abstract

The rapid development of nonvolatile memory technologies such as flash, PCM, and Memristor has made processing in storage (PIS) a viable approach. We present an FPGA module augmented to an SSD storage controller that provides wire-speed object deserialization, referred to as HODS for hardware object deserialization in SSD. A pipelined circuit structure was designed to tailor to high-speed data conversion specifically. HODS is capable of conducting deserialization while data is being transferred on I/O bus from the storage device to host. The FPGA module has been integrated with our newly designed NVM-e SSD. The working prototype demonstrated significant performance benefits. The FPGA module can process data in line speed at 100MHz on 16 Byte data stream. For integer benchmarks, HODS showed deserialization speedup of 8∼12× as compared to the traditional deserialization on a high-end host CPU. The speedup can reach 17∼21× for floating-point datasets. The measured object deserialization throughput is 1GB/s on average at a clock speed of 100MHz. The overall performance improvements at the application level range from 10% to a factor of 4.3× depending on the proportion of deserialization time over total application running time. Compared to traditional SSD on the same server, HODS showed visible differences regarding application execution time while running Matlab, 3D modeling, and scientific computations.

1.2 Introduction

Object deserialization is a process of creating data structure suitable for applications. It can spend 64% of the total execution time of an application on average if the traditional deserialization process is used [1]. It typically takes three steps: (1) Raw data is read out of storage device and buffered in the host memory; (2) Host CPU transforms raw data into binaries; (3) Application computation executes

using binary results of object deserialization. This CPU-centric approach becomes inefficient for several reasons: First of all, step 2 cannot take full advantages of modern CPUs, because the scanning access of a significant amount of data has poor data locality making the deep cache hierarchy useless. Secondly, it suffers from considerable overhead in the host system because of frequent context switching caused by significant amount of storage I/Os [2]. Finally, host deserialization intensifies the bandwidth demand of both I/O interconnect and CPU-memory bus which may create I/O bottleneck problem.

Realizing the inefficiencies of host-based object deserialization, researchers have tried to offload such operations to data storage. Tseng et al. presented Morpheus that can substantially speed up benchmark applications using processing in storage (PIS) [1]. By making use of the simpler and more energy-efficient processors found inside SSD devices, Morpheus frees up scarce CPU resources that can either do more useful work or be left idle to save energy. At the same time, it consumes less bus bandwidth than the conventional system. While Morpheus demonstrated 1.66× speedup by using embedded processor inside SSD device to carry out object deserialization, it consumes the scarce resource of the SSD controller cores that are meant to carry out FTL, wear leveling, garbage collection, and flash control functions. Besides, the device I/O path is slowed down by the firmware process because of embedded processor's overhead and buffering of intermediate results in ARM D-Cache [3].

This paper presents a hardware approach to providing wire-speed object deserialization, referred to as HODS, hardware object deserialization in SSD storage. We have designed and implemented an FPGA module that is augmented in an SSD along the I/O path to carry out the necessary data conversion. It works in parallel with all storage operations and conducts real-time computation with pipelined

structure. Instead of buffering intermediate results, our FPGA solution converts the data while data is transferred from storage to the host. Therefore, it eliminates the slow down of read I/Os from the SSD storage. HODS brings several benefits compared to the previous solutions: (1) It substantially speeds up host CPU execution time because of PIS. (2) Our architecture eliminates extra memory accesses between embedded processors and its D-Cache memory hierarchy. (3) This new approach is extensible to execute other computations such as object search, image processing and machine learning, all of them can be easily integrated into current storage ASIC design.

To demonstrate the feasibility and effectiveness of HODS, we have built an FPGA prototype based on an NVM-e [4] SSD storage card with PCI-e Gen3×4. The FPGA SSD controller runs at a 100MHz clock with the bus width of 16 bytes. The HW deserialization module is attached along the 16 bytes bus capable of processing 16 bytes of data in parallel per clock cycle. Data conversion is done concurrently with data transfer on the bus when NVM-e command directs the SSD to do so. Such NVM-e directives are passed along from the host NVM-e driver down to the SSD device. To allow applications to use such functions, we have modified the host NVM-e driver to support our prototype implementation. The working prototype SSD is used to carry out performance measurement experiments. Our measurement results show that HODS accelerates object deserialization by 8 to 12× as compared to host CPU execution time for integer data. For floating point data, the speedup ranges from 17 to 21× for deserialization operations. The overall speedup for applications depends on the fraction of deserialization time over the total execution time of benchmarks. For BigDataBench, Rodinia and JAPSPA benchmark applications, we observed an overall speedup of 10% to a factor of 4.3×. Compared to traditional SSD on the same server, HODS showed visible differences

in terms of application execution time while running Matlab, 3D modeling, and scientific computations. The demo video for the Matlab application can be found on YouTube at [5].

This paper makes the following contributions: (1) It presents an FPGA deserialization module that can provide wire-speed data conversion. We have designed and implemented the FPGA module alongside the I/O bus inside a PCI-e SSD card using NVM-e protocol. (2) It realized a PIS function in a modern SSD storage and offered practical benefits to applications. It is also extensible to other PIS functions. (3) A working prototype has been built to be functional running at a clock speed of 100MHz. Even at this low clock speed, it provides data conversion speed of 1GB/s. (4) Extensive performance measurements have been carried out to demonstrate the performance and effectiveness of HODS.

The rest of this paper is organized as follows: Section II describes the motivation of hardware deserialization and its corresponding performance issues. Section III provides detailed design for FPGA object deserialization module including hardware PIS storage architecture, FPGA object deserialization module, and host programming API. Section IV describes the experimental prototype implementation. Section V reports performance results. We conclude our paper in Section VI.

1.3 Motivation of hardware deserialization

Most non-database applications such as scientific data analytics, 3D modeling, or spreadsheet applications use interchangeable data formats such as ASCII code. Such serialized memory objects make it easy to collect, exchange, transmit, or store data [2] because the text-based (e.g., CSV [6], txt) encodings allow machines with different architectures (e.g., little endian vs. big endian) to exchange data with each other. It does not require users to understand memory layout of machines,

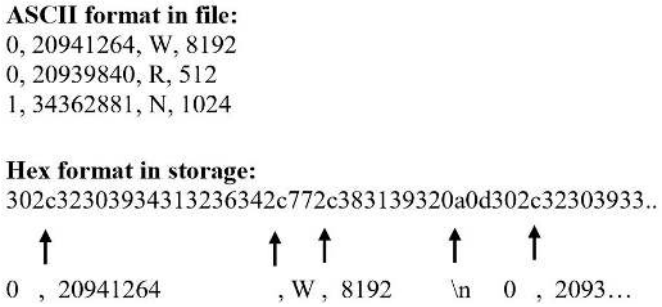

Figure 1. An example ASCII file.

and it is often easy to manage text-based encoding files without using special editing tools.

Figure 1 shows an example of a standard ASCII file chunk. Meaningful ASCII strings are stored between special characters such as space, line-feed, and comma. Before any computation can be done on the data, such text-based encoding strings must be converted into machine binaries readable by applications [7]. To understand how such data conversion affects the overall application performance, we ran a set of benchmarks on a Lenovo server with a quad-core Intel i7-4470 CPU. The benchmark datasets are stored in an Intel 750 series NVM-e SSD. In this experiment, each benchmark application reads the data file from the SSD, converts the file from text to binary in the system RAM, and then processes the data. Figure 2 shows the breakdown of the execution time of the benchmark applications [7, 19, 20]. It can be seen from the figure that the object deserialization (data conversion) takes a significant proportion of the total execution time of applications, ranging from 32% to 85%.

To minimize the overhead of host CPU, object deserialization in PIS has been proposed in flash memory SSDs [1]. Figure 3 illustrates general data flow inside current PIS storage [8∼16]. First, SSD controller loads data from flash to D-Cache using DMA (step 1); Next, the embedded processors (such as ARM core) fetch data from D-Cache and execute PIS functions (step 2). After that, the embedded cores

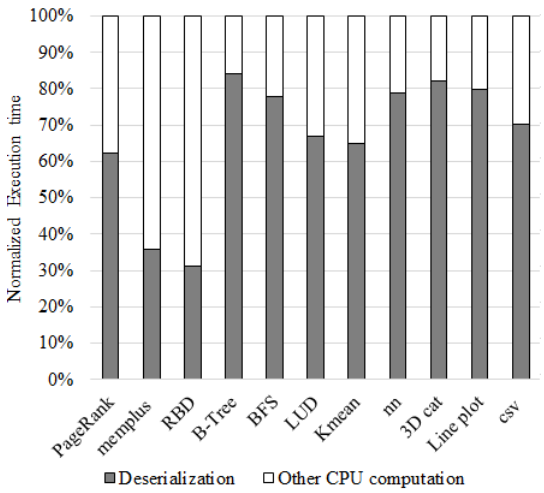

Figure 2. Fractions of object deserialization time over total running time of applications.

write PIS results back into D-Cache (step 3). Finally, host fetches PIS results from the D-Cache to host main memory through NVM-e/PCI-e interconnect (step 4). Although current PIS storage can offload host object deserialization to SSD controller, following limitations exist:

Back and forth accesses of D-Cache stall standard storage IO path. As shown in Figure 3, step 2 and step 3 slow down the I/O operations. Because moving data in and out of D-Cache takes time, and it interrupts standard I/O flow. In conventional PCI-e or NVM-e SSD, storage data can directly move from flash to host main memory by one DMA operation [1, 9, 10]. Because of this PIS architecture, it breaks single DMA data movement into two sub DMA operations.

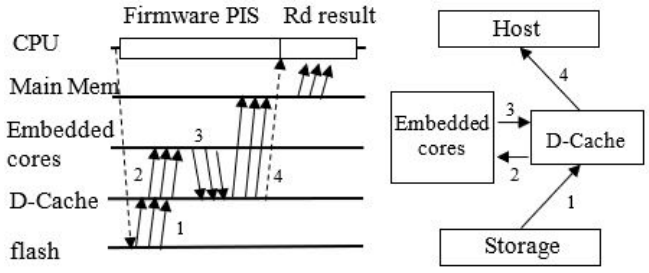

Figure 3. General data flow of existing PIS functions inside an SSD device.

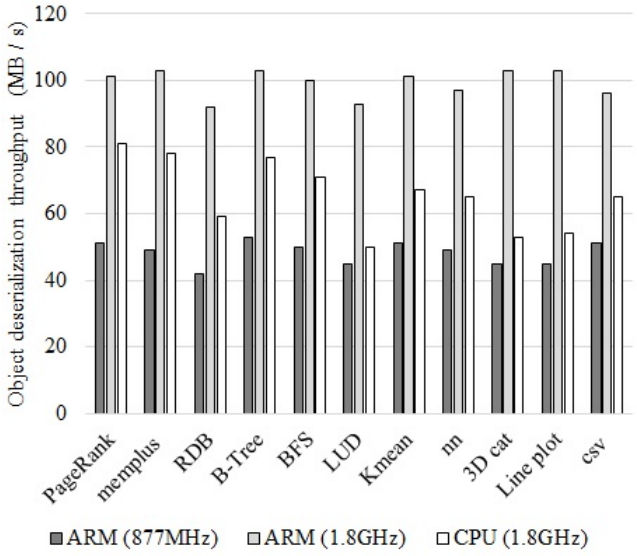

Figure 4. Comparison of object deserialization throughput between embedded ARM inside SSD and host CPU.

One goes in D-Cache, and the other goes out of D-Cache. This modification blocks IO path and slows down storage read speed [17, 18].

PIS using Embedded cores in SSD is not efficient enough. To verify the actual efficiency of using embedded cores for object deserialization, we experimented with ARM Cortex-A9 processor with two different clock settings. As shown in Figure 4, single ARM with 877MHz clock speed can provide 42∼53MB/s throughput on both integer and floating-point benchmarks [19, 20]. There is not much throughput difference between integer and floating-point because of FPU (floating point unit) inside ARM processor in our experiment. Object deserialization throughput increases to 91∼104MB/s when setting ARM clock to 1.8GHz. To make a comparison with the host server, we choose Xeon E5 CPU with 1.8GHz to run the same benchmarks. The host was set up with Linux Ubuntu 16.04. The benchmarks are cached in the host DRAM before the object deserialization execution. Our measurement results show that ARM accelerates object deserialization by 1.25∼1.76× as compared to host CPU execution time when set to the same

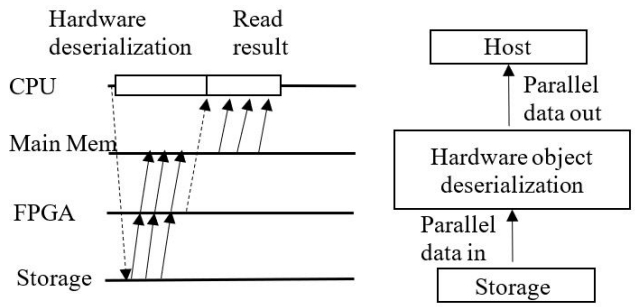

Figure 5. FPGA based object deserialization model.

clock speed. However, current NVM-e/PCI-e bandwidth can reach 4GB/s (e.g., PCI Express Gen3x4). Therefore, there is still a plenty of room for performance improvement for PIS.

In addition to the speed limitations, the embedded cores inside SSD are mainly used for control functions such as FTL, wear leveling, garbage collection, and flash control functions. These controller functions already consume a lot of computing resources of the embedded cores. Adding additional processing tasks for the PIS functions may overload the cores and adversely impact the I/O performance.

D-Cache resource is limited. D-Cache is the precious resource inside a flash SSD controller and its size is limited. The major function of D-Cache inside SSD is to cache FTL table and a small amount of hot data. Because of D-Cache size limitations, flash controller can neither buffer a large amount of data to be converted nor can it hold many intermediate values during PIS processing. Most existing PIS storage systems frequently access D-Cache, increasing the workload of a flash controller and dragging down PIS throughput at the same time [9, 17, 21, 22].

From the above discussions, it is clear that doing object deserializations using software running on embedded cores has limited performance gain. Hardware FPGA accelerators are desirable to speed up such necessary processing step. Our HODS architecture aims at offering such acceleration without slowing down I/O

operations. It does not take away scarce resources from SSD controller cores.

1.4 Hardware Deserialization SSD Architecture

Figure 5 depicts the time slices of FPGA based HODS design. Compared to previous architecture in Figure 3, there is no supefluous memory access to store and fetch intermediate results [23, 24]. We build a direct IO path from storage to host main memory, and PIS is done concurrently with data transfer on the bus. In the following paragraphs, we will describe system architecture, hardware object deserialization module, and host driver program in detail.

1.4.1 System Architecture

Figure 6 shows the overall architecture of the SSD with the hardware PIS for object deserialization. The SSD contains DDR3 for data caching and flash translation layer (FTL). All the storage control functions are implemented on an FPGA. Inside the FPGA chip, major storage logic units include three embedded cores, DRAM/flash controller, NVM-e logic interface, DMA/cache engine and PIS function for hardware object deserialization. All modules are connected to AXI4 bus which is a bridge for data movement among host, flash and DDR3. As the flash controller processor, three embedded cores are responsible for standard storage control workflow. They do not get involved in PIS processing, but only direct storage data flow to go through FPGA object deserialization module.

1.4.2 FPGA object deserialization module

To extract meaningful data structure from ASCII files, we designed and implemented hardware deserialization module. As shown in Figure 7, the hardware object deserialization module is a four-stage pipeline. The first stage pipeline is to search special characters such as space, line-feed, and comma along with n bytes parallel data stream. The special characters' location information will be passed

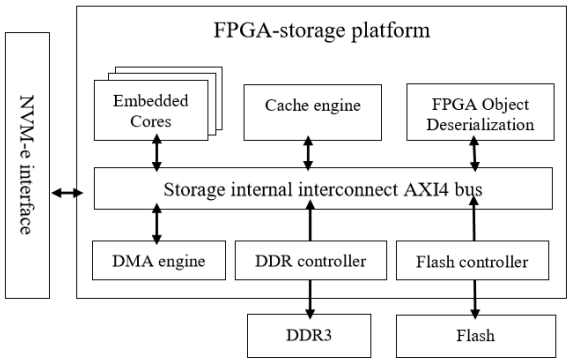

Figure 6. HODS architecture of FPGA based NVM-e storage.

down to the next pipeline stage. The second stage figures out how many object characters are between two special characters. The third stage pipeline converts objects to integer or floating point, and it can bypass ASCII string. At last, object data assembler collects all deserialization results from n parallel modules from stage three. Final object deserialization results are sent to the NVM-e interface directly.

Special character search engine: We search every byte along with n-byte width data stream in each clock cycle, which requires n parallel search units to keep pace with the wire speed. Each search unit corresponds to 1-byte comparator in the circuit. The output of special character search engine is a channel enable switch of the second stage pipeline: object length detector, as shown in upper part of Figure 8.

Object length detector: Input data stream splits into n sliding windows (shingles) as shown at the top part of Figure 8. Each shingle contains $m \times n$ bytes,

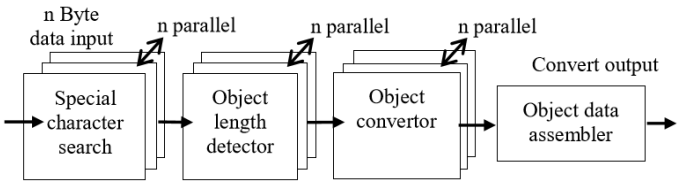

Figure 7. Hardware object deserialization diagram.

where m is an integer that $m \times n$ indicates the maximum object length we can detect between every two special characters. The output from pipeline stage one indicates which shingle's first byte hits special character such as space, line-feed or comma. If a shingle's first byte hits a special character or current shingle is the start shingle of a data file, its length detector is enabled to search the next nearest special character. Otherwise, corresponding shingle length detector is disabled.

Because every first byte of each shingle is used to enable/disable shingle length detector, it requires $m \times n$-1 comparators to work in parallel for the remaining bytes along with the rest shingle content. All comparators' results are assembled into low address arbiter to find out object length from the start byte of the shingle. If shingle's first byte is not a special character, object length detector will disable current shingle output.

The $m \times n$-1 comparators also detect the location of the decimal point. According to the binary values of the shingle content, the object length detector identifies the type of shingle data (integer, floating point or ASCII string) and passes down the shingle type to the object converter along with the object length, shingle content and decimal location.

Object converter: n shingle converters are working in parallel, each one of them processes three types of the shingle data: the floating point shingle is converted to the floating point data by FPU [27]; the ASCII string shingle bypass; the integer shingle goes to the multiplexer matrix. As shown in the middle part of Figure 8, each integer shingle converter is composed of a multiplexer matrix. Each column of multiplexers shares the same weight of multiplicand such as times one thousand or times one hundred.

Object length uses MUX to choose a row of multiplexer matrix. The selected

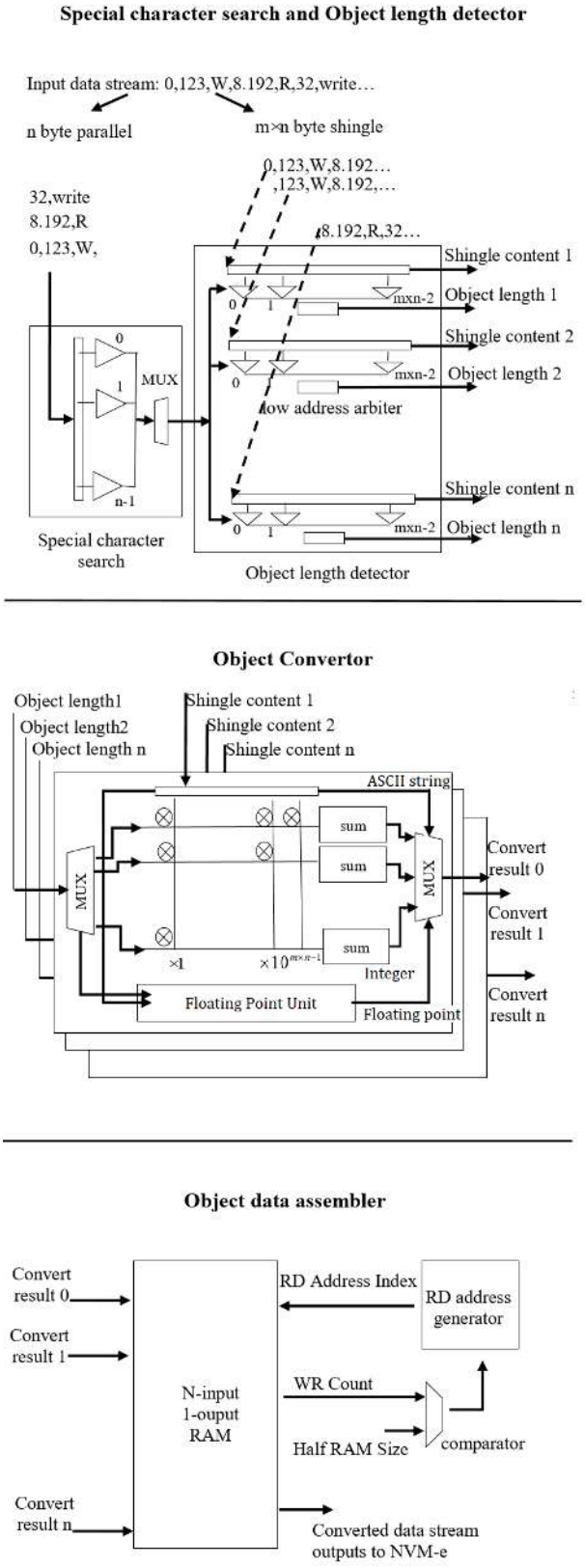

Figure 8. Pipeline stages of hardware object deserialization

row first fetches shingle content to its local buffer and converts ASCII to binary for each byte in the shingle. Secondly, each shingle byte multiplies corresponding multiplicand weight. Finally, selected row sums up all weight values together as converted results. To optimize hardware resource in FPGA implementation, we turned such multiplexer matrix into table lookup structure, which precomputes multiply values and stores into lookup tables. It saves 70% logic resources compared to multiplexer matrix.

Object data assembler: As shown at the bottom part of Figure 8, object data assembler is the collector of n parallel shingles. The output results from the third stage pipeline are fixed size binaries. Object data assembler sequentially buffers such binaries into n-input/*1*-output RAM. Once the write count of the buffer RAM exceeds a threshold value, e.g., half of RAM size count, RD address generator starts to read binary results out of RAM and flushes data to NVM-e interface. Newly converted results can still be buffered by writing into another half of RAM address. It grantees all the converted results can be flushed into NVM-e interface without halt.

1.4.3 Host Driver Program

To allow an application to use the hardware object deserialization module, we have developed a programming framework including libraries and NVM-e driver modifications using C/C++ programming languages. This section will briefly introduce our driver program and show how our driver interacts with hardware object deserialization module.

On the driver side, NVM-e is a scalable host controller interface developed specially for accessing non-volatile memory attached via PCI-e bus. It includes support for parallel operation by supporting up to 64K commands within a single I/O queue to the device. NVM-e encodes commands into 64-byte packets and

uses one-byte command opcode [4]. We modified original host NVM-e module by adding one-bit flag opcode into NVM-e read command. Other commands remain unchanged. The newly added flag bit is a switch to determine whether the storage internal data flow bypasses or goes through the object deserialization module. If the flag bit is not set, SSD controller initiates DMA to move data from flash to host main memory. Otherwise, SSD controller directs flash data to go through hardware deserialization module and sends results to NVM-e interface. Our NVM-e driver does not touch the original submission and completion queue strategy, and modification effort is minimal.

In host application, original C/C++ object deserialization functions such as (*fscanf*) or (*sscanf*) are replaced by our application function (*HODS_scanf*). The HODS converts all variation sized ASCII strings into fix sized binaries and sequentially stores such binaries into host main memory. Our application function (*HODS_scanf*) sequentially access host main memory to fetch results directly, which substantially offloads host CPU's workload.

1.5 Experimental Methodology

We have built an NVM-e SSD prototype that supports hardware object deserialization and carried out performance evaluation using several standard benchmarks. This section discusses the prototype setup and benchmark selection.

1.5.1 Experimental platform

The experimental platform uses Lenovo server with a quad-core Intel i7-4470 running at 3.4 GHz. The system DRAM size is 32 Gbyte. The host was set up with Linux Ubuntu 16.04, kernel version 4.4. Our prototype NVM-e SSD card plugs into host server through PCI-e Gen3x4 interconnect.

We use Xilinx Ultra-scale VU9P as flash controller chip on prototype stor-

age card [5]. All storage logic fits into a single FPGA chip, including embedded processors, DRAM/flash controller logic, NVM-e module, DMA/cache engine and hardware deserialization function. This prototype card contains 8Gbyte DDR3 and 1TB flash memory. To evaluate HODS, we store benchmark dataset on 1TB flash before host starts applications. The following paragraph describes benchmark we used in this paper.

1.5.2 Benchmarks

We selected benchmarks from BigDataBench [20], JASPA [7] and Rodinia [19] with following criteria: (1) The input data of applications are text files. (2) Large and meaningful inputs data can be generated from benchmark tools for our evaluation. (3) The application contains many floating point values that we can evaluate our prototype comprehensively. (4) The application is open source in C programming that is compatible with our prototype. Benchmark applications may apply MPI [25] or openMP [26] to parallelize host computation. Some applications provide data generators such as LU-decompression (LUD), Breadth First Search (BFS), K-mean and B-tree. Other datasets are generated by duplicating benchmark input data. We also provide 3D plot application to demonstrate user experiences of using HODS as compared to existing systems [5]. All benchmark program codes are written in C/C++, and we use Verilog to generate RTL for FPGA.

1.6 Evaluation results

For the purpose of comparative analysis, we consider the baseline as running applications on the server machine with HODS disabled. Using the same server machine, we enable HODS and run the same set of applications to evaluate performance.

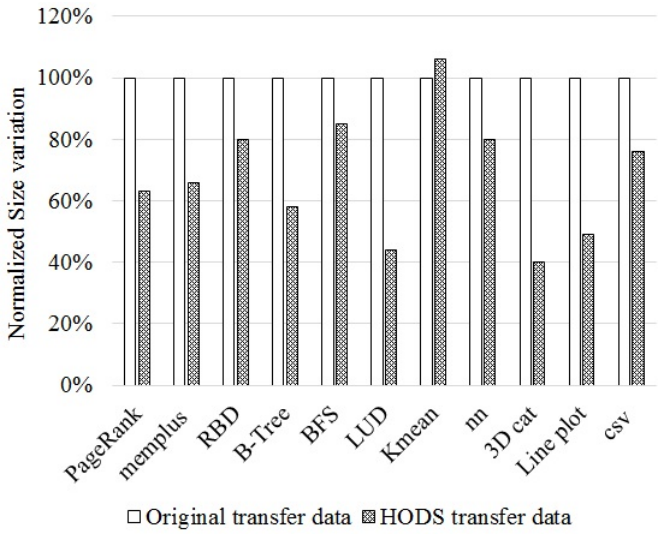

Figure 9. Normalized size variation after hardware deserialization.

1.6.1 Transfer size variation

Figure 9 shows data size changes after FPGA object deserialization. Transfer size shrinks because text-based encoding usually requires more bytes than binary representations. For example, ASCII string "87654321" requires 8 bytes to represent a single object value, but it is only 4 bytes in binary. The longer object is, the smaller converted data size will be. We also eliminate special characters such as space, line-feed and comma, which are unneeded data for benchmark applications.

The size variations of PageRank, memplus, BFS, B-tree and nearest neighbor decreased 15%∼41% after going through the hardware deserialization module. LU-decompression, line plot and 3D cat benchmarks are floating point only. The average object length is 8∼9 bytes each. As a result, the transmitted data size reduces by 51%∼60% after hardware object deserialization. The size variation of Kmean increased 6% after hardware deserialization. Because half of ASCII strings in Kmean are single byte length, data size expands when using 32-bit binary to represent single byte ASCII string.

The size reduction of HODS is a positive side effect of PIS. It reduces the

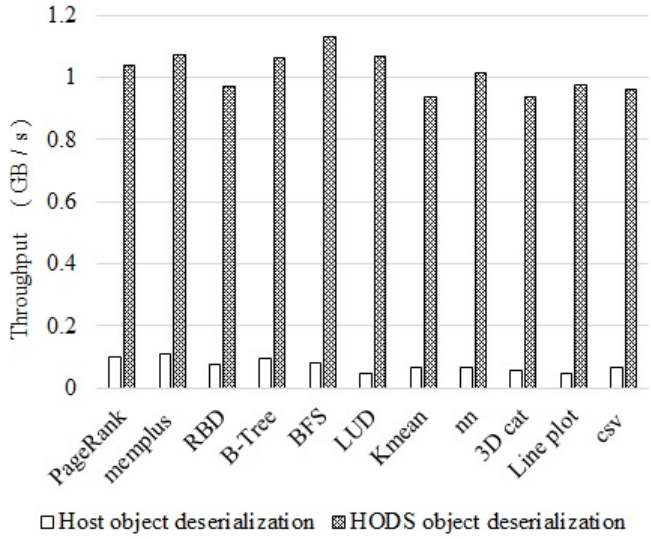

Figure 10. Throughput comparison between hardware object deserialization and host software solution.

I/O bus burden while running applications that require a large data set. It also reduces the IOPS (I/Os Per Second) requirement of the SSD by the applications. Taking PageRank application as an example, a 600K IOPS SSD with HODS would perform the same as a 1 million IOPS SSD without HODS.

1.6.2 Throughput speedup

Figure 10 plots the data conversion throughput. HODS accelerator achieves as much as 935MB/s∼1.13GB/s object deserialization throughput in 100MHz FPGA clock, and host CPU has 58MB/s∼93MB/s throughput at 3.5GHz clock speed. For integer benchmarks such as PageRank, memplus, B-tree and BFS, we observed 8∼12× speedup. These Performance gains can be attributed to two facts. First, it provides at least 100Mhz-16Byte wire-speed processing in HODS. Secondly, resultant data size decreased 15%∼41% after FPGA object deserialization. It can potentially reduce the storage traffic overhead.

Because host CPU takes much longer time to convert floating point numbers from ASCII code, HODS' speedup is even higher for floating point benchmarks such

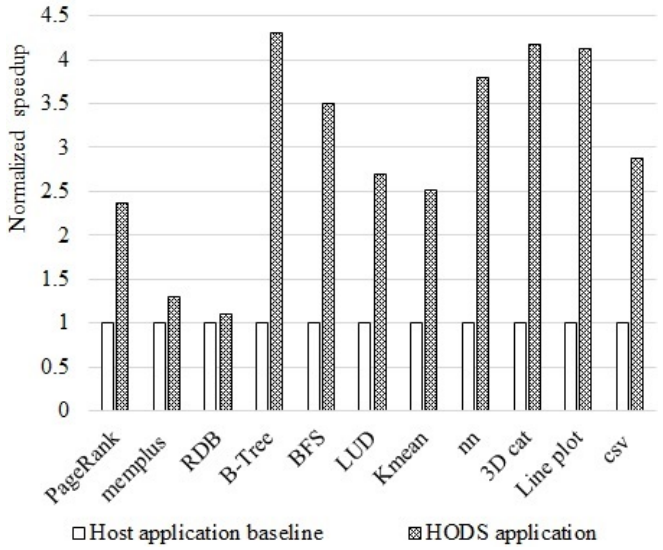

Figure 11. Normalized hardware deserialization speedup.

as LU-decompression, 3D-cat and line plot. Furthermore, data sizes of floating point benchmarks are also reduced by 51%∼60%, giving rise to more speedup. From our experiments, we observed speedup between 17× and 21×. In both integer and floating point object deserializations, HODS runs faster than the existing state of art [1] that has shown 1.66× for the same benchmarks.

1.6.3 Speedup of Application Execution Time

The overall speedup of application programs depends on the fraction of data conversion time over benchmark applications' running time. Our work focuses on object deserialization itself. If benchmark application is computation intensive, the data conversion becomes a small fraction of total time. Then its performance improvement is limited.

Figure 11 plots the HODS' speedup of applications. Benchmarks such as RDB and memplus give only 10% to 30% speedup because they contain matrix multiplication which is computation intensive. The other benchmark applications showed 2.4∼4.3× speedup. Current benchmark applications apply MPI or OpenMP par-

allel model with quad-cores. We expect higher speedup when using more cores or GPUs that run the computation part in parallel but can hardly do anything on data conversion part. Quantitative investigation on such parallel computer architectures is out of our research scope of this paper.

1.7 Conclusion

This paper presents a hardware object deserialization in SSD (HODS) that offloads data-intensive computation to storage where data is stored. Compared to existing state of art [1], HODS eliminates SSD controller's overhead and buffer limitations. It can process storage data in wire speed and does not interfere with SSD controller's firmware resources. Our host driver program provides a user-friendly application interface to replace *fscanf* or *sscanf* function in C/C++, Matlab, python or any other programming languages.

To demonstrate the feasibility and effectiveness of HODS, we have implemented a HODS module inside a prototype NVM-e SSD. The SSD controller is implemented on an FPGA chip running at 100MHz clock with the bus width of 16 bytes. Hardware object deserialization is done concurrently with data transfer on the bus. Our measurement results show that HODS speeds up object deserialization by 8× to 12× as compared to host CPU execution time for integer data. For floating point data, the speedup ranges from 17× to 21× for deserialization alone. The overall speedup for applications depends on the fraction of object deserialization in total benchmark execution time. For BigDataBench, Rodinia and JAPSPA benchmark applications, we observed the speedup of 10% to a factor of 4.3×. Compared to traditional SSD, HODS shows noticeable performance gains while running Matlab, 3D modeling, and scientific data analytics.

List of References

[1] H. Tseng, Q. Zhao, Y. Zhou, M. Gahagan, S. Swanson, *Morpheus: Creating Application Objects Efficiently for Heterogeneous Computing*, in Proceedings of the International Symposium Computer Architecture, 44(3): 53-65, ACM/IEEE,2016.

[2] K. Maeda. *Performance evaluation of object serialization libraries in XML, JSON and binary formats*. in Digital Information and Communication Technology and it's Applications, 2012.

[3] R. Mueller, K. Eguro, *FPGA-Accelerated Deserialization of Object Structures*, Technical report MSR-TR-2009-126, Microsoft Research Redmond (2009)

[4] A. Huffman. *NVM Express Revision 1.1* http://www.nvmexpress.org/resources/specifications/

[5] *HODS demo:* https://youtu.be/8TIDz7eDbHs

[6] *CSV file format:* https://en.wikipedia.org/wiki/Commaseparated-values

[7] Y.Hu, R.Allan and K.Maguire, *Comparing the performance of JAVA with Fortran and C for numerical computing*. in IEEE Antennas and Propagation Magazine, vol. 40, no. 5, pp. 102-105, Oct. 1998.

[8] Y. Xie, D. Feng, Y. Li, and D. D. Long. *Oasis: An active storage framework for object storage platform*. in Future Generation Computer Systems, 2015.

[9] H. W. Tseng, Y. Liu, M. Gahagan, J. Li, Y. Jin, and S. Swanson. *Gullfoss: Accelerating and simplifying data movement among heterogeneous computingand storage resources*. Tech. Rep. CS2015-1015, Department of Computer Science and Engineering, University of California, San Diego technical report, 2015.

[10] J. Power, A. Basu, J. Gu, S. Puthoor, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood. *Heterogeneous System Coherence for Integrated CPU-GPU Systems*. in Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-46, pp. 457-467, 2013.

[11] S. Cho, C. Park, H. Oh, S. Kim, Y. Yi, and G. Ganger. *Disk Meets Flash: A Case for Intelligent SSDs*. in Proceedings of the CMU Technical Report, 2011.

[12] S. Boboila, Y. Kim, S. Vazhkudai, P. Desnoyers, and G. Shipman. *Active flash: Out-of-core data analytic on flash storage*. in Proceedings of the Mass Storage Systems and Technologies, pp. 1-12, 2012.

[13] Y. Kang, Y.-S. Kee, E. L. Miller, and C. Park. *Enabling cost-effective data processing with smart ssd*. in Proceedings of the Mass Storage Systems and Technologies, 2013.

[14] B. Gu, A. Yoon, D. Bae, I. Jo, J. Lee, J. Yoon, J. Kang, M. Kwon, C. Yoon, S. Cho, J. Jeong, and D. Chang. *Biscuit: a framework for near-data processing of big data workloads*. in Proceedings of the International Symposium Computer Architecture, 2016.

[15] D. Tiwari, S. S. Vazhkudai, Y. Kim, X. Ma, S. Boboila, and P. J. Desnoyers. *Reducing data movement costs using energy efficient, active computation on ssd*. in Proceedings of the 2012 USENIX Conference on Power-Aware Computing and Systems, pp. 4-14, 2012.

[16] J. Do, Y.-S. Kee, J. M. Patel, C. Park, K. Park, and D. J. DeWitt. *Query processing on smart ssds: Opportunities and challenges*. in Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data, pp. 1221-1230, ACM, 2013.

[17] S. Seshadri, M. Gahagan, S. Bhaskaran, T. Bunker, A. De, Y. Jin, Y. Liu, and S. Swanson. *Willow: A user-programmable ssd*. in Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation, pp. 67-80, USENIX Association, 2014.

[18] S.-W. Jun, M. Liu, S. Lee, J. Hicks, J. Ankcorn, M. King, S. Xu, and Arvind. *Bluedbm: An appliance for big data analytics*. in Proceedings of the 42Nd Annual International Symposium on Computer Architecture, ISCA 15, pp. 1-13, ACM, 2015.

[19] M. B. S. Che, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, *Rodinia: A Benchmark Suite for Heterogeneous Computing*, in Proceedings of the IEEE International Symposium on Workload Characterization, pp. 44-54, 2009.

[20] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu. *Bigdatabench: A big data benchmark suite from internet services*. in Proceedings of the High Performance Computer Architecture, pp. 488-499, Feb 2014.

[21] I. S. Choi and Y.-S. Kee. *Energy efficient scale-in clusters with in-storage processing for big-data analytics*. in Proceedings of the 2015 International Symposium on Memory Systems, pp. 265-273, ACM, 2015.

[22] A. Acharya, M. Uysal, and J. Saltz. *Active disks: Programming model, algorithms and evaluation*. in Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 81-91, ACM, 1998.

[23] R. Mueller, J. Teubner, and G. Alonso. *Streams on wires: A query compiler for fpgas*. in Proceedings of the Proc. VLDB Endow., vol. 2, no. 1, pp. 229-240, 2009.

[24] R. Mueller, J. Teubner, and G. Alonso. *Data processing on FPGAs*. Proc. VLDB Endow., pp. 910-921, 2009.

[25] *MPI org:* http://mpi-forum.org/mpi-40/

[26] *OpenMP org:* http://www.openmp.org/specifications/

[27] *Floating Point Unit IP:* https://opencores.org/project,fpu

MANUSCRIPT 2

CISC: Coordinating Intelligent SSD and CPU to Speedup Graph Processing

by

Dongyang Li[1], Weijun Li[2], Qing Yang[3]

is published in the 17th International Symposium on Parallel and Distributed Computing (ISPDC), Geneva, Switzerland, 2018

[1]Ph.D Candidate, Department of Electrical, Computer and Biomedical Engineering, The University of Rhode Island, Kingston, RI 02881. Email: lidongyang@ele.uri.edu

[2]CTO, Shenzhen DAPU microelectrical inc., Shenzhen, China, Email: wenjunli@dputech.com

[3]Distinguish Professor, Department of Electrical, Computer and Biomedical Engineering, The University of Rhode Island, Kingston, RI 02881. Email: qyang@ele.uri.edu

2.1 Abstract

Minimum Spanning Tree (MST) is a fundamental problem in graph processing. The current state of the art concentrates on parallelizing its computation on multi-cores to speedup MST. Although many parallelism strategies have been explored, the actual speedup is limited, and they consume a large amount of CPU power. In this paper, we propose a new approach to the MST computation by coordinating computing power inside SSD storage with host CPU cores. A comprehensive framework of software-hardware co-design, referred to as CISC (coordinating Intelligent SSD and CPU), preprocesses MST graph edges inside storage and parallelizes the remaining computation on host CPU. Leveraging the special properties of modern SSD storage, CISC exploits a divide and conquer approach to reordering graph edges. We have implemented an FPGA circuit that reorders chunks of graph edges inside an SSD. The ordered chunks are then loaded to the system RAM and processed by the host CPU to build a B-Tree structure by repetitively picking up edges at heads of chunks. A working prototype CISC has been built using NVM-e SSD on a server. Extensive experiments have been carried out using real-world benchmarks to demonstrate the feasibility and performance of deploying CISC in NVM-e SSD storage. Our experimental results show 2.2∼2.7× speedup for serial version implementation and 11.47× to 17.2× speedup for the parallel version with 96-cores.

2.2 Introduction

Processing of graph-structured data has become increasingly important and has brought to the forefront of computational challenges. Graphs with up to billions of vertices and trillions of edges are commonplace in today's big data era [1]. Minimum Spanning Tree (MST) is a fundamental problem in graph processing to compute a subset of a graph with the total edge weight being minimum. It is

pervasive throughout science, broadly appearing in fields such as social network, biological science, transportation, VLSI and so forth. A classical way of computing MST is the well-known Kruskal algorithm which sorts edges in ascending order first and then merges them into a subset without overlaps. Extensive research has been reported in the literature to speed up Kruskal MST computation using parallel architectures [2∼7]. On parallel architectures such as FPGA, multicores and GPUs, MST computation can be divided into multiple tasks that are executed concurrently in parallel, and hence speeding up MST computation.

However, effectively parallelizing MST faces many challenges. The most critical challenges are data dependency and potential deadlock problems. Li et al [2] used High-Level Synthesis (HLS) in FPGA to solve graph edge dependency in MST. The FPGA works as co-processor of CPU and aborts the conflict task when data dependency happens. Experimental results show that such CPU-FPGA co-processing achieves up to 2.2× speedup compared with the single core computation. Subramanian et al [4] presented FRACTAL to speed up MST using multi-cores. Their work is based on a cycle-accurate, event-driven simulator to model parallel system with 256 cores [7]. To avoid data dependency in MST, they modified task scheduler and used timestamps to determine which tasks execute in high priority. Their simulation shows 40× speedup when configured with 256 multicores. Manoochehri et al [6] proposed MST implementation on GPU. They used Software Transactional Memory based synchronization to alleviate data dependencies among GPUs. It outperforms MST running on single core CPU by 4.5×.

While existing research made efforts on solving task deadlock and data dependency problems [2∼7], the ultimate speedups obtained by the current state of art are still limited. In order to further improve MST performance, we carried out extensive experiments to study what is holding the MST from running faster. In

the ordinary storage system, a large number of graph edges are stored in SSD or HDD without ordering. In most existing works, parallel computing architecture loads unsorted graph edges into local memory and sorts them before MST merge. Sorting data in the main memory of host is computation intensive, and it consumes enormous CPU resources. We observed in our experiments that edge sorting takes a significant portion of total MST execution time ranging from 36% to 75% of the total time. We therefore believe that there is a great potential for further performance improvement by leveraging the intelligence available inside SSD where a huge amount of graph edges is stored.

In this paper, we present a new approach to the MST computation by means of CISC (Coordinating Intelligent SSD and CPU). The idea is to exploit the controller logic inside the SSD to preprocess graph edges while being loaded to the main memory of the host. CISC divides the large amount of graph edges into chunks and sorts each chunk of edges in order using hardware. In this way, the edges loaded into the internal memory of the host consist of multiple sorted chunks. To allow software MST to efficiently use sorted chunks, we developed two software programs for the host servers of serial and parallel MST, respectively. The serial MST forms a B-tree holding the smallest edges of all the chunks and merges smaller edges into MST in high priority. In the multicore system, we optimized the classical sample sort algorithm [8] of parallel MST, and the remaining computation can be effectively parallelized on multicores. Such an efficient data distribution in the host main memory ensures smaller weight edges can be processed very efficiently. To demonstrate the feasibility and performance potential of CISC, we have implemented CISC using FPGA inside an SSD. A working prototype has been built that consists of both software running on the host and hardware circuit inside SSD. Using the CISC prototype, we run standard graph benchmarks to measure

performances. Experimental results show that CISC outperforms pure Software MST substantially.

This paper made the following contributions:

- A pipeline structure of FPGA sort module has been presented that can provide wire-speed hardware sort of multiple edge chunks. We have designed and implemented the FPGA module alongside the I/O bus inside PCI-e SSD realizing a true processing in storage (PIS) for graph processing. It is also extensible to other sort-based software applications.

- A B-tree based selection algorithm and an optimized sample sort algorithm have been proposed that run on single core and multicore systems, respectively. CISC coordinates the chunk sorting inside SSD and selection/merging of minimum weight edges on CPU cores efficiently. The software and hardware co-design framework is the first of the kind for graph processing.

- A working CISC prototype has been built that works as expected. The prototype has been used to carry out extensive experiments for performance measurements. Our experimental results demonstrated the superb performance and effectiveness of CISC for MST over existing approaches.

The rest of this paper is organized as follows: In section II, we discuss the related work. Section III provides detailed design for in-storage sort module. Section IV describes the two MST software modules of CISC that run on single-core and multicores, respectively. Section V presents experimental results and discussions. Section VI concludes the paper.

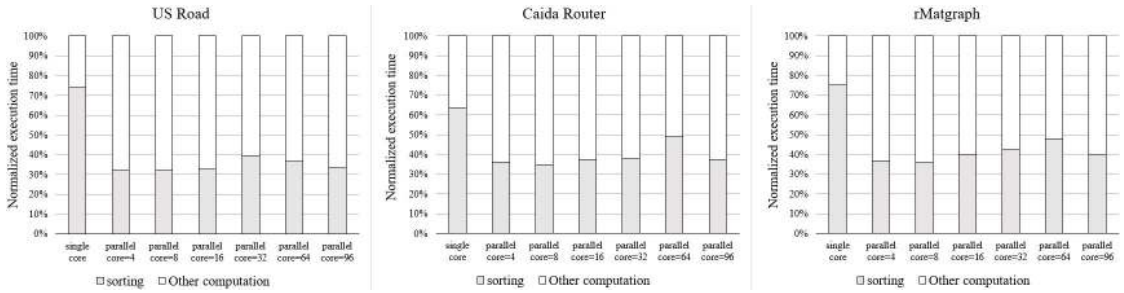

Figure 12. Fractions of graph sort time over total execution time of the serial and parallel MST running on multi-cores.

2.3 Background

2.3.1 Overhead of Sorting in MST

MST computation consists of a number of tasks. The first time-consuming task is edge sorting. To understand how significant the sorting part contributes to the total computation time of MST, we measured the actual sorting time of Kruskal MST and estimated the proportion of sorting time in the entire MST computation. We set up the experiment environment with Intel Xeon processor having 96 cores running at 2.5 GHz. Linux system with kernel version 4.14 was installed on the server. We selected three benchmark datasets from [28∼30]. The parallel software code [9] uses OpenMP when configuring multicores.

Figure 12 shows the breakdown of edge sort time and other computation time of the MST running on 1 to 96 cores. It can be seen from this figure that edge sorting takes a significant proportion of overall MST execution time. For all three benchmarks, we observed consistent behavior. The fraction of time taken for edge sorting ranges from 36% to 75%. In addition to execution time, edge sorting consumes computation resources that could otherwise be used for other computation tasks. Examining the experimental results, we believe such expensive edge preprocessing can be offloaded to data storage device where the large amount of edges is stored.

2.3.2 Previous Work on Near-Data Processing

In many computer systems for the data mining, big-data, and database, the data movement becomes the bottleneck that it causes performance degradation and power waste [34]. Data processing is swiftly moving from computing-centric to data-centric. Inspired by these trends, the concept of NDP [10] (Near-Data Processing) has recently attracted considerable interest: Placing the processing power near the data, rather than shipping the data to the processor. The NDP computation might execute in memory or in the storage device where the input data reside [11], and it can be divided into two main categories: PIM and PIS.

PIM aims at performing computation inside main memory. Various PIM approaches have been proposed since the pioneering work by Gokhale et al. [12]. Recently, Yitbarek et al. [13] have reported accelerator logic for string matching, memory copy, and hash table lookups in hybrid memory cube (HMC) [14][15]. Ahn et al. [16] proposed a scalable PIM architecture for graph processing with five workloads including average teenage follower, conductance, PageRank, single-source shortest path, and vertex cover. They verified the graph processing performance by simulation.

PIS aims at processing in storage (PIS). Early PIS approaches include the Active Disks architecture proposed by Acharya et al. [17]. They perform the scan, select, and image conversion in storage system and provides a potential reduction of the data movement between disk and CPU. Patterson et al. [18] proposed an architecture (IDISK) which integrates the embedded processors into the disk and push computation closer to the data. Their results suggest that a PIS based architecture can be significantly faster than a high-end symmetric multiprocessing (SMP) based server. Choi et al. [19] implemented algorithms for linear regression, k-means, and string matching in the flash memory controller (FMC). BlueDBM

[20] is a PIS system architecture for distributed computing systems with a flash memory-based FPGA. The authors implemented nearest-neighbor search, graph traversal, and string search algorithms by High-Level Synthesis (HLS) in FPGA. Morpheus [33] frees up scare CPU resources by using embedded processor inside SSD to carry out object deserialization. Recently, Biscuit [21] equipped with FMCs and processes pattern matching logic in storage which speeds up MySQL requests. Lee et al [35] proposed ExtraV, a framework for near storage graph processing such as Average Teenage followers, PageRank, Breadth-First Search and Connected Components. It efficiently utilizes a hardware accelerator at the storage side to achieve performance and flexibility at the same time.

Our focus in this paper is on speeding up graph processing that has become increasingly important in today's big data era. As will be evidenced shortly, the benefit is great to preprocess a huge amount of graph data inside SSD where the data is stored.

2.4 Hardware Architecture of In-storage Sort

2.4.1 System architecture

The large fraction of time that edge sorting takes in MST and the intelligence available inside modern SSD motivate us to propose a new and practical PIS architecture. Compared with the existing PIS approaches, CISC is unique in that it uses Verilog to generate RTL and provides wire-speed sort in hardware. A pipelined circuit structure was designed to tailor to high-speed storage data sort especially. Graph edge sort is done concurrently with data transfer on the bus. It minimizes sort overhead of the host server CPU which is computation intensive and time-consuming.

As shown in Figure 13, PIS augments a special functional logic to perform the desired function inside a storage device, in this case, SSD. All the storage control

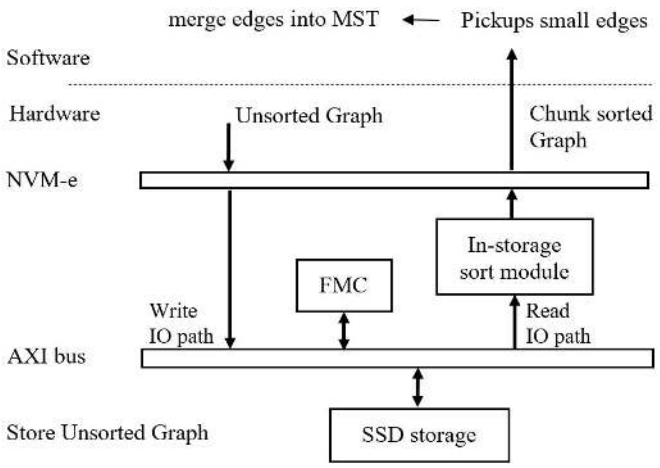

Figure 13. System architecture of the CISC.

functions are implemented on an FPGA. Inside FPGA chip, major storage logic units include the flash controller, NVM-e interface, DMA engine and in-storage sort module. All modules are connected to AXI4 bus which is a bridge for data movement between host and flash memory. The data width of AXI4 bus is 8 bytes with clock speed of 250MHz. As shown in Figure 13, in-storage sort module is added between AXI bus and NVM-e interface along with storage read I/O path. It provides sort function that is activated by NVM-e command and is done while data is being read from the storage to the host.

2.4.2 In-storage sort module

A challenging problem of hardware sort is to sort the large-scale dataset. Due to the on-chip memory size limitations of FPGA, the existing work [22∼25] partially buffers sorted results in the off-chip memory such as DRAM or SSD and reads them back when FPGA performs merge sort. Such off-chip buffer strategy causes multiple FPGA memory accesses and slows down the hardware sort performance of the large-scale dataset.

In order to eliminate the off-chip memory accesses in FPGA sort, CISC takes a divide and conquer approach. Instead of sorting the entire edge list that is huge, we

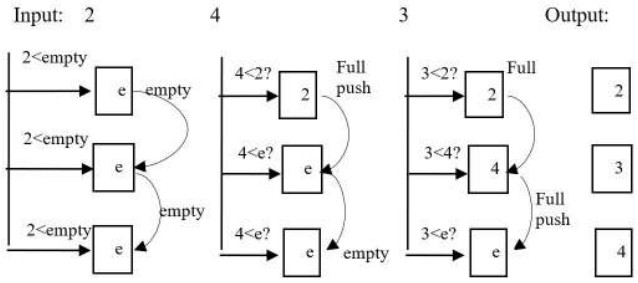

Figure 14. The architecture of the linear-time sorter.

divide the large edge list into chunks and sort these chunks using hardware. Each chunk can fit into FPGA on-chip memory. The pipeline architecture of in-storage sort module provides wire-speed sort of data streams. There are two benefits of dividing edges into chunks to sort. The first one is memory resource savings. It is impossible to hold and reorder large-scale data in FPGA alone. We choose the right chunk size to fit the internal memory space of the FPGA. The second benefit is to bound the in-storage sort latency to match the normal read I/O speed so that the host can read the sorted chunks as if they were directly read from flash memories with no interruption. Once the sorted chunks of edges are loaded to the system RAM, the software on the host can efficiently execute the remaining computation of MST.

The in-storage sort pipeline is composed of the linear-time sorters [25] and several stages of FIFO mergers [22] [24]. We design this architecture especially for the in-storage graph processing with the minimal PIS latency and hardware cost.

As the first stage of the pipeline, the linear-time sorter uses n buffers to hold sorted graph edges. It compares each incoming edge's weight in parallel with all already sorted edges in the buffers and inserts the new graph edge into the appropriate location in the buffers to maintain the existing sorted order [25]. Figure 14 shows an example of n equal to 3 to demonstrate how the linear-time sorter works. Such linear-time sorter generates the sorted sequence of n edges after

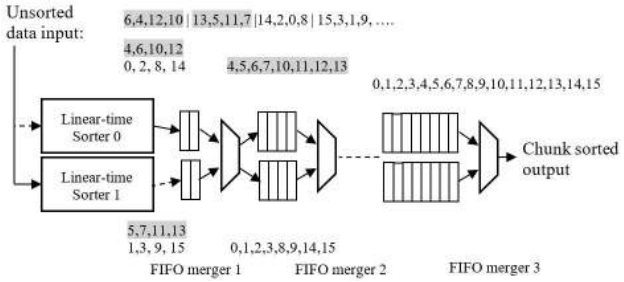

Figure 15. The pipeline architecture of the in-storage sort module.

n clock cycles.

Upon a read I/O from the host, the storage data need to be continuously fed into the PIS function. In order to sort data stream in wire-speed, two linear-time sorters are deployed to work in parallel. As shown in Figure 15, the two linear-time sorters alternate working on the input data and switch functions after every n clock cycles with one sorting the incoming data stream and the other sending out the sorted results to the next pipeline stage.

The linear-time sorter requires buffers and parallel comparators for the parallel comparisons. Such buffers and comparators will become prohibitive costly if the sorted data size becomes very large. Our solution is again divide and conquer by dividing each chunk of data to be sorted into smaller segments. The dual linear-time sorters only sort the initial segment with a small data size. The in-storage sort module then doubles up such segment by FIFO mergers [24] that form the rest of the pipeline stages as shown in Figure 15. To connect the first pipeline stage (dual linear-time sorters) with the rest of pipeline stages (FIFO mergers), the n sorted edges from the linear-time sorter0 are immediately forwarded to one of the FIFO buffers of the next pipeline stage, FIFO merger1, as shown in Figure 15. During the next n cycles, the linear-time sorter1 fills up the other FIFO buffer of the next pipeline stage. The same process repeats when the storage data continuously flushes into the PIS module.

Each FIFO merger stage doubles up the segment of the previous pipeline stage [24]. For example, the size of data sort doubles up from 4 to 16 when the data stream passes through two stages of the FIFO mergers. As shown in Figure 15, each stage of the FIFO merger has two FIFOs. At any given time, the data stream from the previous pipeline stage flushes to one of the FIFOs. If the flush size reaches the size of the previous segment, a control logic switches the data stream to the other FIFO. After one of the FIFO has finished fetching data with the size of the previous segment, the data merge starts and the fetching data flushes into the other FIFO at the same time. Data in the two FIFOs are merged in ascending order to the next pipeline stage of the FIFO merger, that is, we always pick up the smaller data from two FIFOs to be flushed to the next stage [24]. In this way, the current segment merges two of the previous segments and doubles up the sort size. The sort size of the last segment is the chunk size that depends on the FPGA's internal resources (numbers of FIFO merger stages). After passing through the in-storage sort module, the graph edges are loaded into the host main memory in form of multiple sorted chunks.

The startup time of such pipeline of FIFO mergers depends on the data transfer delay of the last stage of the FIFO merger [24]. The delay is the data transfer time of the first chunk of the graph data. Therefore, PIS latency is only the pipeline's startup time when the host server reads the first chunk of a large number of sorted chunks from the storage.

2.5 Software Design of CISC

To allow the MST application to use the in-storage sort module, we developed two CISC software modules running on the host, one for single core CPU and the other for parallel MST running on multicores. The following paragraph describes the software design of CISC.

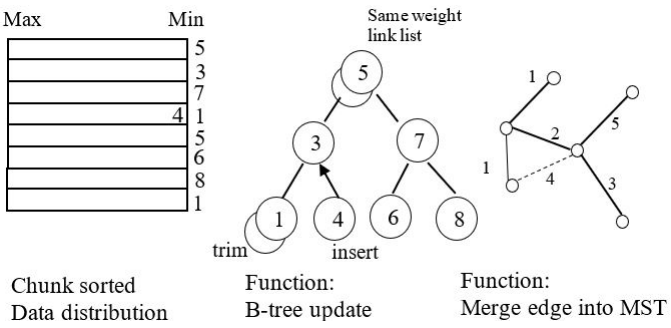

Figure 16. MST software of CISC on the single-core system.

2.5.1 Serial CISC software

As shown in Figure 16, a B-tree based selection algorithm has been developed in the serial CISC software and coordinates with the chunk sorting circuit. To initialize the B-tree, the serial CISC software picks up graph edges from all the chunk heads. It has n_{edge}/chk_size B-tree nodes holding the smallest edges of all the chunks, where n_{edge} denotes edge numbers and chk_size is the chunk size, i.e. the number of sorted edges per chunk. In order to avoid collision, each B-tree node adds the same weight edges into a linked list. After the initialization of B-tree, the serial CISC software trims the minimum edge from the B-tree by the rule of in-order traversal [26]. A new edge from the chunk head of the trimmed edge is the next B-tree candidate, and the software inserts it into the B-tree after the previous minimum edge is trimmed. The size of the B-tree remains the same (n_{edge}/chk_size) during software execution.

As shown in Algorithm 1, the serial CISC software merges the trimmed edges from the B-tree into a graph subset. Once the growing subset forms a cycle, the software abandons the currently selected edge and picks the next edge from the B-tree to grow MST. Such a process stops when MST traverses all n_{vertex} vertices of the graph, which takes n_{merge} iterations in Algorithm 1. The B-tree selection avoids sorting all the graph edges because the smaller edges are placed at the heads of chunks and merge process always picks up the smallest from the B-tree

into MST.

For the same graph, both the software MST and the CISC MST will select the same set of graph edges to form the MST and take same number of iterations, n_{merge}, to merge the small edges into the MST. The superiority of serial CISC MST comes from the graph sort. Instead of sorting n_{edge} edges, the serial CISC MST picks up a minimum graph edge from the B-tree and merges the graph edge in every iteration. The number of iterations (n_{merge}) is related to n_{vertex} and much smaller than n_{edge} in most graphs in practice.

Table 1 shows the time complexities of the sorting part of the MST algorithm of traditional software MST and our CISC. While the best time of software sort is $n_{edge} \times log_2(n_{edge})$, CISC sort time is $n_{merge} \times log_2(n_{edge}/chk\_size)$. During the CISC software execution, the B-tree size remains the same (n_{edge}/chk_size). For B-tree updates, the time complexity is $log_2(n_{edge}/chk\_size)$. Such B-tree update is performed concurrently with the merge operations of MST. CISC sort finishes when all of the graph vertices (n_{vertex}) are merged, taking n_{merge} iterations. From the comparison of these two formulas, we can see CISC takes advantages of both smaller value of n_{merge} and the efficient data distribution of sorted chunks.

Table 1. Comparison of the serial sort between software and CISC

	Execution time of serial sort
Software	O($n_{edge} \times log_2(n_{edge})$)
CISC	O($n_{merge} \times log_2(n_{edge}/chk\_size)$)

2.5.2 Parallel CISC software

The parallel CISC software cannot use the B-tree selection algorithm because of data dependency. Each edge selection depends on the previous updates of the B-tree, and it may cause task deadlocks wasting the multicores' computational resources.

Algorithm 1: The serial CISC software of MST

```
0:   n_merge = 0;
1:   Initial B-tree size = n_edge/chk_size;
2:   for k < n_vertex
3:        select edge = trim (B-tree);
4:        update (B-tree);
5:        if (merge_MST (select edge) == success) k++;
6:        else k = k;
7:        n_merge++;
8:   end for
```

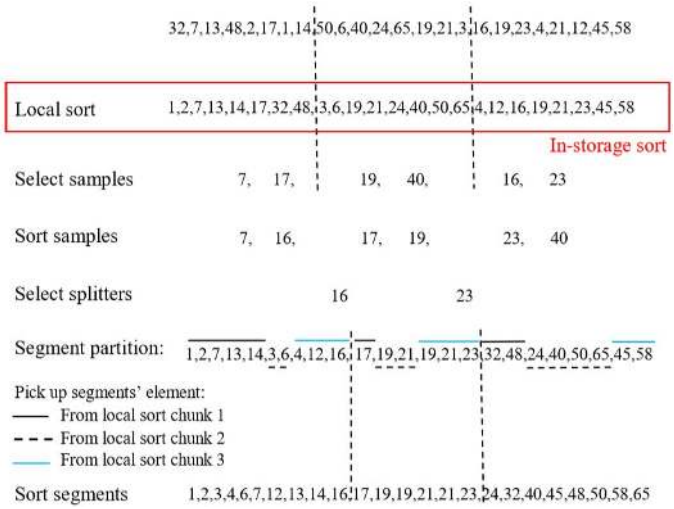

Figure 17. CISC optimizes sample sort algorithm in MST.

In order to speed up MST in the multicore system, we optimized the classical sample sort algorithm [8] of parallel MST. The concept of the sample sort is to divide the dataset into segments, and the data values within each segment have a range. The ranges among segments are non-overlapping. CPU cores sort these segments in parallel and complete the sample sort after combining all of the sorted segments. However, in most cases, the unsorted data does not follow the above segments' data distribution. The sample sort algorithm needs reshuffle the dataset by selecting samples and partition segments. Figure 17 shows a sample sort example of the n_{total}=24 sorting elements with p=3 parallel tasks. There are four major steps of the sample sort algorithm:

(1) Local sort: Multiple tasks divide the n_{total} elements into p chunks of the size n_{total}/p each and sort these chunks in parallel.

(2) Select & sort samples: The sample sort algorithm chooses m=2 samples evenly from each sorted chunk and then sorts all these selected samples with the total number of $m \times p$.

(3) Segment partition: From the above $m \times p$ samples, the sample sort algorithm evenly selects *p-1* samples as splitters. These splitters partition the dataset into p segments with non-overlapping ranges.

(4) Segment reorganization: The multiple tasks pick up the segments' candidates from each local sorted chunk according to the value ranges and then sort each segment in parallel. The sample sort completes after combining all the sorted segments.

The sample sort algorithm is suitable for the multicores system because the local sort (step 1) and segment reorganization (step 4) can be executed in parallel. However, each parallel task still sorts a large number of graph edges, which is computation intensive and time-consuming. It also has a synchronization problem of multiple tasks because the sample sort waits for all the parallel tasks to be finished before the next step of processing.

The parallel CISC software optimizes the sample sort algorithm by skipping the local sort (step 1). The in-storage sort circuit divides a large amount of graph edges into chunks of size n_{total}/p each and sorts each chunk of edges in order using hardware. As shown in Figure 16, CISC provides an efficient data distribution for the rest of the sample sort's steps and avoids the local sort of parallel tasks in the host main memory.

In the parallel CISC software of MST, we did not change the original design of graph merge. According to the benchmark baseline [9], the parallel MST starts

to merge after graph sort (sample sort) is completed. It merges sorted edges into the graph subsets with multiple tasks and grows by several sub-trees in parallel. The parallel MST computation finishes when all the sub-trees join together and MST traverses all the graph vertices. As will be shown latter in our experiment, CISC offers overall speedup of MST due to the optimized sample sort.

2.6 Evaluation

In order to evaluate how CISC performs in comparison with traditional approaches, we have built an NVM-e SSD prototype that implements CISC. The hardware chunk sort module is augmented inside the FPGA controller of the PCIe SSD. The PCI-e SSD card is inserted to a multi-core server to carry out a performance evaluation of CISC. This section discusses the prototype setup and evaluation results.

2.6.1 Experimental Platform and Benchmark Selection

We set up the experimental environment on an Intel Xeon processor with 96 cores. It runs at 2.5 GHz and hosts a Linux system with kernel version 4.14. The system contains a PCI-e 3×4 that connects our CISC storage and other peripherals.

We built our CICS prototype on top of the Open-SSD platform [27]. All storage logic fits into Xilinx Zynq-7000 series FPGA, including a dual-core ARM processor, DRAM/flash controller logic, NVM-e interface and CISC's in-storage sort module. The ARM processor runs at 1GHz clock speed, and this platform contains 1GB DDR2 and 256GB flash memory. To evaluate CISC, we store MST benchmark files on SSD before the host starts the MST application. The in-storage sort module is set to sort 128K edges per chunk.

Three benchmark datasets are chosen from [28∼30], including transportation, Internet data analysis and Graph Mining, as listed in Table 2. The PBBS

benchmark [9] source code is used in our design as the baseline to evaluate the performance difference between CISC and the traditional software. We compose CISC software to replace the sample sort and serial MST in the baseline. The parallel software code uses OpenMP configured for multicores.

Table 2. The Benchmark datasets we used in this paper

	Node number	Edge number	Description
US-Road	23,947,347	58,333,344	Transportation
Caida Router	12,190,914	34,607,610	Network
rMatgraph	10,000,000	50,000,000	Graph Mining

2.6.2 Numerical Results and Discussions

Since edge sort is the main part that CISC offers performance advantages for MST computation, we first carried out experiments to measure the execution times of edge sort using CISC and traditional software approach.

Figure 18 plots the speedup of CISC sort over the traditional software sort. As shown in the figure, CISC achieves as much as a 4.6∼6× speedup compared to the pure software sort on the single core. The B-tree algorithm can process smaller edges effectively and finishes MST as soon as the software traverses all the graph vertices. The traditional software sort, on the other hand, needs to sort all the graph edges before the MST merge can start.

The speedup of parallel software sort increases with the increase of the number

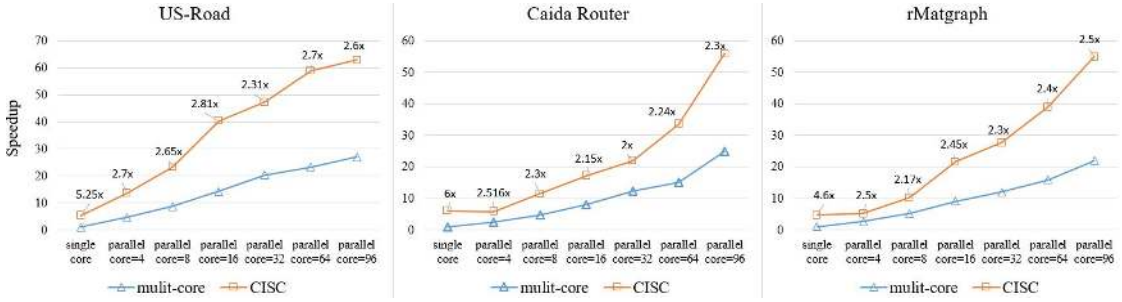

Figure 18. The sort speedup of CISC, the baseline is serial software sort running on single-core.

of cores on the host server. Compared with single core, the speedup increases to 22∼27× as the number of cores increases to 96, as illustrated by the blue line plots in Figure 18. For the same number of cores, our parallel CISC outperforms the traditional software sort. For all the benchmarks considered, we observed 2∼2.81× speedup compared to the traditional software sort with the same number of cores. These speedups can be mainly attributed to the elimination of parallel local sort tasks and partially offloading of computational resources from multi-cores to the SSD. As shown in Figure 18, the parallel CISC sort on 96 multicores shows 55× to 62× speedup compared to the traditional software sort on a single-core.

The overall speedup of the MST application depends on the fraction of sort time over total execution time. For a comparative analysis, we consider the baseline as running MST on a single-core with the in-storage sort module disabled. Figure 19 shows measured results for the benchmarks considered. We observed speedups of 2.2∼2.7× on single-core and a 1.3× speedup on multicores on average. The speedup ratio of a single-core is more significant than multicores because of the time fraction difference of edges' sort. The larger the fraction of graph sort time it takes, the more speedup CISC can obtain. As shown in Figure 12, the sort execution time on single-core consumes 65% to 75% of the overall MST execution, and parallel MST takes 31% to 46% execution time for the graph sort. Thus, the speedup ratio of multicores' MST is less significant than for single-core.

The speedup of parallel MST increases when using more CPU resources of the host server. As shown in Figure 19, CISC always runs faster than the traditional software with the same number of cores. It outperforms purely multicore systems because CISC obtains performance gains from both multicores and the in-storage sort. Compared to a single-core MST baseline, CISC outperforms traditional software by 11.47 to 17.2 times on 96-cores systems.

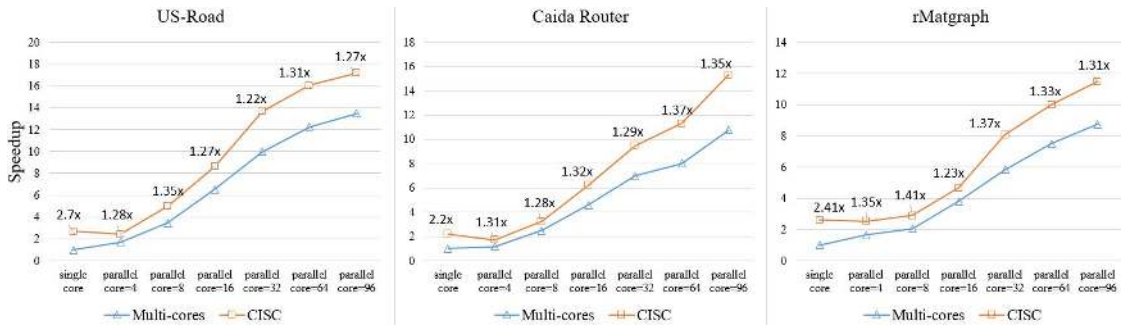

Figure 19. The MST speedup of CISC, the baseline is serial MST running on single-core.

2.6.3 Hardware Cost Analysis

CISC partially offloads the expensive computation from the host server to the SSD. The additional hardware cost of implementing CISC inside an SSD controller includes logic cells, LUT, Flip-flops, and RAM. Table 3 lists the usage of hardware resources of CISC's in-storage sort module, as a fraction of total available hardware resources of FPGA chips.

Our CISC prototype is built on top of the open-SSD platform [27] with Zynq XC7z045 chip. It is not the latest FPGA with limited hardware resources. As shown in Table 3, the in-storage sort module takes 11% of LUT and 41% of RAM resources on Zynq XC7z045 chip. More recent FPGAs doubled and even quadrupled the on chip resources. The Ultra-Scale series FPGA of Zynq and Virtex are commonly used in the modern SSD controllers [31][32]. The hardware cost of CISC becomes insignificant on the latest Ultra-Scale series FPGAs. As shown in Table 3, the hardware resource utilization on such FPGAs is very low. The in-storage sort module of CISC takes 9.3% of LUT and 20% of RAM on the Ultra-scale Zynq. It only takes 1.3% of LUT and 1.7% of RAM on the Ultra-scale Virtex. Therefore, the hardware cost of CISC can be considered negligible on modern FPGAs. The sort module can also be extensible to many sort-based PIS functions and storage ASIC design.

Table 3. Hardware resource utilization of CISC on different FPGAs

	CISC	Zynq dev: XC7z045		Zynq Ultra-Scale dev: ZU9CG		Virtex Ultra-Scale dev: VU13P	
		Total	Used	Total	Used	Total	Used
Logic cells	21.7K	350K	6.2%	600K	3.6%	3780K	0.5%
LUT	25.7K	218K	11%	274K	9.3%	1728K	1.4%
Flip-flop	17.3K	437K	4%	548K	3.1%	3436K	0.5%
RAM	1MB	2.4MB	41%	5MB	20%	57.5MB	1.7%

2.7 Conclusion

In this paper, we have presented a new approach to the MST computation by coordinating computing power inside SSD storage with the host CPU, referred to as CISC. CISC exploits the controller logic inside the SSD to sort graph data while being loaded to the main memory of the host. In order to achieve wire speed, CISC takes a divide and conquer approach by partitioning MST graph edges into chunks and sorts each chunk using hardware. In this way, the MST can then proceed by selecting the smallest edge among the chunks and ensures smaller weight edges can be processed efficiently. To demonstrate the feasibility and performance potential of CISC, we have built a working prototype that consists of both software running on the host and hardware sort module inside the SSD. Extensive experiments have been carried out using real-world benchmarks to demonstrate the feasibility and performance of deploying CISC in NVM-e SSD storage. Our experimental results show a 2.2∼2.7× speedup for the serial version implementation and 11.47× to 17.2× speedup for the parallel version with a 96-core baseline. We believe the PIS function of CISC can be extended to other applications requiring data sort with an addition of similar CISC software module running on the host.

List of References

[1] Zhu, X, Han, W, Chen, W., *GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning* In Proceedings of the Usenix Annual Technical Conference (2015), USENIX Association, pp. 375386.

[2] Zhaoshi Li, Leibo Liu, *Aggressive Pipelining of Irregular Applications on Reconfigurable Hardware*, proceeding of the 44th Annual International Symposium on Computer Architecture ISCA 17.

[3] David A. Bader, Guojing Cong, *Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs* in Parallel Distrib. Comput. 66 (2006) 13661378.

[4] S. Subramanian, M. C. Jeffrey, M. Abeydeera, H. R. Lee, V. A. Ying, J. Emer, and D. Sanchez, *Fractal: An execution model for ne-grain nested speculative parallelism* in ISCA-44, 2017.

[5] S. Rostrup et al., *Fast and memory-efcient minimum spanning tree on the GPU* in processing of IJCSE, vol. 8, no. 1, pp. 21-33, 2011.

[6] S. Manoochehri , B. Goodarzi , D. Goswami *An Efficient Transaction-Based GPU Implementation of Minimum Spanning Forest Algorithm* in processing of High Performance Computing & Simulation (HPCS), 2017 International Conference

[7] C. Luk, R. Cohn, R. Muth et al., *Pin: building customized program analysis tools with dynamic instrumentation* in PLDI, 2005.

[8] Sample sort algorithm: *http://parallelcomp.uw.hu/ch09lev1sec5.html*

[9] PBBS benchmark suit of Minimum spanning tree baseline code reference: *http://www.cs.cmu.edu/ pbbs/benchmarks.html*

[10] R. Balasubramonian, J. Chang, T. Manning, J. H. Moreno, R. Murphy, R. Nair, and S. Swanson, *Near-data processing: Insights from a MICRO-46 workshop* in processing of Micro, IEEE, vol. 34, no. 4, pp. 36-42, 2014.

[11] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, *PRIME: A novel processing-in-memory architecture for neural network computation in reram-based main memory* in Proceedings of the 43rd International Symposium on Computer Architecture. IEEE Press, 2016, pp. 27-39.

[12] M. Gokhale, B. Holmes, and K. Iobst, *Processing in memory: The Terasys massively parallel PIM array* in processing of Computer, vol. 28, no. 4, pp. 23-31, 1995.

[13] S.F. Yitbarek, T. Yang, R. Das, and T. Austin, *Exploring specialized near-memory processing for data intensive operations* in Design, Automation & Test in Europe Conference & Exhibition (DATE), 2016. IEEE, 2016, pp. 1449-1452.

[14] T.S. Chung, D.-J. Park, S. Park, D.-H. Lee, S.-W. Lee, and H.-J. Song, *A survey of ash translation layer* in Journal of Systems Architecture, vol. 55, no. 5, pp. 332-343, 2009.

[15] J.T. Pawlowski, *Hybrid memory cube (HMC)* in Hot Chips 23 Symposium (HCS), 2011 IEEE. IEEE, 2011, pp. 1-24.

[16] J. Ahn et al, *A Scalable Processing-in-memory Accelerator for Parallel Graph Processing* in ISCA-42, 2015, pp. 105117.

[17] A. Acharya, M. Uysal, and J. Saltz, *Active disks: Programming model, algorithms and evaluation* in ACM SIGOPS Operating Systems Review, vol. 32, no. 5. ACM, 1998, pp. 81-91.

[18] K. Keeton, D. Patterson, and J. Hellerstein. *A Case for Intelligent Disks (IDISKs)* SIGMOD Record,27(3):42-52, September 1998.

[19] I.S. Choi and Y.S. Kee, *Energy efcient scale-in clusters with instorage processing for big-data analytics* in Proceedings of the 2015 International Symposium on Memory Systems. ACM, 2015, pp. 265-273.

[20] S.-W. Jun, M. Liu, S. Lee, J. Hicks, J. Ankcorn, M. King, S. Xu et al., *Bluedbm: an appliance for big data analytics* in Computer Architecture (ISCA), 2015 ACM/IEEE 42nd Annual International Symposium on. IEEE, 2015, pp. 1-13.

[21] B. Gu, A.S. Yoon, D.H. Bae, I. Jo, J. Lee, J. Yoon, J.-U. Kang, M. Kwon, C. Yoon, S. Cho et al., *Biscuit: A framework for near-data processing of big data workloads* in Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on. IEEE, 2016, pp. 153-165.

[22] Wei Song, Dirk Koch, Mikel Lujan and Jim Garside. *Parallel Hardware Merge Sorter* In FCCM 2016

[23] Sang-Woo Jun, Shuotao Xu, Arvind. *Terabyte Sort on FPGA-Accelerated Flash Storage* In FCCM 2017

[24] Dirk Koch, Jim Torresen. *FPGASort: A High-Performance Sorting Architecture Exploiting Run-time Reconguration on FPGAs for Large Problem Sorting* In FPGA 2011

[25] Linear sort FPGA design: *https://hackaday.com/2016/01/20/a-linear-time-sorting-algorithm-for-fpgas/*

[26] In-order traverse: *https://en.wikipedia.org/wiki/Tree_traversal*

[27] Open-SSD+ platform: *http://www.openssd.io/.*

[28] Benchmark: *https://www.cc.gatech.edu/dimacs10/downloads.shtml*

[29] DIMACS 10 challenge graph collection Graph Partitioning and Graph Clustering: *http://www.cc.gatech.edu/dimacs10/downloads.shtml.*

[30] D. A. Bader and K. Madduri, *GTgraph: A Synthetic Graph Generator Suite* Technical Report, 2006.

[31] Zynq FPGA: *https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html*

[32] Virtex series FPGA product page: *https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus.html*

[33] H. Tseng, Q. Zhao, Y. Zhou, M. Gahagan, S. Swanson, *Morpheus: Creating Application Objects Efficiently for Heterogeneous Computing*, in Proceedings of the International Symposium Computer Architecture, 44(3): 53-65, ACM/IEEE,2016.

[34] H. Choe, S. Lee, H. Nam, S. Park, S. Kim, E. Chung, S. Yoon, *Near-Data Processing for Differentiable Machine Learning Models*, arXiv:1610.02273v3.

[35] Jinho Lee, Heesu Kim, Sungjoo Yoo, Kiyoung Choi, Peter Hofstee, Gi-Joon Nam, Mark Nutter, Damir A. Jamsek, *ExtraV: Boosting Graph Processing Near Storage with a Coherent Accelerator*, PVLDB 10(12): 1706-1717 (2017)

MANUSCRIPT 3

A Parallel and Pipelined Architecture for Accelerating Fingerprint Computation in High Throughput Data Storages

by

Dongyang Li[1], Qingbo Wang[2], Cyril Guyot[3], Ashwin Narasimha [4], Dejan Vucinic[5], Zvonimir Bandic[6], Qing Yang[7]

is published in the 23rd Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM), Vancouver, Canada, 2015

[1]Ph.D Candidate, Department of Electrical, Computer and Biomedical Engineering, The University of Rhode Island, Kingston, RI 02881. Email: lidongyang@ele.uri.edu

[2]Senior Staff Engineer, HGST, a Western Digital Company, San Jose Research Center, San Jose, CA, 95135

[3]Senior Staff Engineer, HGST, a Western Digital Company, San Jose Research Center, San Jose, CA, 95135

[4]Senior Staff Engineer, HGST, a Western Digital Company, San Jose Research Center, San Jose, CA, 95135

[5]Senior Staff Engineer, HGST, a Western Digital Company, San Jose Research Center, San Jose, CA, 95135

[6]Senior Director, Storage System Architecture, HGST, a Western Digital Company, San Jose Research Center, San Jose, CA, 95135

[7]Distinguished Professor, Department of Electrical, Computer and Biomedical Engineering, The University of Rhode Island, Kingston, RI 02881.

3.1 Abstract

Rabin fingerprints are short tags for large objects that can be used in a wide range of applications, such as data deduplication, web querying, packet routing, and caching. We present a pipelined hardware architecture for computing Rabin fingerprints on data being transferred on a high throughput bus. The design conducts real-time fingerprinting with short latencies, and can be tuned for optimized clock rate with split fresh technique. A pipelined sampling logic selects fingerprints based on the Minwise theory and adds only a few clock cycles of latency before returning the final results. The design can be replicated to work in parallel for higher throughput data traffic. This architecture is implemented on a Xilinx Virtex-6 FPGA, and is tested on a storage prototyping platform. The implementation shows that the design can achieve clock rates above 300 MHz with an order of magnitude improvement in latency over prior software implementations, while consuming little hardware resource. The scheme is extensible to other types of fingerprints and CRC computations, and is readily applicable to primary storages and caches in hybrid storage systems.

3.2 Introduction

Identifying and reducing redundancies in data storage and transmission become more and more important nowadays [1]. One of the common techniques used in locating redundant data is comparing sketches of data chunks to find duplication or similarity. A sketch typically consists of a few fingerprints representing a data chunk [2]. Rabin fingerprint has proved to be very effective and is widely used in forming such a sketch [2]. To derive a sketch, a data chunk is scanned a shingle by shingle, a fix-sized window, (e.g. 8 bytes long) that shifts forward one byte every step. A Rabin fingerprint is calculated for each shingle. A random sampling technique, such as Minwise theory [3], is then used to select a few among all Rabin

fingerprints as a sketch for the data chunk.

Deriving such sketches is computationally intensive. For example, to obtain a sketch of 4KB data chunk with a shingle size of 8 bytes, $4K$-7 Rabin fingerprints need to be calculated and the sampling process is also time consuming. Existing software programs typically take around 30 microseconds to generate a sketch for each 4KB data chunk on a commercial CPU [4]. For data deduplications in data backup and archive applications, such a delay might be tolerable. However, with todays storage devices approaching gigabyte per second in throughput and sub-milliseconds in latency [5], this delay is inadequate for real-time data processing for primary storages and storage caches.

This paper presents a hardware approach to Rabin fingerprint computation and sampling to produce a sketch for a data chunk. By means of effective pipelining and split fresh technique, our hardware implementation is able to achieve one order of magnitude speedup over the existing software implementation [6]. Moreover, the design consumes 2$\sim$10 times less hardware resource than a comparable configuration of the existing HW solution [7]. Our design also overcomes the drawback of [7] that has a latency linearly increasing with data input size. A working prototype of our new design has been successfully implemented on an FPGA and tested to work properly at clock rate above 300 MHz. The architecture is configurable according to the characteristics of the input data, and a single unit of the design can be replicated to work in parallel accommodating higher throughput demand.

The paper is organized as follows. Section II provides the background and a preview on the pipeline architecture. Section III explains the overall design as well as the optimizations. Implementation experiences on an FPGA board are shared in Section IV along with its performance evaluation. Section V concludes

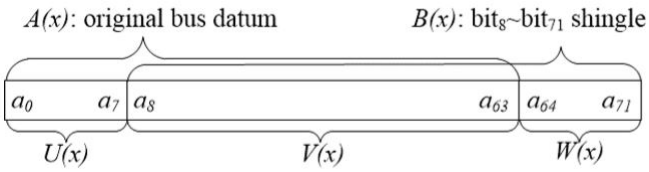

Figure 20. The diagram of shingles.

the paper with future plans.

3.3 Background and architectural overview

The Rabin fingerprint scheme considers an *n*-bit message $m=(m_0,m_1,m_2...m_{n-1})$ represented by *f(x)* in (1), a degree *n-1* polynomial over GF(2). A random polynomial *p(x)* , not necessarily irreducible, is picked over the same field with degree *k-1*, as in (2). The remainder *r(x)* of dividing *f(x)* by *p(x)* over GF(2), a *k*-bit number, is returned as the fingerprint of the message *m*. This process is shown in (3).

$$f(x) = m_0 + m_1x + m_2x^2 + m_3x^3 + ... + m_{n-1}x^{n-1} \quad (1)$$

$$p(x) = p_0 + p_1x + p_2x^2 + p_3x^3 + ... + p_{k-1}x^{k-1} \quad (2)$$

$$r(x) = f(x)mod(p(x)) = r_0 + r_1x + r_2x^2 + ... + r_{k-1}x^{k-1} \quad (3)$$

In the formal algebra system, a single modulo operation can be turned into multiple calculations, each of which is responsible for one bit in the result. Such scheme, normally involving just *XORs*, is suitable for hardware implementations. We group these bit-wise calculations to form a computational module for Rabin fingerprints, and call it the *fresh* function.

Figure 20 illustrates two consecutive shingles when scanning a data block to compute fingerprints. The bits from a_0 to a_{63} form the first shingle denoted by *A(x)* and the bits from a_8 through a_{71} form the second shingle denoted by *B(x)*. Shingles can shift in multiple bytes other than just one byte to speed up

fingerprinting. Once we have the fingerprint for *A(x)*, the fingerprint of *B(x)* can also be expressed by

$$BmodP = (W \times x^{56} - U \times X^{-8})modP + (X^{-8} \times (AmodP))modP \qquad (4)$$

As shown in (4), the fingerprint of the second shingle *B(x)* can be obtained using the fingerprint of the first shingle *A(x)*, the first byte, *U(x)*, of the prior shingle and the last byte, *W(x)*, of the current shingle. We call this formula the *shift* function, which generally leads to a simpler design than the *fresh* function, and should consume less resource when being implemented in hardware.

3.3.1 Pipelining with *Fresh* and *Shift* stages

Based on the above analysis, a pipelined architecture can be drawn using *fresh* and *shift* as shown in Fig.21. In the example of a 64-bit wide data bus and a 64-bit shingle, the data in the illustration comes from back-to-back clock cycles, where *(a_0, a_1,..a_{63})* is from the proceeding cycle, and *(a_{64}, a_{65},..a_{119})* from the following cycle. The *fresh* treats the first shingle *(a_0, a_1,..a_{63})* at Stage 0. The shifts in the following stages generate fingerprints for their corresponding shingles utilizing the results from the previous shingles, the *evicted* byte from the beginning of last shingle, the *absorbed* byte from the end of its own shingle. It should be noted that the entire data from the following clock, i.e, *(a_{64}, a_{65},..a_{127})* is treated by the fresh function at Stage 0 during its arrival.

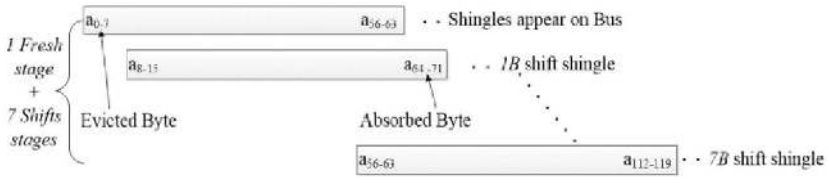

Figure 21. Pipeline with the fresh and shift stages.

Further optimization is possible by directly splitting the fresh function into multiple sub-functions, and hence multiple stages in the pipeline. For example, we can express the *fresh* function as a bit-wise *XOR* of *fresh1* and *fresh2*, where *fresh1* draws all of its bits only from *(a_0, a_1,..a_{38})* of the *fresh*, and *fresh2 (a_{39}, a_{40},..a_{63})* in the 64-bit example above. Table 4 lists the complexity of the individual split *fresh* modules, the combined of the two, and that of the original single *fresh* function, given the polynomial *p(x)= x^{16} + x^{13} + x^{12} + x^{11} + 1*. While the resource consumption may not change much at the end, the clock rate should improve for the case of the split *fresh* due to more and simpler pipeline stages.

Table 4. Design Complexity Comparison

Logic Utilization	*Fresh1*	*Fresh2*	*Split combined*	*Original fresh*
Fan-in	13	11	13	24
Fan-out	7	9	9	11
XORs	139	143	368	362

3.3.2 Sampling of Fingerprints

The total number of fingerprints generated for a w-byte data chunk in our application will be *w-b+1*, where b is the size of the shingles. After all Rabin fingerprints are computed for a block, a number of fingerprints are chosen as a sketch to represent the block. Udi Manber [8] provided two methods to decide which fingerprints to select. One is selecting fingerprints that have their last n bits being all zeros. The other is selecting fingerprints according to some keyword because keywords are in a sense universal and they are selected truly at random. Broder showed a scheme based on Minwise theory [3]. Minding the principle of random sampling, to select Rabin fingerprints with the upper N bits being a specific pattern shall present a fairly good approximation because these upper bits in each fingerprint can be considered as randomly distributed. We choose this scheme for

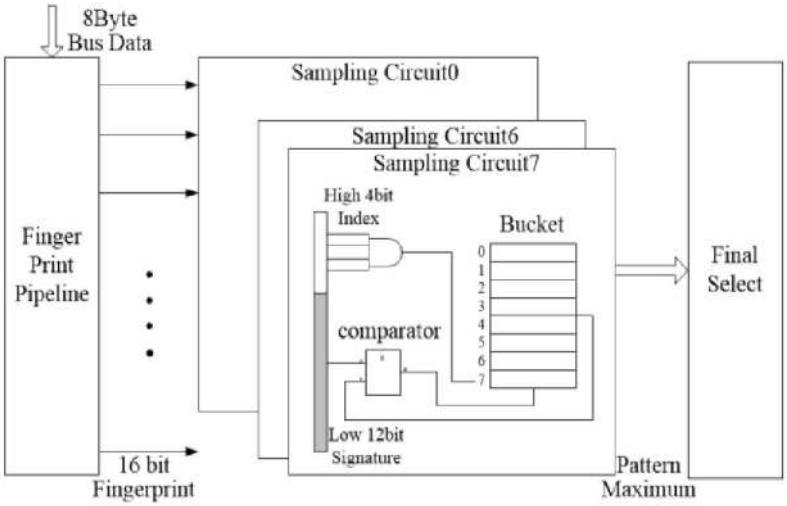

Figure 22. Design with fingerprint pipeline and signature selection.

its processing speed, and similarity detection qualities [1], as will be discussed in Section III B.

3.4 Designe and optimization

Our design is illustrated in Figure 22 with three major function modules: Rabin fingerprint computation pipeline, channel sampling, and final selection. The data, 8 bytes per clock cycle in this diagram, flows from top to bottom through the Rabin fingerprint pipeline. The fingerprints produced at every pipeline stage are sent rightward to the corresponding channel sampling units. As the data chunk runs through the pipeline, the fingerprints are sampled and stored in the intermediate buffer of the channel units. When the sampling for a data chunk is done, the final selection unit then chooses from the intermediate samples and returns a sketch for the data chunk.

3.4.1 Rabin Fingerprint Pipeline Design

The Rabin fingerprint pipeline in Figure 23 has two split fresh stages followed by seven shift stages. The two fresh modules compute the fingerprint FP_0 for the eight bytes of data from the proceeding clock. The first seven bytes from the

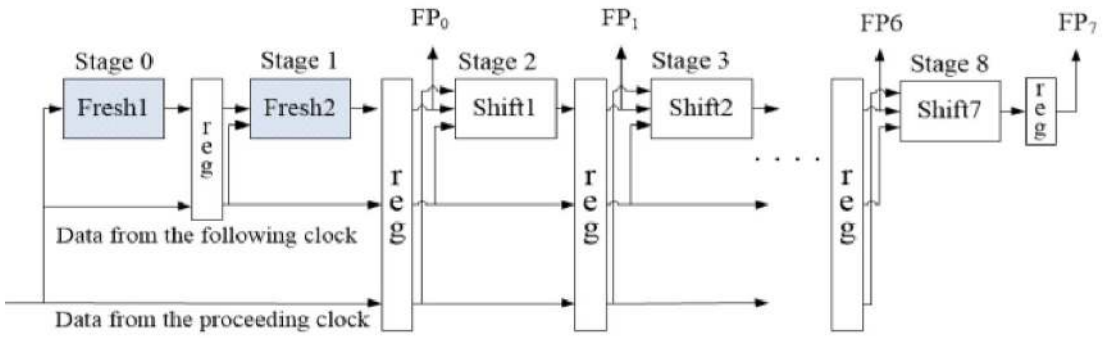

Figure 23. Rabin fingerprint pipeline.

proceeding clock and the first seven bytes from the following clock are passed via pipeline registers to Stage 2, where shift1 conducts fingerprint computation for its shingle. After the computation is done there, the *evicted* and *absorbed* bytes are dropped. Continuously this way, the size of the pipeline registers decrease by two bytes every step forward, until there are no *evicted* and *absorbed* bytes at the end of the pipeline. Each clock takes turn as the proceeding clock, and its data goes through the *Fresh* units during its time.

Compared to a pipeline with a single fresh unit, this design introduces one more cycle latency to the final result, which is not detrimental to the system performance. If needed for higher clock rate, the fresh, as well as the shift can be further split into more stages.

3.4.2 Channel Sampling and Final Selection

During sampling, each computed fingerprint is divided into two parts: index and signature, where the index is a few of MSBs, and the signature the remaining LSBs. Say the index has m bits, then the signatures can be categorized into 2^m bins. Within a bin, the signatures are selected as candidates for the final sketch. For a channel sampling unit, there can be up to 2^m candidates for the final selection.

Figure 24(a) shows the design of each channel sampling unit in our Rabin-*16* example. Four MSBs address the buffer where the selected signatures are

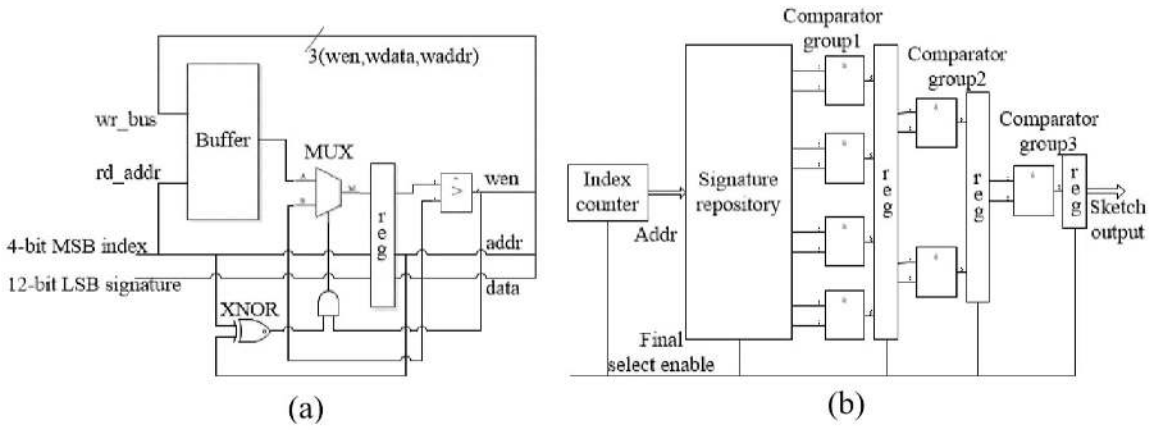

Figure 24. Channel sampling (a) and Final selection (b).

stored. The comparator, generating the write enable signal, decides either the minimum or maximum value is sampled into the buffer. To avoid RAW hazards, a data forwarding function is adopted to control which value to compare with the incoming signature. The *XNOR* gate checks whether the read address and the write address clash. If they do, and the write enable is active at the moment, the current write value will be forwarded to the comparator. This forwarding is done by the *MUX* controlled by the output of the *AND* gate.

When all signatures are processed with the candidates settling in the channel buffers, the final selection unit activates the index counter to fetch the candidates according to a pre-defined index sequence, such as *0, 1, 3, 5, 7, 11, 13*, and *15* in our design. Taking advantage of concurrently available buffers, and with the pipeline registers between the comparators, the final selection in Figure 24(b) conducts a binary tree reduction over the candidates.

3.4.3 Parallel Pipelines

The pipeline design can be duplicated to accommodate a data bus wider than the defined shingle size. Suppose the input data comes in at 16 bytes per clock, and the shingle size remains 8 bytes. The data can be divided into low 8 bytes and

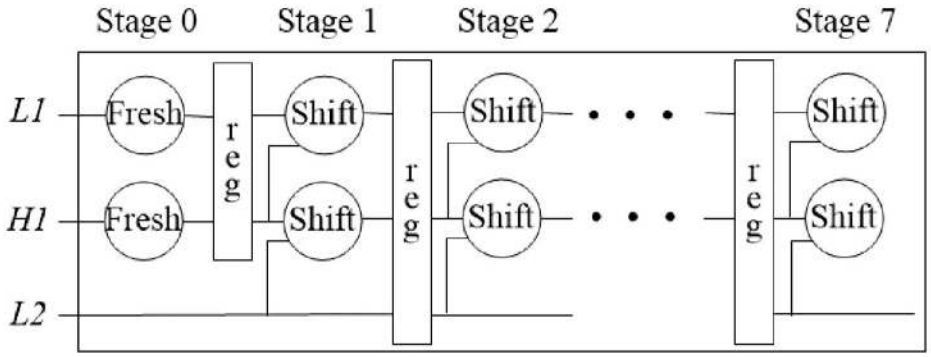

Figure 25. Parallel pipelines.

high 8 bytes, denoted by L1 and H1 in Figure 25. Similarly, L2 refers to the lower 8 bytes from the following clock. L1, H1, and L2 are fed into the pipelines, where L1 and H1 go through the upper pipeline producing eight sets of fingerprints, and H1 and L2 go through the lower pipeline. Note that L2 will pair up with H2 going through the upper pipeline in the following clock. In this fashion, each stage produces two fingerprints during a clock cycle.

3.5 Implementation and evaluation

Our fingerprint design is a part of a primary storage prototype that is implemented on a Xilinx ML605 board. As seen in Figure 26, a host PC reads from and writes to the storage media via an NVMe interface [9][10].

3.5.1 Hardware Implementation Evaluation

Using the example polynomial, we implemented three designs, i.e. with pipeline having a single stage fresh, a two-stage split fresh, and replicated eight

Table 5. Synthesis Report (Fingerprint + Sampling)

Logic Utilization	*Replicate eight copies of fresh*	*Pipeline with single fresh*	*Pipeline with split fresh*
number of slice registers	1435	898	938
number of slice LUTs	2797	823	760
number of LUT-FF pairs	3140	1041	919
maximum clock frequency	213 *(MHz)*	279 *(MHz)*	301 *(MHz)*
latency of clock cycles	12	18	19

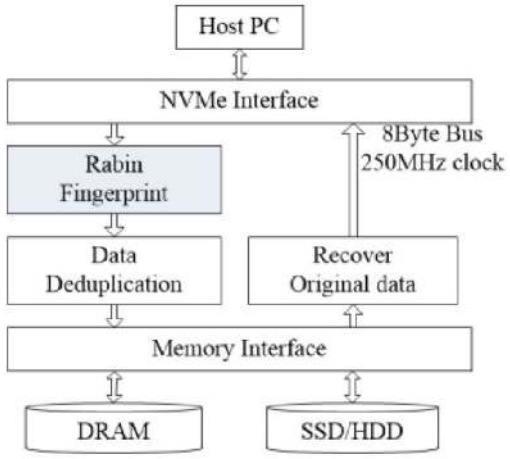

Figure 26. Primary storage prototype.

copies of fresh. Table 5 first shows the resource consumptions of the three designs, all of which include the channel sampling and final selection modules. Excluding those modules, our Rabin fingerprint pipeline exhibits a 2∼10 times saving in hardware resources compared to existing designs. Moreover, with 30∼45% clock rate improvement compared to the nave HW design, our design can run way above the required 250 *MHz* system clock frequency, while replicated fresh is not able to support this clock rate. The latencies for our design exhibit a 20 nanoseconds overhead compared to replicated eight copies fresh. This difference does not affect the performance of primary storage applications.

The split fresh design uses less LUTs and a little bit more registers compared to the single fresh design. However, the implementation does run at higher clock rate because the delays are more uniform across all stages in the pipeline. This improvement is consistent with our analysis in Section II.A, and the scheme offers a promise for possibly higher clock speed.

3.5.2 Software Comparison

We further implemented the software design in [6] on *2.8 GHz* Intel Core i5 processor with *2GB* DRAM. The computation utilizes a sliding window based

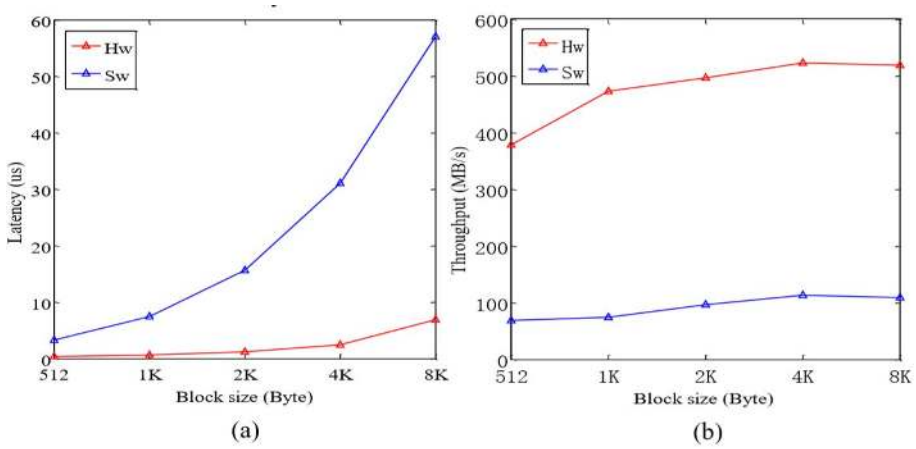

Figure 27. Hardware and software comparison: (a) Latency (b) Throughput.

Rabin fingerprint library to process the same sets of data used in our hardware experiments. We constrained the program to run on one core only, and compare the results with that of our hardware module implemented on the FPGA.

Figure 27(a) plots the latency as a function of the data block size. The software latency is measured from when the data chunk is ready in the memory to the finish of sketch generation. The hardware latency measurements begin with data arriving at the module and finishes after the result returns. The figure shows that our hardware implementation has a clear advantage with the latency difference increasing along the size of data blocks. The measured throughput in Figure 27(b) also shows $\sim 5X$ improvement on the hardware over the software solution.

3.6 Conclusion and future works

The proposed hardware approach for fingerprinting large data objects can operate at wire speed. The major techniques include fresh/shift pipelining, split fresh optimization, online channel sampling, and pipelined final selection. Demonstrated on FPGA using Rabin fingerprint, the whole computation adds just a few clocks latency to the data stream. Measured throughput satisfies the requirement of primary storages. The architecture is extensible to other types of CRC and

fingerprint computations, and can be adapted to large shingle sizes and wide data buses.

Future optimization can still be achieved by streamlining the final selection to reduce latency. Or by shingling more than one byte, and interleaving the shingled bytes, we should be able to make the single pipeline itself to a parallel one.

List of References

[1] Lior Aronovich, Ron Asher, Eitan Bachmat, Haim Bitner, Michael David Hirsch, Shmuel Tomi Klein, *The design of a similarity based deduplication system*, in SYSTOR '09': The Israeli Experimental Systems Conference. no. ACM, p. 6, 2009.

[2] A. Z. Broder, *Some applications of Rabins fingerprinting method*, in Communications, Security, and Computer Science, 1993, p. 143152.

[3] A. Z. Broder, M. Charikar, A. M. Frieze and M. Mitzenmacher, *Min-Wise Independent Permutations*, in Computer and System Sciences, pp. 21-29, 1998.

[4] Q. Yang, and J. Ren, *I-CASH: Intelligently coupled array of ssd and hdd*, in High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on, IEEE, 2011, pp. 278-289.

[5] HGST product page: *https://www.hgst.com/*

[6] Brandon Amos, detect sensitive data before commits by using fuzzy Rabin fingerprints, *https://github.com/bamos/safegit*

[7] R. Ramaswamy, L. Kencl and G. Iannaccone, *Approximate fingerprinting to accelerate pattern matching* in Proceedings of the 6th ACM SIGCOMM conference on Internet measurement New York, 2006.

[8] U. Manber, *Finding similar files in a large file system* in Proceedings of the USENIX winter 1994 technical conference, San Fransico, 1994.

[9] A. Huffman, *NVM-express: going mainstream and what's next*, Intel IDF 2014

[10] NVM Express: *http://www.nvmexpress.org/*

MANUSCRIPT 4

Hardware Accelerator for Similarity Based Data Dedupe

by

Dongyang Li[1], Qingbo Wang[2], Cyril Guyot[3], Ashwin Narasimha[4], Dejan Vucinic[5], Zvonimir Bandic[6], Qing Yang[7]

is published in the 2015 IEEE International Conference on Networking, Architecture and Storage (NAS), Boston

[1]Ph.D Candidate, Department of Electrical, Computer and Biomedical Engineering, The University of Rhode Island, Kingston, RI 02881. Email: lidongyang@ele.uri.edu

[2]Senior Staff Engineer, HGST, a Western Digital Company, San Jose Research Center, San Jose, CA, 95135

[3]Senior Staff Engineer, HGST, a Western Digital Company, San Jose Research Center, San Jose, CA, 95135

[4]Senior Staff Engineer, HGST, a Western Digital Company, San Jose Research Center, San Jose, CA, 95135

[5]Senior Staff Engineer, HGST, a Western Digital Company, San Jose Research Center, San Jose, CA, 95135

[6]Senior Director, Storage System Architecture, HGST, a Western Digital Company, San Jose Research Center, San Jose, CA, 95135

[7]Distinguished Professor, Department of Electrical, Computer and Biomedical Engineering, The University of Rhode Island, Kingston, RI 02881.

4.1 Abstract

Deduplication has proven essential in backup storage systems as large amount of identical or similar data chunks exist. Recent studies have shown the great potential of data deduplication in primary storage and storage caches [1]. For such application scenarios, processing speed for similar data chunks becomes more important to the system success. This paper presents a FPGA accelerator for similarity based data deduplication. It implements three hardware kernel modules to improve throughput and latency in dedupe system: block sketch computation, reference block indexing and similar block delta compression. The accelerator connects to the host system through a PCIe Gen 2×4 interface. By means of pipelining and parallel data lookup across multiple hardware modules, our new HW design is capable of processing multiple data units, say 8-byte long, in parallel every clock cycle and therefore provides line speed similar block dedupe. Our experiments have shown that the similarity based data dedupe performs 30% better in data reduction than the conventional dedupe techniques that only look at identical blocks. Compared our hardware implementation with its software counterpart, the experimental results show that our preliminary FPGA implementation with clock speed of 250 MHz provides at least 6 times speedup in throughput or latency over software implementation running on the state-of-art servers.

4.2 Introduction

Data deduplication has become increasingly important due to explosive data growth in the Internet world. It has been highly successful in enterprise backup environments [2]. Typically, companies execute daily incremental backups and weekly full backups to protect their data. The great amount of duplicate data drives widespread use of deduplication in enterprise backups.

The success of data deduplication in backup systems inspired a large amount

of efforts in primary storage deduplication. Unlike the backup system, the primary deduplication is used in a production environment [3] [4], which brings in multiple challenges. Firstly, a primary storage does not have as much duplicate data as in the backup systems. Data sent to primary storage comes from user level applications, such as database and MS-Office. The main operations for these kinds of applications are modify, add and delete. These operations generate a lot of similar data blocks as opposed to duplicated blocks, making it more sensible to look at deduplications at sub-block level. The second challenge is the performance requirement. Backup storage deduplication is throughput sensitive while the primary storage is mainly used in production environment and is latency sensitive. The required response time for each data unit is much shorter than backup dedupe systems. The last challenge is the limitation of resources. Primary storage deduplication system often shares the production environments resources while backup deduplication system has its own resources. Taking server resources such as the CPU and the RAM resources to perform deduplication may drag down application performance running on the server, which is undesirable.

Files or data blocks are frequently modified and reassembled in different contexts and packages. By deriving the differences between near-duplicate data blocks, delta compression can effectively dedupe data at both file or block levels. The central task of delta compression is to find difference content between two data chunks, and try to only keep them. Philip Shilane et al built a delta compression and dedupe storage [2]. The extra deduplication benefit gains owing to delta compression is 1.4 times compared to the conventional dedupe techniques. However, the throughput of the system is ranging only from 30MB/s to 100MB/s which are not suitable for primary storage or cache systems that demand close Gigabyte per second throughput and submillisecond in latency.

In order to make similarity based dedupe applicable to primary storages or caches, hardware acceleration should be explored. A hardware implementation not only can offer high speed dedupe, but also offload dedupe functions from servers so that application performance is not negatively affected. In this paper, we present the first hardware design, to our knowledge, for similarity based dedupe for primary storages and storage caches. By means of pipelining and parallel structures, our design provides high throughput and fast response time. The proposed architecture was implemented on a Xilinx Virtex-6 FPGA development board. Three major hardware modules for the dedupe system were fully tested to be functional. Extensive experiments have been carried out to evaluate their performance and compression ratio as compared to software implementations. Our experimental results show that the hardware implementation provides at least 6 times speedup, over its software counterpart while the compression ratio is comparable. We also show that similarity based dedupe offers 30% better data reduction ratio than the typical dedupe techniques.

This paper makes the following contributions:

1) Design and implementation of hardware solutions for three major modules of similarity based data deduplication: fingerprint computation to derive the sketch of a data block; indexing structure and search logic for finding reference blocks that are used as bases for delta compressions; and hardware delta compression logic.

2) Integration of the hardware modules into software dedupe platform [5]. The integrated system is shown to function correctly and efficiently.

3) Performance evaluations have been carried out using real world data sets. We conducted extensive experiments to show the achievable speedup and data reduction ratios as compared to existing solutions.

The rest of this paper is organized as follows. In Section 2, the related back-

ground work is presented. Section 3 presents our design of the 3 hardware modules. The FPGA implementation, the test setup, and the experimental results are detailed and discussed in Section 4. We conclude our paper in Section 5.

4.3 Background

4.3.1 Standard dedupe

A typical process of data deduplication involves the following processes. Firstly, it splits files into multiple chunks and generates a fingerprint for each chunk. The fingerprint usually is a strong hash digest of the chunk. If two fingerprints match, it means their contents are duplicate. When a new incoming chunk's fingerprint matches an existing one in deduplication system, only the chunk's metadata such as file name or LBA and a reference to the existing content will be stored [6].

4.3.2 Similarity based dedupe

It is often the case that data chunks are frequently modified by cut, insert, delete, or update a part of the content. Though slightly changed chunk will generate different strong hashes and could not be indexed by standard dedupe, the sketch of the chunk may stay the same if a weaker hash function is used [7]. Such weaker hash sketches typically consist of several Rabin fingerprints and have the property that if two chunks share a same sketch then they have a lot of same content, i.e. they are likely near-duplicates. Note that we will use the terms "chunk" and "block" interchangeably in this paper to refer to the basic unit of data deduplicaiton.

In similarity based deduplication, a new block searches for a near-duplicate block in a set of reference blocks by comparing their sketches. If a matched sketch is found in a list of reference blocks, a delta compression is performed against the found reference block and only the delta is stored with a pointer to the reference

block. Therefore, similarity based dedupe requires three key functions: 1) computing the sketch of a block; 2) select and store reference blocks against which the delta compression will be performed after a matched sketch is found; 3) delta compression.

4.3.3 Delta compression

For two near-duplicate files f_{old} and f_{new}, delta compression is to compute a minimal size of f_{delta} that new could be reconstructed by f_{old} [8]. Delta compression constructs a dictionary of observed sequences, and looks for repetitions as it goes. It writes the number of the dictionary entry when a repetition encountered, and store the unique token if no match happened. The output thus consists of appropriately labeled f_{new} and references to f_{old} repetitions.

Though extensive work has been done on hardware compression, none of them were designed specifically for delta compression in dedupe system [9, 10]. Also, current hardware based delta compression has to compress data chunk f_{new} byte by byte. It takes $4K$ loops to compute f_{delta}, which may form a performance bottleneck for high throughput storage systems. I/O buses are usually more than one byte in width. A compression unit whose latency increases linearly with the input width is not acceptable for modern data storage applications. Inspired by WK algorithms for Compressed Caching in Virtual Memory Systems [11], we choose multiple bytes as the token size that can be processed in parallel hardware for delta compressions.

4.4 Design and optimization

4.4.1 Compute Sketches

To derive a sketch, a data chunk is scanned shingle by shingle, a fix-sized window (e.g. 8 bytes long), that shifts forward one byte every step as shown in Figure 28. A Rabin fingerprint is calculated for each shingle scanned. In the formal algebra system, Rabin fingerprint computation can be turned into multiple calcula-

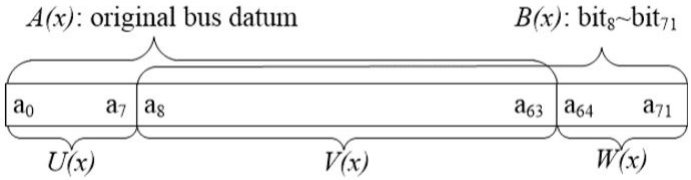

Figure 28. An example showing two shingles.

tions, each of which is responsible for one bit in the result. Such scheme, normally involving just *XORs*, is suitable for hardware implementations [12]. We group these bitwise calculations to form a computational module for Rabin fingerprints, and call it the *"fresh"* function.

Within two consecutive shingles shown in Figure 28, bits from a_0 to a_{63} form the first shingle denoted by *A(x)* and the bits from a_8 through a_{71} form the second shingle denoted by *B(x)*. Shingles can shift in multiple bytes other than just one byte to speed up fingerprinting. Once we have the fingerprint for *A(x)*, the fingerprint of *B(x)* can also be obtained as follows,

$$BmodP = (W \times x^{56} - U \times X^{-8})modP + (X^{-8} \times (AmodP))modP \qquad (5)$$

As shown in Equation (1), the fingerprint of the second shingle *B(x)* can be obtained using the fingerprint of the first shingle *A(x)*, the first byte, *U(x)*, of the prior shingle and the last byte, *W(x)*, of the current shingle [13]. We call this formula the *"shift"* function, which generally leads to a simpler design than the fresh function, and should consume less resources when being implemented in hardware. Further optimization is possible by directly splitting the fresh function, *"split shift"*, into multiple sub-functions, and hence multiple stages in the pipeline.

Based on the property of *"fresh"*, *"shift"* and *"split shift"*, we designed a Rabin fingerprint pipeline which provides a line speed sketch computation. As shown in Figure 29, it has two split fresh stages followed by seven shift stages. The two fresh modules compute the fingerprint FP_0 for the eight bytes of data from the

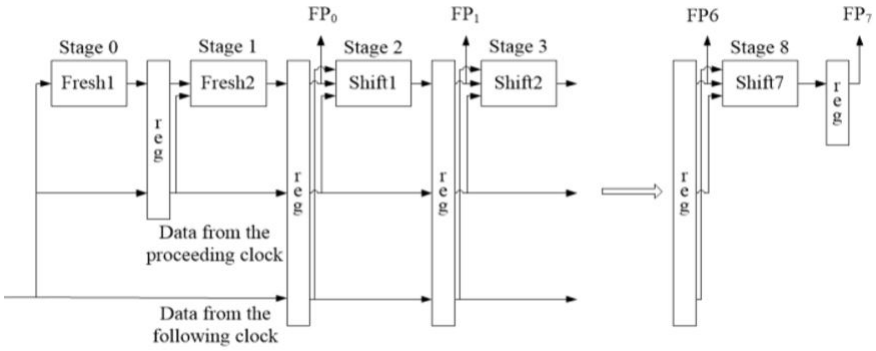

Figure 29. Rabin fingerprint pipeline.

proceeding clock. The first seven bytes from the proceeding clock and the first seven bytes from the following clock are passed via the pipeline registers to Stage 2, where shift1 conducts fingerprint computation for its shingle. After the computation is done there, the evicted and absorbed bytes are dropped. Continuously this way, the size of the pipeline registers decreases by two bytes every step forward, until there are no evicted and absorbed bytes at the end of the pipeline. Each clock takes turns as the proceeding clock, and its data goes through the fresh units during its time.

A random sampling technique, such as Minwise theory [14], is then used to select a few among all Rabin fingerprints as the sketch for the data chunk. As shown in Figure 30, sketch is generated after fingerprints are produced at every pipeline stage and are sent rightward to the corresponding channel sampling units. As the data chunk runs through the pipeline, the fingerprints are sampled and stored in the intermediate buffer of the channel units. When the sampling for a data chunk is done, the final selection unit then chooses from the intermediate samples and returns a sketch for the data chunk.

4.4.2 Reference block index

After the sketch of each block is calculated, we use the sketch to represent each data block and keep track of I/O access patterns to all sketches. Based on

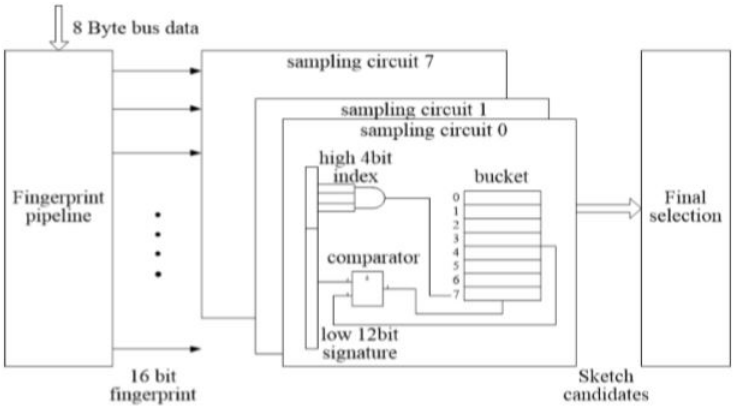

Figure 30. Block diagram of hardware design for sketch computation with fingerprint pipeline and sketch selection.

the content locality, i.e. access frequency and recency of data contents [15], we select and cache two thousands most popular blocks as reference blocks. These reference blocks and their sketches are stored in a reference list. Every newly generated block sketch is used as a key to search the reference list to find a match. The new block is then delta compressed against the matched reference block. The compressed delta and a pointer are stored in the primary storage or cache rather than the original *4 KB* block.

In our design, we assume that a sketch contains 8 fingerprints each of which is one byte long. If two data blocks have n matched fingerprints between their respective sketches (n from 4 to 8), we consider they are near duplicate blocks. n is referred to as similarity threshold. Once such near duplicate block is found in the reference index, the corresponding reference block will be read out and delta compression against it is performed.

For every two 8 fingerprint sketches, there are C_8^n possible n byte in order match of fingerprints. Every reference block's sketch gives C_8^n different n byte permutations. We create a cuckoo hash for each permutation. Inspired by FPGA hash table design in [16], Figure 31 depicts our cuckoo hash engine for similar block index. The lookup starts by parallel computing of hash keys for n bytes sketch

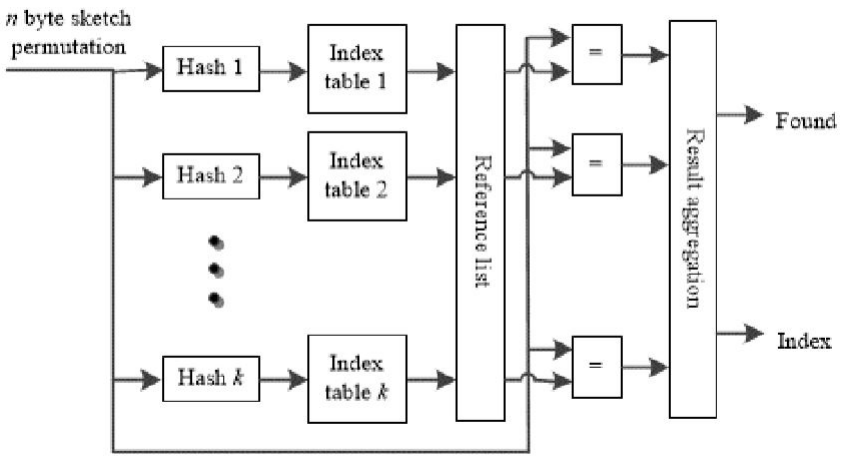

Figure 31. Block diagram of hardware cuckoo hash search engine.

permutations using CRC implementation with a 13-bit polynomial. In cuckoo hash index table, each record forms a pair composed of a hash key and an index to the reference list. Subsequently, the input sketch is compared with the reference sketch that shared same n bytes permutation. Cuckoo hash splits n byte permutations into multiple tables that each unique key appears only in a single place at a time. If none of the n bytes permutations is equal to the searched reference sketch, the found flag is cleared, otherwise it is set and a similar block is found.

Taking advantage of FPGA's parallel computation, our hardware design for reference index module allows parallel search among all the C_8^n parallel paths at same time. Once a match case is found, reference index can spot the location in the reference list as illustrated in Figure 32.

4.4.3 Delta compression

The PCI-e bus connecting our hardware platform to the host is 8 Bytes in width. In order to provide a line speed compression for similarity based data deduplication, we looks for every 8 Bytes repetitions from near duplicate blocks. Figure 33 illustrates delta compression process of two data blocks: Blk_{ref} is a reference block to be delta compressed against. It is loaded into the dictionary

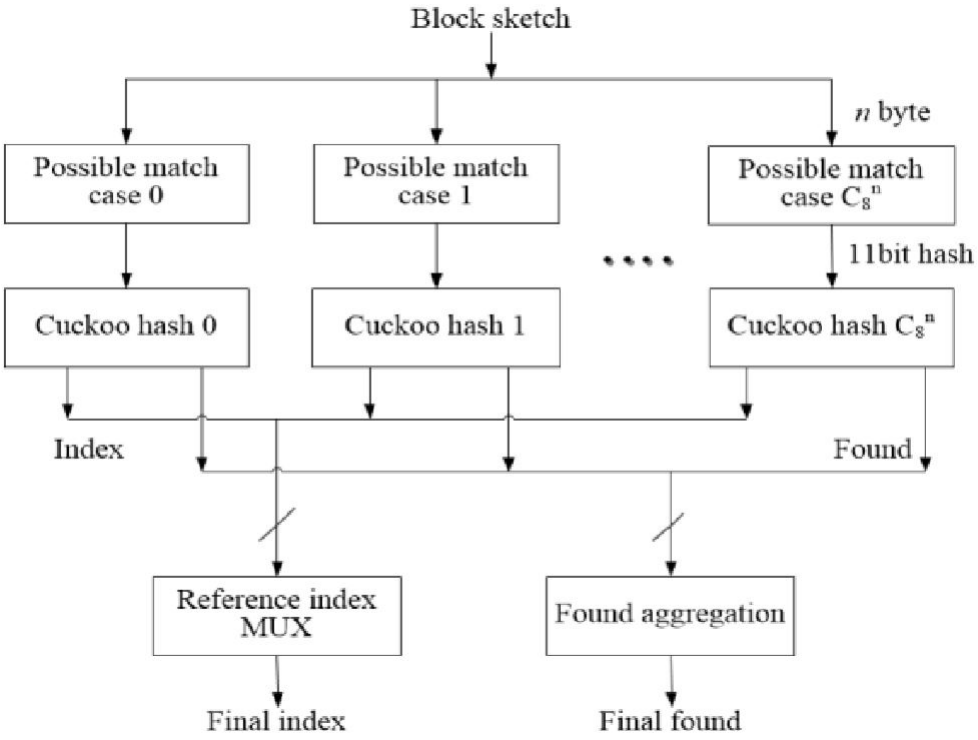

Figure 32. Parallel search structure for reference block index.

first. Blk_{new} is the associated block to be compressed. It is fed into the delta compressor following the reference block. While the two blocks are feeding into the compressor, repetitions between the two blocks were searched. As shown in Figure 33, as the two blocks were scanned, encoded data form the output of the compression. The encoded data consists of an offset, a flag, an index, and the length. For example, in data block Blk_{new} data words *Dw1* and *Dw0* matched data stream starting from position 7 with length 2 in reference block Blk_{ref}. And hence the encoded output is *(1,1,7,2)* as shown in the figure. However data word *Dw4* in *(3,0,Dw4)* has no match in Blk_{ref}, the encoded out is then Blk_{new}. In general, encoder output contains four fields of information while scanning data block: The flag field indicates whether the current string has a match in the Blk_{new}. It is set to 1 if a match string is found in Blk_{ref} for the current string being scanned and clear to 0 if no match is found; The index field gives the starting position of the

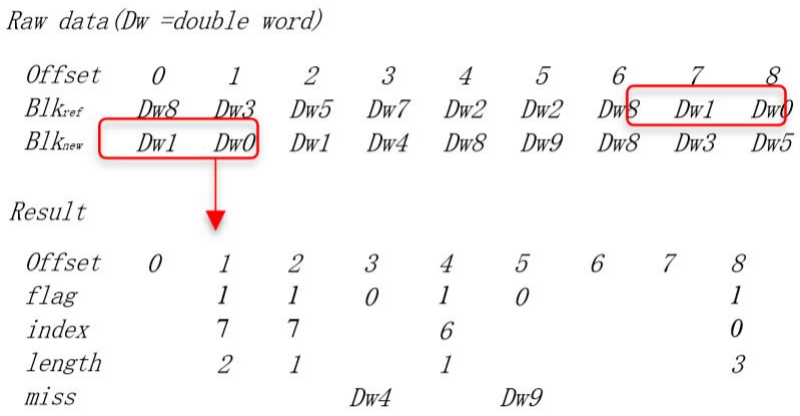

Figure 33. An example showing hardware delta compression encoding.

matched string in Blk_{ref}; The length field records the total length of the matched string. The miss field records the data which does not appear in Blk_{ref} when the flag filed is 0.

Two parallel pipeline structures were designed as shown in Figure 34. One pipeline builds the dictionary using the reference block while the other scans the incoming data block to be compressed. The matched reference block, Blk_{ref} takes the reference pipeline to load the reference data into the dictionary. The associated block, Blk_{new}, goes to compression pipeline and provides line speed delta compression encoding.

In order to do a quick search in the reference dictionary, we need a Content Addressed Memory (CAM) architecture to associatively search for 8 Bytes matched data string in Blk_{ref}. Though CAM in FPGA is an easy solution for fast data search as compared to other memory implementations and offers parallel content comparison to find a valid address, it requires excessive amount of hardware resources [17]. Instead, we use a hash function, a hash table RAM, and reference dictionary RAM to replace CAM IP core in FPGA.

The dictionary RAM has 512 entries and stores every 8 byte data of Blk_{ref} sequentially. To avoid linear search, we use another block RAM to build the hash

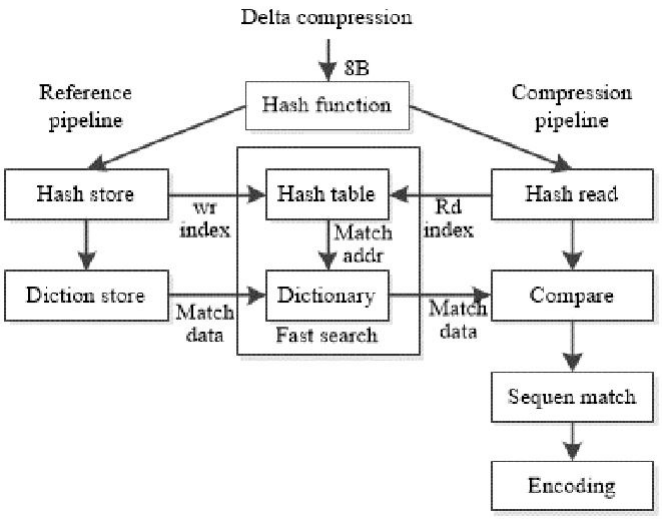

Figure 34. Delta compression engine for every 8-byte data quantum.

table. It has 2^{11} entries representing 11 bits hash values of an 8 byte input. A hash entry corresponds to 9 bits reference address value (from 0 to 511 dictionary index).

After the reference block finishes updating the hash table and the dictionary, the Blk_{new} goes through the compression pipeline so that it does a quick search for repeated strings through fast search structure. Each 8 byte shingle in the Blk_{new} , shifted by one byte at a time, is hashed into 11 bits hash value that is used to search for a matching string in the dictionary. A bitwise comparison is also performed to make sure the two strings match exactly bit by bit. Once a match is found, a sequential search is performed to maximize the matching length. The search results are then encoded based on the encoding scheme described above.

String matching is done for every 8 byte shingle, i.e. subsequent shingles in a data block are shifted by just one byte at a time. Since our bus width is 8 byte, one set of delta compression engine cannot keep pace with the data transfer speed of the bus. To have compression hardware that has the wire speed, we construct 8 modules working in parallel, as shown in Figure 35. Each channel stores and

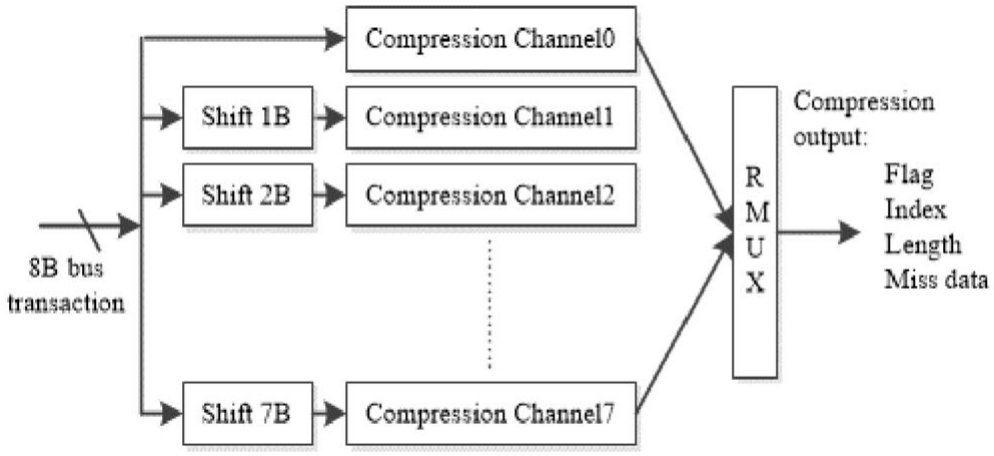

Figure 35. Parallel delta compression structure for every one byte shift shingles.

compresses one shingle. The 8 parallel modules work concurrently on 8 shingles that are one byte apart from each other.

4.5 Implementation and evaluation

4.5.1 Experimental setup

The three major hardware modules for similarity based dedupe as discussed above are built on Xilinx ML605 development board with V6-240T FPGA and our maximum clock speed is 250MHz. As a hardware coprocessor, it connects to the host through a PCI-e 2×4 bridge. A data deduplication software simulator [5] is running on the host PC with Intel(R) core(TM) 2 Duo CPU E7500 with 2.93GHz and 4GB DRAM. Figure 36 shows the block diagram of how the hardware modules are connected to the host system. For the purpose performance evaluation and comparison, we installed the standard dedupe software downloaded from [18]. By standard dedupe, we mean the dedupe function that perform data reduction only on identical data chunks. An open source software package [15] that does similarity based dedupe was also installed in order to evaluate the efficiency and effectiveness of our newly design hardware modules. Therefore, in the following discussions we will compare the three dedupe systems: the standard dedupe, the software module,

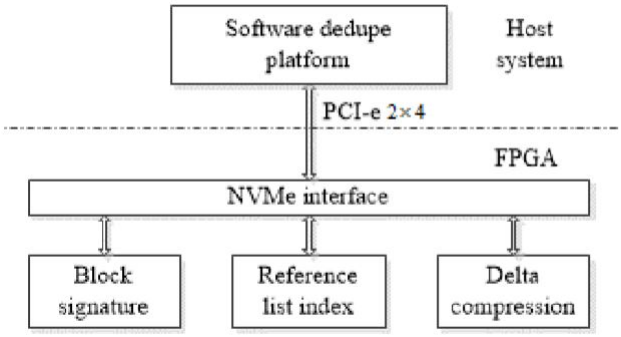

Figure 36. Experiment platform for hardware accelerating similarity based data deduplication.

and our hardware modules.

One important challenge for evaluating dedupe efficiency is the data sets used for the evaluation. Published I/O traces and standard benchmarks do not provide real data contents that are critical for assessing data redundancies. With the lack of standard benchmark data sets, we collected three real world data sets: 2.1GB Linux source code with different versions, 480 MB Japanese Census data, and 1GB Google book. These three data sets have different amounts of data duplications and compression ratios and represent different types of real world data. Our purpose here is to validate that our new hardware design can perform the dedupe function of existing software packages and to show the feasibility of using the hardware accelerator to carry out online dedupe for primary storage and storage caches.

4.5.2 Latency

Since data dedupe for primary storage and caches is on the critical path for production I/Os, minimizing dedupe latency is essential to storage I/O performance. We first evaluate latencies of the dedupe functions.

The first function for similarity based dedupe is fingerprint computation to derive sketches for data blocks. Our first experiment is to measure the times taken for computing the Rabin fingerprints and deriving a sketch for each data block.

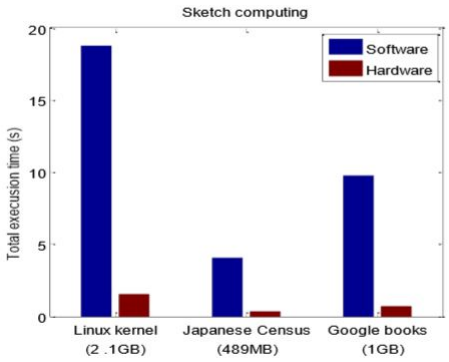

Figure 37. Sketch computing time on three datasets.

Figure 37 shows the measured latency for computing sketches of 4KB blocks using software implementation and our FPGA implementation, respectively. Three sets of bars are shown in this figure corresponding to the three data sets: Linux kernel, Japanese Census data, and Google book, respectively. It is shown in the figure that our hardware implementation running at 250 MHz clock, it reduces the total computation time substantially. At least an order of magnitude improvement has been observed from our experiments although the software version ran on the 2.93GHz Intel(R) core2 server with no other applications running. For example, for Linux kernel data set, using software fingerprint computation to derive sketches of all data blocks takes over 18.9 seconds, while our hardware implementation takes only 1.37 seconds. The average delay for computing a sketch of a 4KB block is about 2.5 us using the hardware module. But it takes over 30 us to do the same using the software module. For high performance storage such as SSD, this difference can have a significant impact on the production performance of disk I/Os. Not only does the hardware implementation speedup fingerprint computation greatly, but also offloads the computation to the accelerator allowing the server CPU to concentrate on application performance.

The second function is reference block search to find the best reference block for delta compressions. We measured the time it takes to search for a matched

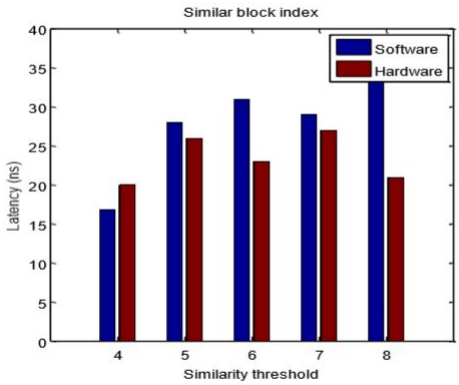

Figure 38. Average latencies of reference block search for different similarity thresholds.

reference block upon each new coming block. Figure 38 shows the measured results for different similarity threshold values. The similarity threshold is used to determine how many fingerprints in a sketch should match before a delta compression is performed. The lower, the threshold value is, the higher the chance to find similar blocks. However, it also increases the chance of false positive, i.e. two blocks are considered similar based on a few fingerprints match but they are not delta compressible. Higher threshold value, on the other hand, give a better chance that two blocks are delta compressible because they have more matched fingerprints in their respectively sketches. But some compressible blocks may be missed if the threshold value is too high.

From Figure 38 we can see that our hardware implementation of reference block search takes less time than its software counterpart. However, the latency reduction is not as substantial as the fingerprint computation part. From our experiments, we observed two reasons for this. First of all, our hardware design on this part is still preliminary and there are rooms for optimization given more time. The software implementation, on the other hands, is pretty mature with many built in optimizations. Secondly, the latency time is on the order of the 10s of nanoseconds. There is not much space for hardware to do much better since

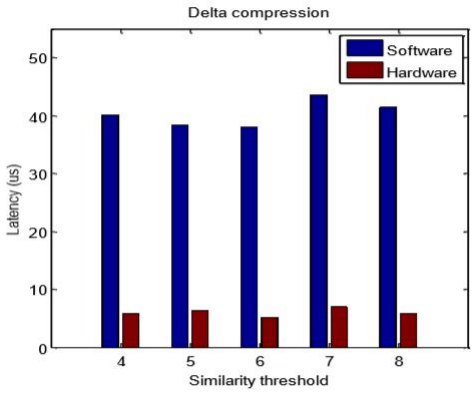

Figure 39. Delta compression time comparison between hardware and software with different thresholds.

the clock speed of the FPGA on which we implemented the circuit is 100 Mhz while the CPU used to run the software module is a multicore with 2.93 GHz clock speed. Fortunately, this part of the processing should drag down the I/O performance significantly since it is in nanosecond range, a negligible time for disk I/O operations.

The third function is delta compression. We measured the delta compression time of each 4KB associated block against a 4KB reference block using our hardware compressor. We also measured the same compression time using the software delta compressor MiniLZO [19]. The performance comparison is shown in Figure 39. The compression time varies depending on the chosen similarity threshold values. It is clear from this figure that the hardware compressor shows 6 to 8 times faster performance than its software counterpart. This substantial speedup on delta compression is very important for dedupe on primary storage and storage caches. It is noticed that the compression time of the hardware module is about 5 us while the software module takes around 40 us that could take a significant part of an I/O response time.

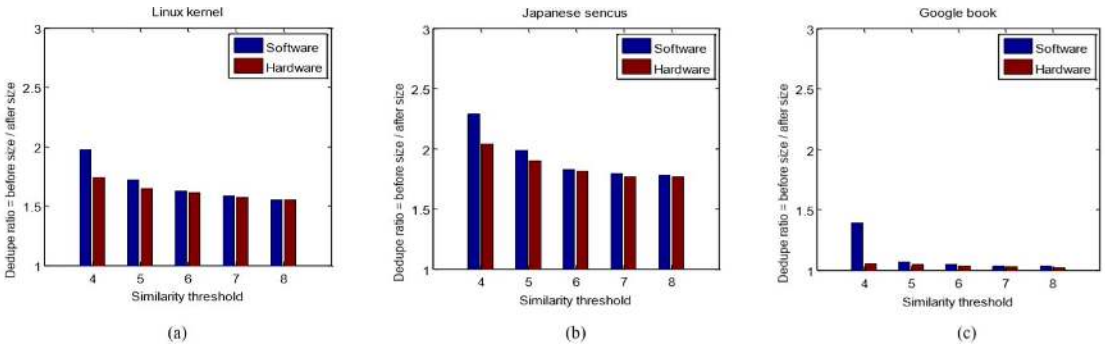

Figure 40. Similarity based data reduction comparison between hardware and software for three datasets with different similarity thresholds.

4.5.3 Data Reduction Ratio

In order to validate dedupe capability of our hardware design, we carried out experiments to measure the data reduction ratio of the hardware dedupe system. We compare this ratio with mature software dedupe systems. The purpose is to make sure the high speed hardware can achieve the expected data reductions. Figure 40 shows the data reduction ratios of similarity based dedupe for both software package and the hardware implementation for data set Linux kernels. It can be seen in this figure that the data reduction ratios of the two systems are comparable for all similarity threshold values considered. We noticed that for lower similarity threshold values such as 4, software package does a little bit better job than the hardware implementation. Our analysis of the hardware design and the software package suggests the following reasons for this. First of all, with the software compressor, data compression can be done both inner block and inter blocks between the reference block and the associated block. In the hardware implementation, on the other hands, only inter block compression is performed. Further improvement on the hardware design is possible. Secondly, for smaller value threshold, software can work much harder with more iterations to find string matches within and between blocks. The hardware implementation will perform just one pass and it may miss some substring matches.

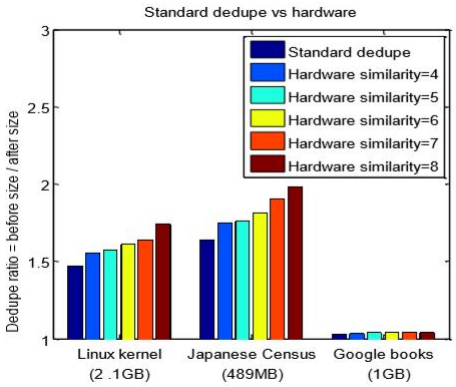

Figure 41. Comparison between standard dedupe and hardware dedupe for three datasets.

Figure 40(b) and Figure 40(c) show the data reduction ratios of data sets Japanese census data and Google book, respectively. Similar observations are obtained to that of Figure 40(a). The data reduction ratios of the software package and hardware implementation are fairly close but there are small differences between the two for smaller threshold values.

We have also carried out experiments to compare the data reduction ratios of standard dedupe and similarity based dedupe. The measured results are shown in Figure 41. From this figure, one can see that the similarity based dedupe shows better data reduction than standard dedupe because of the existence of similar data blocks. From our experiments, we observed about 30% better data reduction of similarity based dedupe than standard dedupe. We believe such improvement should be much bigger in real world environments and in production systems. More similarity exists in real world data such as databases, big data, large files, sensor data, and data being processed by servers. As a result, similarity based data dedupe should perform much better in terms of data reduction.

4.6 Conclusion and future works

We have proposed a hardware accelerator to speed up similarity based deduplication for primary storages or storage caches. The new hardware implemen-

tations have achieved compression ratios that are comparable to mature software implementation, while providing 6 to 8 times higher throughputs than software implementation. A working prototype has been built using Virtex-6 FPGA under Xilinx environment. Extensive experiments have been carried out by connecting the prototype to a host server through a PCI-e interface to test the validity and performance of the hardware implementation. Our experiments have demonstrated the performance in terms of latency and data reduction ratio of the hardware design. The architecture is readily applicable in primary data storages and caching in hybrid data storage systems.

Future optimization can still be achieved by improving hardware delta compression ratio. There is a room for improvement for reference block index to eliminate or minimize the usage of FPGA RAM resources for the hash table. We currently working on integrating all the hardware modules together to form a complete dedupe system for primary storages and storage caches.

List of References

[1] Kiran Srinivasan, Tim Bisson, Garth Goodson, and Kaladhar Voruganti, NetApp, Inc. *iDedup: Latency-aware, Inline Data Deduplication for Primary Storage*, USENIX conference on Hot Topics in Storage and File Systems, 2012

[2] Philip Shilane, Grant Wallace, Mark Huang, and Windsor Hsu, *Delta Compressed and Deduplicated Storage Using StreamInformed Locality*, in HotStorage'12 Proceedings of the 4th USENIX conference on Hot Topics in Storage and File Systems, 2012.

[3] Jingxin Feng, Jiri Schindler, *A Deduplication Study for Host-side Caches with Dynamic Workloads in Virtualized Data Center Environments*, Mass Storage Systems and Technologies (MSST),, p. 1 6, 6-10 May 2013.

[4] Jian Liu, Yunpeng Chai, Xiao Qin, Yuan Xiao, *PLC-cache: Endurable SSD cache for deduplication-based primary storage*, Mass Storage Systems and Technologies (MSST),, p. 1 12, 2014.

[5] Min Fu, Dan Feng, Yu Hua, Xubin He, Zuoning Chen, Wen Xia, Fangting Huang, and Qing Liu, *Accelerating Restore and Garbage Collection in*

Deduplication-based Backup Systems via Exploiting Historical Information, in USENIX ATC'14.

[6] James J. Hunt, Kiem-Phong Vo , Walter F. Tichy, *Delta algorithms: an Emprical Analysis*, ACM Transactions on Software Engineering and Methodology, 1998. pp. 192-214.

[7] Philip Shilane, Mark Huang, Grant Wallace, and Windsor Hsu, *WAN Optimized Replication of Backup Datasets Using StreamInformed Delta* , in ACM Transactions on Storage (TOS) , p. Volume 8 Issue 4, November 2012.

[8] Torsten Suel, Nasir Memon, *Algorithms for Delta Compression and Remote File Synchronization*, in Work supported by NSF CAREER Award NSF CCR-0093400 and by Intel Corporation., 2002..

[9] E.Jebamalar Leavline, D.Asir Antony Gnana Singh, *Hardware Implementation of LZMA Data Compression Algorithm*, International Journal of Applied Information Systems (IJAIS) ISSN : 2249-0868, p. Volume 5 No.4, March 2013.

[10] W. Cui, *New LZW Data Compression Algorithm and Its FPGA Implementation*, in processing of 26th Picture Coding Symposium , 2007.

[11] Paul R. Wilson, Scott F. Kaplan, and Yannis Smaragdakis, *The Case for Compressed Caching in Virtual Memory Systems*, Proceedings of the USENIX Annual Technical Conference, 1999.

[12] M. Rabin, *Fingerprinting by random polynomials*, Aiken Computation Laboratory, Univ., 1981.

[13] A. Z. Broder, *Some applications of Rabins fingerprinting method*, in Communications, Security, and Computer Science, 1993, p.143152.

[14] A. Z. Broder, M. Charikar, A. M. Frieze and M. Mitzenmacher, *Min-Wise Independent Permutations*, in Computer and System Sciences, pp. 21-29, 1998.

[15] Q. Yang, and J. Ren, *I-CASH: Intelligently coupled array of ssd and hdd*, in High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on, IEEE, 2011, pp. 278-289.

[16] Lukas Kekely, Martin Zadnk, Jir Matousek, Jan Korenek, *Fast lookup for dynamic packet filtering in FPGA*, in Design and Diagnostics of Electronic Circuits and Systems, 17th International Symposium on, 23-25 April 2014 pp. 219 - 222.

[17] Xilinx, *http://www.xilinx.com/products.html*

[18] Destor Github: *https://github.com/fomy/destor*

[19] M. Franz, J. Oberhumer, *mini subset of the LZO real-time data compression library*, in Github:
https://github.com/dfelinto/blendergit/tree/master/extern/lzo/minilzo