2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines

HODS: Hardware Object Deserialization inside SSD Storage

Dongyang Li[1,3], Fei Wu*[,2], Yang Weng[2,3], Qing Yang[1,3], Changsheng Xie[2]
[1]*Dept. of Electrical, Computer and Biomedical Engineering, University of Rhode Island, Kingston, RI, US*
[2]*Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, WuHan, China*
[3]*Shenzhen Dapu Microelectronics Co., Ltd., Shenzhen, China*
*Email: lidongyang@ele.uri.edu, *Corresponding author: Fei Wu, wufei@mail.hust.edu.cn*

***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.**

***Keywords*-– Processing In Storage (PIS); Object Deserialization; NVM-e SSD;**

I. 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

2576-2621/18/$31.00 ©2018 IEEE
DOI 10.1109/FCCM.2018.00033

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.

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

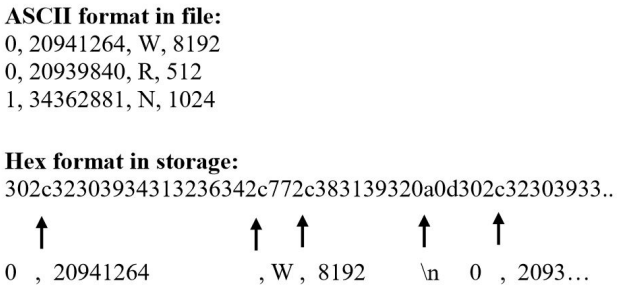

Figure 1. An example ASCII file.

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

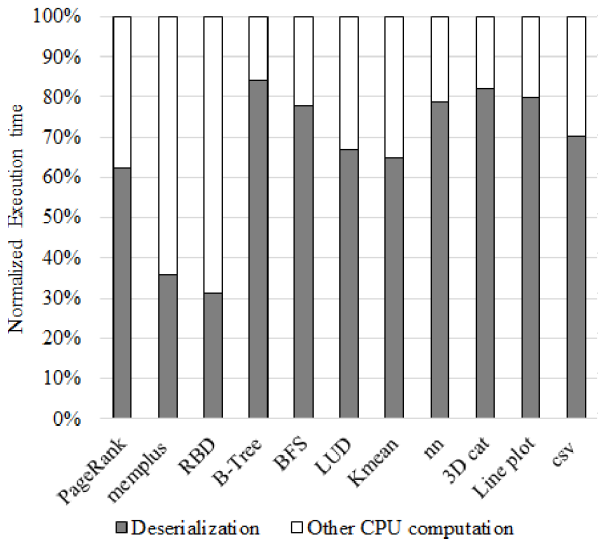

Figure 2. Fractions of object deserialization time over total running time of applications.

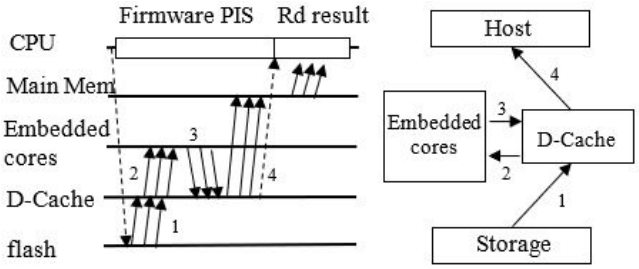

Figure 3. General data flow of existing PIS functions inside an SSD device.

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 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. 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 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.

III. 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

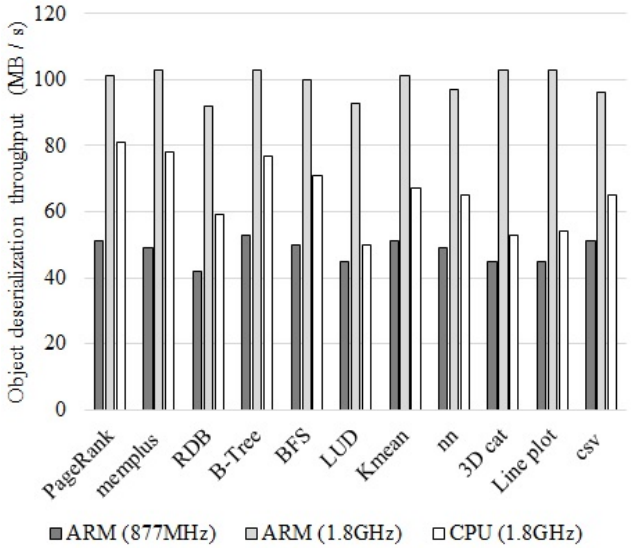

Figure 4. Comparison of object deserialization throughput between embedded ARM inside SSD and host CPU.

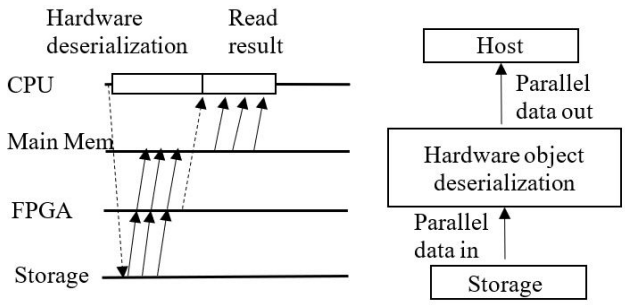

Figure 5. FPGA based object deserialization model.

transfer on the bus. In the following paragraphs, we will describe system architecture, hardware object deserialization module, and host driver program in detail.

A. 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.

B. 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

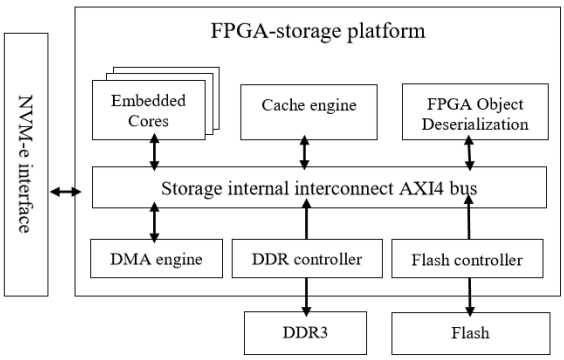

Figure 6. HODS architecture of FPGA based NVM-e storage.

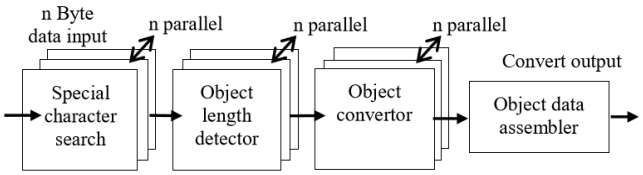

Figure 7. Hardware object deserialization diagram.

comma along with n bytes parallel data stream. The special characters' location information will be passed 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, 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 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.

C. 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

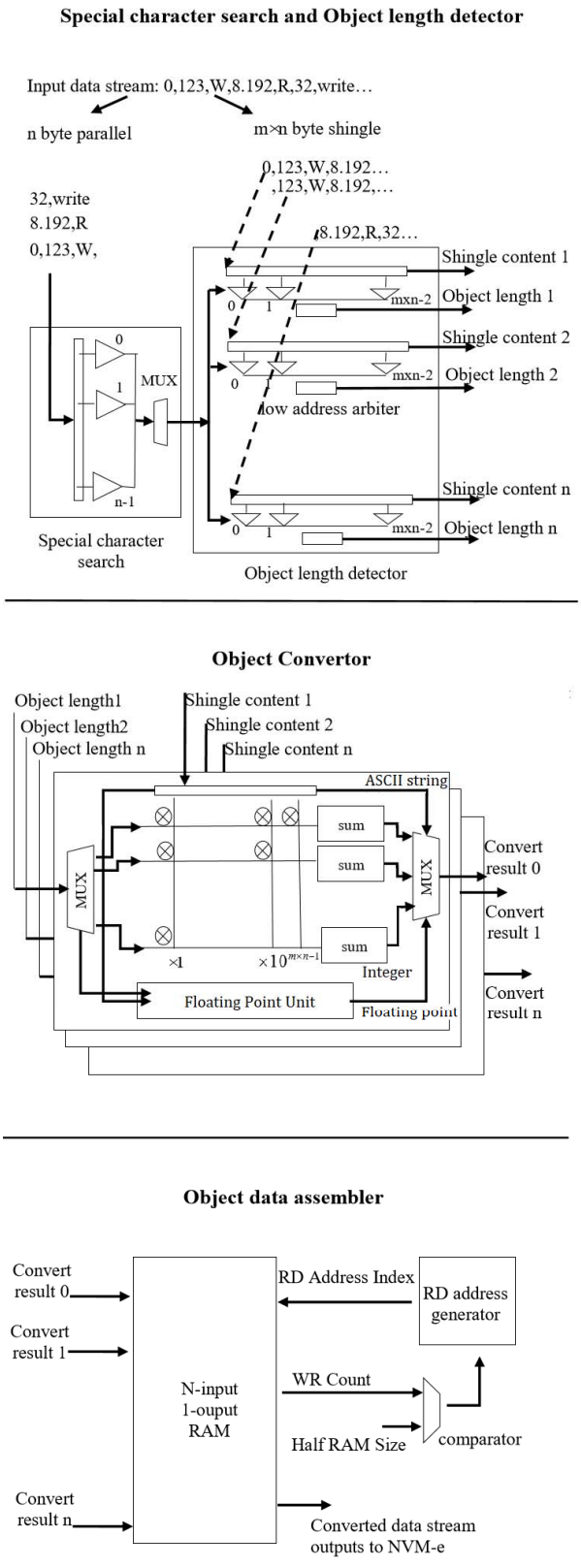

Figure 8. Pipeline stages of hardware object deserialization

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.

IV. 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.

A. 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 storage 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.

B. 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.

V. 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.

A. 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 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

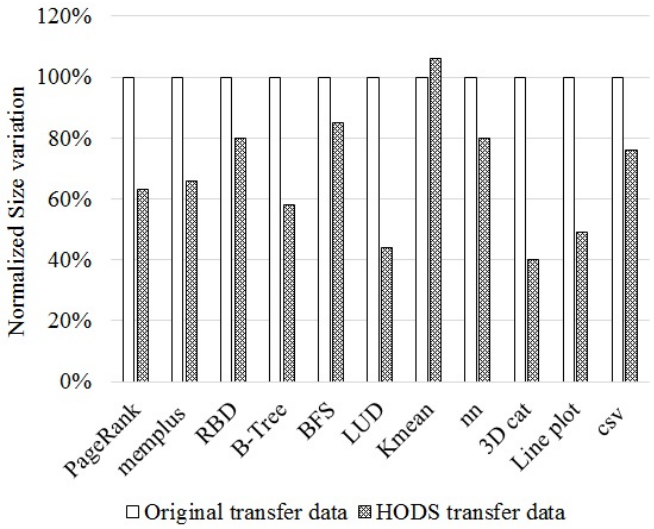

Figure 9. Normalized size variation after hardware deserialization.

SSD with HODS would perform the same as a 1 million IOPS SSD without HODS.

B. 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 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.

C. 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.

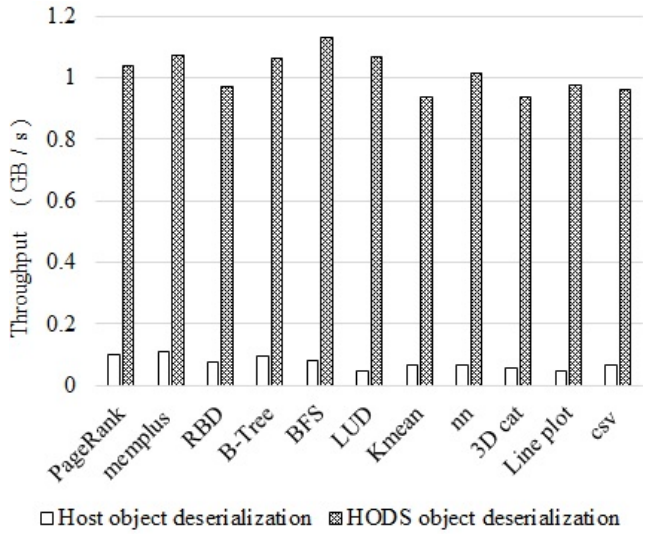

Figure 10. Throughput comparison between hardware object deserialization and host software solution.

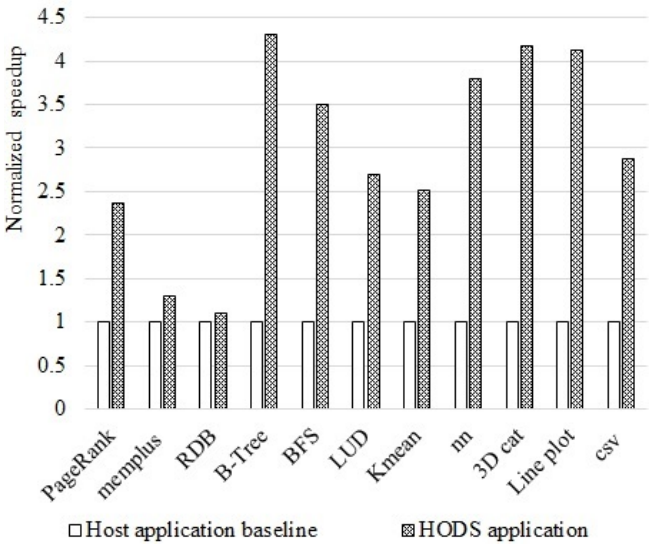

Figure 11. Normalized hardware deserialization speedup.

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 parallel 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.

VI. 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.

VII. Acknowledgements

This research is supported in part by the NSF grants CCF-1439011 and CCF-1421823. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF. It is also partly supported by a research contract between URI and Shenzhen Dapu Microelectronics Co., Ltd, Shenzhen Peacock Plan (KQTD2015091716453118), the National Natural Science Foundation of China under Grant No.U1709220, No.61472152, No.61572209, Wuhan Science and technology Project No.2017010201010108, Shenzhen basic research project No.ICYJ20170307160135308, the Fundamental Research Funds for the Central Universities No.2016YXMS019, the 111 Project (No.B07038), and the Key Laboratory of Data Storage System, Ministry of Education.

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. F. Hu, R. J. Allan and K. C. F. Maguire *Comparing the performance of JAVA with Fortran and C for numerical computing*. http://yifanhu.net/SOFTWARE/JASPA/index.html

[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