FIDR: A Scalable Storage System for Fine-Grain Inline Data Reduction with Efficient Memory Handling

Mohammadamin Ajdari
Department of Computer Science and Engineering
POSTECH
majdari@postech.ac.kr

Wonsik Lee
Department of Electrical and Computer Engineering
Seoul National University
wonsik.lee@snu.ac.kr

Pyeongsu Park
Department of Electrical and Computer Engineering
Seoul National University
pyeongsu@snu.ac.kr

Joonsung Kim
Department of Electrical and Computer Engineering
Seoul National University
joonsung90@snu.ac.kr

Jangwoo Kim*
Department of Electrical and Computer Engineering
Seoul National University
jangwoo@snu.ac.kr

ABSTRACT

Storage systems play a critical role in modern servers which run highly data-intensive applications. To satisfy the high performance and capacity demands of such applications, storage systems now deploy an array of fast SSDs per server. To reduce the storage cost of employing many SSDs per server, storage systems actively perform inline data reduction (e.g., data deduplication, compression). Existing inline data reduction studies can achieve high performance and scalability by offloading computation-intensive data-reduction operations to dedicated hardware accelerators. However, such existing studies suffer from limited workload support and scalability. For example, they reduce only large data blocks, which incur many IO requests, leading to low data reduction rates, and their offloading overlooks memory-intensive operations, leading to the unoptimal scalability.

In this paper, we propose *FIDR*, a highly scalable storage system to enable the inline data reduction of fine-grain data. We first identify key limitations of existing studies, and then set our scaling storage server design which effectively resolves the limitations by employing an optimal offloading mechanism. The key ideas of FIDR are to achieve high applicability by enabling fine-grain data reduction and high scalability by distributing data-reduction operations to optimal devices (e.g., host processor, accelerator, network interface card). The proposed offloading mechanism considers computation, memory capacity, and memory bandwidth requirements altogether. For evaluation, we implement an example FIDR system prototype using FPGAs. Our prototype system outperforms a current state-of-the-art data reduction system up to 3.3 times by significantly reducing both computation and memory resource requirements.

*Corresponding author.

MICRO-52, October 12–16, 2019, Columbus, OH, USA

ACM ISBN 978-1-4503-6938-1/19/10…$15.00
https://doi.org/10.1145/3352460.3358303

CCS CONCEPTS

• **Information systems** → **Information storage systems**; • **Computer systems organization** → **Architectures**; • **Hardware** → **Communication hardware, interfaces and storage**.

KEYWORDS

deduplication, compression, FPGA, SSD array, small chunk, table management, memory management

ACM Reference Format:
Mohammadamin Ajdari, Wonsik Lee, Pyeongsu Park, Joonsung Kim, and Jangwoo Kim. 2019. FIDR: A Scalable Storage System for Fine-Grain Inline Data Reduction with Efficient Memory Handling. In *The 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52), October 12–16, 2019, Columbus, OH, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3352460.3358303

1 INTRODUCTION

With massive data generation in recent years, data-intensive applications such as database and machine learning have gained significant popularity in datacenters [18]. Along with different components of datacenters, storage servers are evolving to be denser (i.e., higher capacity per node) and faster (i.e., higher throughput and lower latency). Recent commercialization of extremely high-throughput high-capacity IO devices have paved the way for storage servers with *multi terabit-per-second (Tbps)-scale* performance and *petabyte (PB)-scale* capacity. For example, a 1-PB capacity server as small as 1 rack unit is shown to be possible with an array of 32-TB ruler SSDs [42]. As another example, a single node flash appliance of SmartIOPS can be considered [35]. It provides half-PB capacity, along with 6.4 Tbps performance using an array of 6.8+ GB/s SSDs [36].

To minimize the cost of SSD arrays, commercial storage servers typically apply inline data reduction (i.e., deduplication and compression) to the data before they are written to the SSDs [1, 12, 31]. Data deduplication and compression have been shown to remove the data redundancies in the real systems by over 50% for database datasets and over 80% for virtual desktop infrastructures [31]. Such significant reduction of data footprint not only improves an SSD

lifetime, which is limited by the number of writes to its flash cells, but also reduces the initial cost per GB of an SSD array.

Existing work has shown that completely relying on the CPUs for the data reduction is not scalable [2, 5, 9, 16]. Therefore, they propose to offload compute-intensive tasks from CPUs to a hardware (HW) accelerator. For example, CIDR [5] uses FPGA acceleration of hash computation step in deduplication and the compression to achieve over 20 GB/s data reduction throughput with two FPGAs.

Unfortunately, existing HW-accelerated systems have very limited applicability (e.g., supporting only workloads with large block accesses) and very limited scalability (due to overlooking memory management overheads). In this paper, we first identify the limitations of existing approaches by profiling CIDR [5], a state-of-the-art HW-based data reduction system. To saturate 1 Tbps PCIe bandwidth (BW) of a CPU socket[1], CIDR demands *over 317 GB/s of host memory bandwidth*, which is far more than the theoretical BW (170 GB/s) on a high-end socket [7]. We also identify a new CPU bottleneck which comes from inefficient memory management and heavy-weight accelerator scheduling rather than straightforward computations. For example, we show that to provide around 1 Tbps throughput, a CIDR-like system *demands over 67 Xeon cores*, which is significantly more than the number of cores in a high-end CPU socket (e.g., 22 cores [20]).

To guide the design for a scalable data reduction system, we perform detailed profiling of the state-of-the-art data reduction system and make four new observations.

Observation #1. Data reduction has operations with different memory-bandwidth and -capacity requirements. Our characterization shows that some data reduction operations (e.g., accelerator scheduling and data buffering) consume up to 85.1% of the host memory bandwidth while using less than 1 GB of the memory capacity. In contrast, some operations such as data reduction metadata caching consume up to 25% of host memory bandwidth while demanding 100s of GB of DRAM capacity.

Observation #2. Up to 85.1% of data movements through host memory are for extracting short metadata and temporary data buffering. Complex processing (e.g., compression) of the client's data only happens in HW accelerators. This means that many host-side data movements become unnecessary if some simple buffering and processing are offloaded to accelerators.

Observation #3. Integration of hashing step (of deduplication) and compression in an accelerator requires heavy-weight host-side accelerator scheduling. CIDR aims to reduce the hardware costs by integrating hashing and compression cores in the same accelerator. As deduplication and compression are performed in series, the hashing cores first compute hash values for all chunks and then the compression cores should operate on the data that have not been eliminated by deduplication (i.e., non-duplicate data). This requires unscalable host-accelerator interactions or expensive buffering in an accelerator. CIDR addresses this problem by using special host-side software to predict non-duplicate data blocks in advance and efficiently scheduling the batch for the accelerator. However, we observe that such predictor is a new source of CPU- and memory-bottleneck.

Observation #4. Neither host CPU/memory-only approaches nor accelerator-only approaches can provide scalable data reduction table caching. Caching of data reduction metadata requires 100s of GB of memory to scale to PB-scale system. While host memory provides this capacity easily, simply using CPU to index such cache makes a bottleneck. Our detailed profiling shows that most CPU-intensive tasks for metadata cache management are accessing small data structures (e.g., less than 3 GB tree indexing 500 GB metadata content cache). In contrast, only 6.3% of CPU utilization is used for accessing the cached metadata contents (with 100s of GB size).

To the best of our knowledge, our work is the first to identify the problems in the accelerator management, accelerator scheduling and the table caching for realizing a scalable data-reduction system. Based on the above observations, we propose *FIDR*, a scalable data reduction system for **F**ine-grain **I**nline **D**ata **R**eduction using three main ideas: (a) hash offloading to NIC, (b) in-NIC buffering + PCIe peer-to-peer transfers and (c) hybrid CPU/memory/FPGA (metadata) table caching. First, by offloading hashing from an accelerator to a NIC, the NIC can detect non-duplicate chunks early. This capability enables FIDR to remove the CPU- and memory-intensive *predictor* (that exists in CIDR). , in addition to enabling the transfer of only the unique chunks (instead of the total data chunks) to other devices which minimizes the PCIe bandwidth requirements. Second, by offloading buffering of client's requests from the host memory to the NIC and taking advantage of PCIe direct peer-to-peer (P2P) feature, FIDR completely bypasses the host-memory and directly transfers the required data across devices (NIC→Compression accelerator→ data SSDs). This idea makes scalable accelerator management possible in a data reduction system. Finally, we propose a hybrid host CPU/memory/FPGA acceleration to ensure scalable data reduction table caching. We use FPGAs for CPU-intensive memory management tasks (i.e., tree indexing and table cache replacement) and host memory/CPU for accessing the cached table content.

We made an example prototype of FIDR on multiple FPGAs to show its performance and cost-effectiveness for a large-scale system. Our evaluation shows that FIDR provides a significant speedup of up to 3.3× compared to the state-of-the-art HW-based data reduction system. FIDR also successfully reduces the memory bandwidth utilization and CPU utilization by up to 79.1% and 68%, respectively. Our performance results and cost analysis prove that FIDR is a cost-effective solution for a PB-scale storage server.

In short, we make the following contributions:

- **Critical Problem Identification**. To the best of our knowledge, our work is the first to identify the problems in accelerator management, accelerator scheduling and table caching for realizing a scalable data-reduction system.
- **Guidelines for Building a Scalable Storage System**. We characterize the state-of-the-art HW-based storage system in detail and provide four insightful observations to resolve our identified scalability limitations.
- **Effective System Architecture and Implementation**. We propose a carefully-designed, end-to-end data reduction architecture using (a) hash offloading to NIC, (b) in-NIC buffering + PCIe direct P2P, (c) and hybrid host CPU/memory/FPGA acceleration for table caching. We prototype our system to validate its high performance and cost-effectiveness.

The rest of the paper is organized as follows. Section 2 explains the basics of a data reduction system and our baseline. Section 3–4

[1]Maximum PCIe BW supported in a CPU socket is 1 Tbps in some high-end systems

motivate our work and provide our observations. Section 5–6 show our proposed system architecture and implementation. Section 7–9 explain our evaluation results, related work and discussion. Finally, Section 10 concludes the paper.

2 BACKGROUND

2.1 Major Data Reduction Components

To improve the cost-effectiveness of a storage server, two data reduction techniques are commonly used: *deduplication* to eliminate the data redundancy across different data blocks (inter-block redundancy) and the *compression* to eliminate data redundancy inside each block (intra-block redundancy). Data reduction requires four major components (mostly for deduplication). In the following, we briefly explain each component.

2.1.1 Data chunking. The first component splits the client's data into smaller data blocks (called *chunks*). Chunking has two parameters (fixed/variable chunking and chunking granularity). To support various workloads, many commercial systems use fixed sized small chunking [37] or use variable sized chunking [31]. Due to high computational overheads of variable sized chunking, we use fixed sized small (4-KB) chunking in this paper.

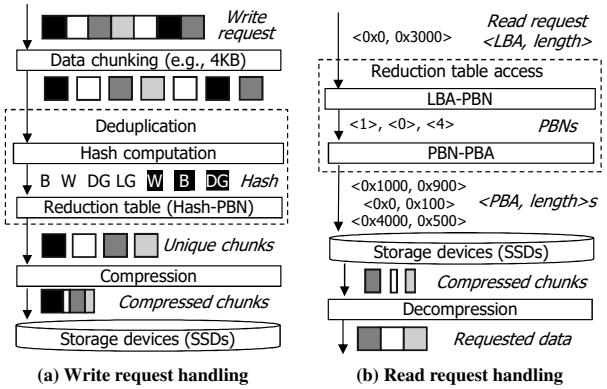
2.1.2 Chunk hashing. Computing the *hash* value of a chunk and using it as the signature of a data chunk is necessary for deduplication. Comparing the signatures (instead of the bit-by-bit comparison of the raw data) enables fast detection of duplicate chunks. Functionally correct deduplication requires different data to have different signatures (i.e., no hash collisions). To ensure such property, similar to existing work [5, 14, 43], we use strong hash functions (e.g., SHA2) with no practical collisions in petabytes of data.

2.1.3 Hash-PBN table. This is a key-value store of metadata containing the hash value of each stored chunk (as the key) and its chunk physical block number (PBN) (as the value). By checking the existence of a given chunk hash in the Hash-PBN table, the storage server determines if a chunk is duplicate or non-duplicate (also known as *unique*).

One common implementation of the Hash-PBN table is a bucket-based table, containing many pairs of (key, value) in each bucket. Upon receiving a client's write request, the server uses a simple modular function to calculate the bucket index that may hold the chunk hash. Then, the corresponding bucket is scanned to find the respective hash value. If the chunk hash is not found (as it is unique), the pair of <new chunk hash, newly allocated PBN> is inserted into the respective bucket.

Considering that each entry of the Hash-PBN table is 38 bytes (32 bytes for hash, 6 bytes for PBN), with 4-KB chunking and 1-PB unique chunk storage, the Hash-PBN table is 9.5 TB large. Such size is much larger than the available DRAM in a typical system. Therefore, the server caches only part of the table in DRAM and keeps the full table in separate SSDs (we call *Table SSDs*).

2.1.4 LBA-PBA table. This is another key-value store of metadata that maps a client's logical block address (LBA) for each chunk to its respective physical block address (PBA). Because chunks have variable sizes after being compressed, we use two level mapping of LBA to PBA. For efficient data storage in an SSD, the server



(a) Write request handling **(b) Read request handling**

Figure 1: Overview of data reduction flow in a storage server

usually makes a large container of compressed chunks and stores them as a single large block. For this case, the LBA-PBA table internally has LBA-PBN mapping (an array whose index is LBA and its value is the PBN in a container) and PBN-PBA mapping (an array whose index is PBN and its value is <offset address in the container, compressed chunk size>). Using the PBN in a container and offset in the container, the server simply calculates the PBA.

The size of this table is multi-TBs in a PB-scale storage. Each entry in this table has a small size (6 bytes for PBN, 2 bytes for the offset, 2 bytes for the compressed size). As workloads usually exhibit some address locality, a small DRAM-based cache for the LBA-PBA table is enough for a high performance system. Therefore, we focus on Hash-PBN caching effects rather than LBA-PBA caching.

2.2 Basic Operation Flow of Data Reduction

Figure 1 shows the flow of the data reduction with deduplication and compression in series. Upon receiving a client's write request, the storage server first splits it into data chunks (Figure 1a). Next, the server applies deduplication to remove the data chunks that are already stored in the SSDs (i.e., the chunks are *duplicate*). The server computes the hash for each chunk and looks up those hash values in the Hash-PBN table. Then, if the server detects a chunk as duplicate, it only updates the LBA-PBA table. If the chunk is unique, the server compresses the chunk content and stores them in the storage devices (i.e., SSDs). The server also updates the Hash-PBN and LBA-PBA tables to include the metadata of the new unique chunks.

Figure 1b shows the flow for handling read requests. The storage server receives a client's request to read data at a specific LBA with a specific length. The server finds the PBA of the requested chunks by looking up the LBA in the LBA-PBA table. Then it reads the compressed data chunks at the PBA of the SSDs, decompresses them and sends them to the client.

2.3 Baseline HW-based Data Reduction

Some existing work proposes to use HW-based data reduction systems to accelerate compute-intensive operations. Among them, we modify a recent HW-based data reduction (known as CIDR [5]) to serve as our baseline architecture. CIDR uses an array of FPGAs to accelerate hashing, compression, and decompression. As data

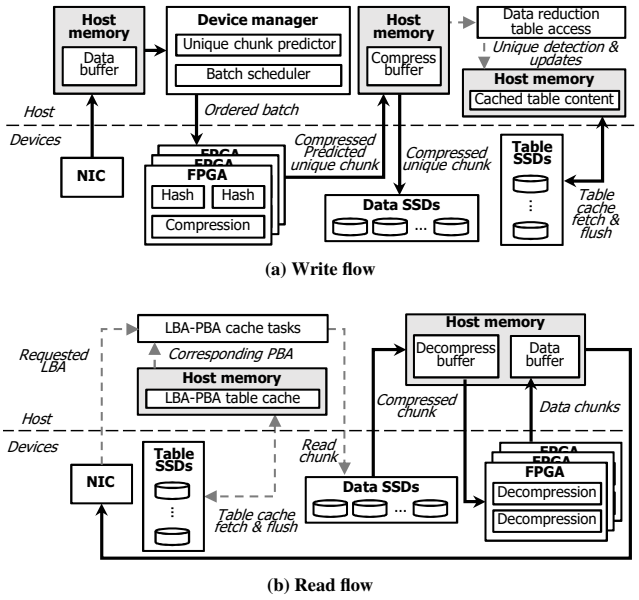


Figure 2: Simplified operation of baseline with HW accelerated hash computation, compression and decompression.

reduction metadata tables are on the host, it requires multiple interactions with FPGAs to receive the hash values, detect unique chunks and informs the FPGAs for compressing only unique chunks. Such acceleration requires keeping large batches of chunks for some time on the accelerator expensive memory, while the host detects unique chunks and informs the accelerator for the compression of selected unique chunks. An alternative is to use two separate data transfers to the FPGAs (one for hashing, one for compression). Both of these approaches limit scalability. Therefore, CIDR used a special software-based module that predicts unique chunks before deduplication and enables the one-time transfer of batches to FPGAs. In this way, while the FPGA hash units hash all the chunks, the compression units can simultaneously compress predicted unique chunks. Due to CIDR low target capacity (i.e., tens of TBs) and large sized chunks, the authors assumed that the data reduction tables fit in memory [5]. With our PB-scale target capacity and small (4-KB) chunking, data reduction tables require multi-TB memory capacity. Therefore, we assumed that the data reduction tables are in dedicated SSDs (i.e., *Table SSDs*) and a software module manages caching of the tables in host memory (the main reasons that we modified CIDR).

The write request handling flow in our baseline is as follows (Figure 2a). First, the server's NICs receive the client's data and buffer them in host memory. Then the unique chunk predictor reads the data buffer and predicts which chunks are unique. Next, the batch scheduler groups the chunks based on the prediction result and sends them to the FPGA array. After the FPGAs perform hashing on all received chunks and compress predicted unique chunks, the FPGAs transfer the compressed unique chunks, along with the hash values of all chunks to the host memory. At this point, software-based table management checks the validity of the prediction. It looks up the hash values in the Hash-PBN table cache in host memory. If the

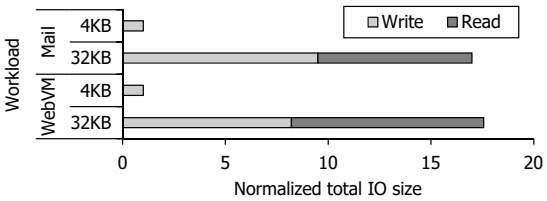


Figure 3: Significant increase of IO requests in large chunking due to read-modify-writes and deduplication degradation.

target bucket is unavailable in the host memory, the server would fetch the bucket from the table SSDs. Finally, the baseline writes the compressed validated unique chunks to the *data SSDs*.

Read request handling has a much simpler flow (Figure 2b). Upon receiving a read request from a client, the server's NICs send the LBA of the request to the host. The host software looks up the LBA in the LBA-PBA table and finds the respective PBA and the size of the compressed chunk. It then reads the compressed chunk from the data SSDs to the host memory. Then the data are forwarded from host memory to the FPGA array for decompression. After the FPGAs write the decompressed data to host memory, the NIC reads the data and sends them to the client.

3 MOTIVATION

In this section, we first motivate the use of small chunking. We then profile the baseline hardware-based data reduction and identify new bottlenecks on memory and CPU at scale.

3.1 Limitation of Large Chunking

Large data chunking (e.g., 32 KB chunking used in CIDR) reduces the data reduction table size and its access frequency; however, the large chunking is not suitable for every workload. We observe that large chunking suffers from frequent read-modify-writes which cause up to *17.5x increase in IO overhead* (Figure 3). We simulated the deduplication with large chunking for the write requests of two real traces (mail server and webVM [39]). We assumed a 4 MB request buffer before deduplication. Due to the existence of many 4-KB accesses and some randomness in LBA accesses, a deduplication system with 32-KB chunking cannot easily make a 32-KB chunk from the client's requests. Thus the deduplication module fetches the missing 4-KB blocks from the SSDs, forms a 32-KB chunk, applies deduplication and if the 32-KB chunk is unique, it writes the chunk back to the storage. Such operation adds significant additional reads and writes. Furthermore, large chunking deduplication degrades duplicate detection capability and adds more writes. Overall, the large chunking can make a bottleneck on the SSD bandwidth. Therefore, in this paper, we used small (4-KB) chunking.

3.2 Limitation of Existing HW-based Approach

In this section, we profile naively extended CIDR, a recent HW-based data reduction system [5]. Our extension enables small chunking and large storage capacity. We observe that such systems only focus on offloading straightforward compute-intensive operations to hardware accelerators and miss new memory-related bottlenecks. We show our profiling results on two example workloads: one workload with

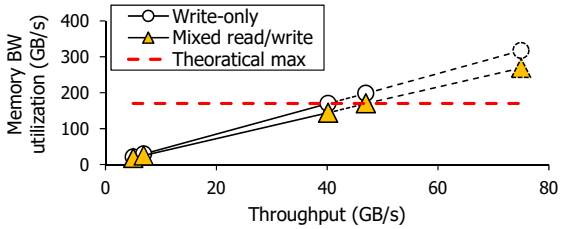
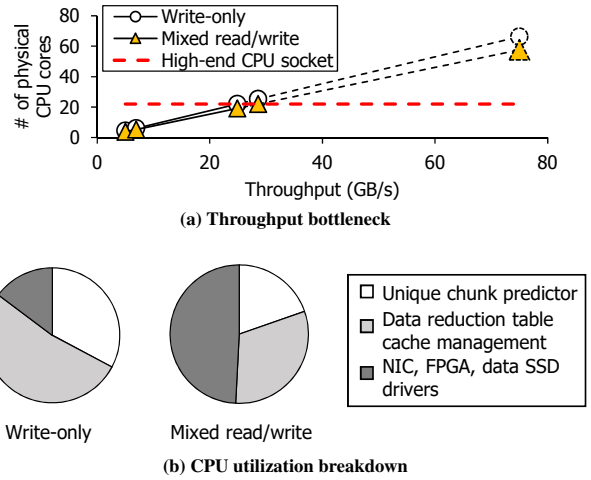


Figure 4: Memory bandwidth bottleneck in the baseline HW-based data reduction system.



(a) Throughput bottleneck

(b) CPU utilization breakdown

Figure 5: CPU bottleneck in the baseline data reduction system.

only write requests (i.e., *Write-only*) in which deduplication and compression ratios are both set to 50%, and another workload with half read and half writes (i.e., *Mixed read/write*) in which the write requests have same deduplication and compression ratio.

Considering that high-end CPUs (e.g., [7]) provide 128 GB/s (= 1 Tbps) theoretical PCIe IO bandwidth, we conservatively set our target throughput to 75 GB/s per socket. Setting the throughput to 60% of the theoretical 128 GB/s is to conservatively consider various IO overheads such as DMA management [46]. Because each socket has independent CPU cores, independent memory, and IO buses, for simplicity, our measurements for memory bandwidth and CPU utilization are also on the per-socket basis.

3.2.1 Memory Bandwidth Bottleneck. To achieve 75 GB/s, the HW-based data reduction baseline requires *1.9x more memory bandwidth* than a high-end socket can theoretically provide (Figure 4). We measured the memory bandwidth of the baseline at two points with different data reduction throughputs (5 GB/s and 6.9 GB/s) and linearly projected the trend. Our measurements indicate that at 75 GB/s, the baseline requires 317 GB/s memory bandwidth for the *Write-only* and 269 GB/s for the *Mixed read/write*. Considering that a high-end socket has maximum 8 memory channels, the theoretical bandwidth that a socket can provide is only 170 GB/s [7]. With conservative estimation, this limits the baseline throughput to 40-47 GB/s, up to 1.9x lower than our target throughput.

3.2.2 CPU Bottleneck. The HW-based baseline also requires an infeasible number of CPU cores mostly for non-computational tasks (i.e., memory or IO device management). Our measurement at two points and projection to higher throughputs (assuming memory bandwidth problem is not present) show that *up to 67 CPU cores in a socket* are required to provide 75 GB/s (Figure 5a). Such number of cores is up to 3 times more than what a typical high-end CPU (e.g., 22-core Xeon [20]) provides.

Our CPU utilization breakdown shows that 85.2% of CPU utilization in *Write-only* and 50.8% of CPU utilization in *Mixed read/write* are due to memory management or accelerator scheduling related tasks (Figure 5b). The majority of overhead is for the data reduction table cache management (52.4%) and the remaining is for a unique chunk predictor that helps FPGA batch scheduling (32.7%).

4 DESIGN CHALLENGES AND OUR OBSERVATIONS

In this section, we perform a more detailed characterization of the baseline to determine the root causes of memory- and CPU bottlenecks. We also reveal the challenges to address the mentioned scalability limitations of the baseline.

4.1 Resolution of Memory Bottleneck

A naive solution is to replace the host DRAM with emerging higher bandwidth memory such as HBM [34]. However, such approach is not cost-effective or practically feasible due to the limited HBM capacity and its high price. The current generation of HBM supports only 8-16 GB memory capacity in a package [8, 34]. However, with a PB-scale SSD array and multi-TB sized data reduction tables, table caching requires 10s to 100s of GBs of host memory. Providing such amount of memory with HBM may be impractical due to its cost and the required modification to the motherboard. Therefore, we consider storage systems with typical DRAM memory.

Our analysis on memory bandwidth usage in the baseline provides two major observations.

Observation #1. Data reduction has operations with different memory-bandwidth and -capacity requirements. Our characterization of the baseline shows that 74.4-85.1% of the memory bandwidth is used by the operations that require less than 1 GB of memory capacity (Table 1). In contrast, data reduction table caching uses up to 25.7% of the memory bandwidth but requires 10s-100s GB of memory capacity. Our observation shows the need to separate the

Table 1: Breakdown of memory bandwidth utilization and memory capacity requirements of baseline components.

Data Path	Memory BW (Write-only)	Memory BW (Mixed)	Memory capacity
NIC ↔ host memory	23.6%	27.7%	KBs-MBs
Host memory (unique prediction)	23.7%	13.9%	MBs
Host memory ↔ FPGAs	25.4%	35.6%	MBs
Table cache management	25.7%	15.1%	10-100s GB
Host memory ↔ data SSD	1.7%	7.9%	KBs-MBs

bandwidth- and capacity-intensive operations and use two different memory types for a scalable data reduction system.

Observation #2. The baseline consumes up to 85.1% of the host memory bandwidth for simple data processing and data buffering operations. The data reduction software buffers the data in host memory and forwards them from one IO device to another (e.g., NIC→FPGA). Some of this data forwarding requires simple host-side processing. For example, the unique chunk predictor on the host operates on the buffered data before forwarding the data to the FPGA. Considering that most simple operations in data reduction use the host memory and complex data processing (e.g., compression) are on FPGAs, our second observation shows the potential to reduce the use of host-memory to achieve a more scalable storage system.

4.2 Resolution of Scheduling Overhead

Observation #3. Integration of hashing and compression in an accelerator requires heavy weight host-side accelerator task scheduling. CIDR aims to reduce the hardware costs by integrating hashing and compression in the same accelerator. However, such integration requires a host-side software to predict unique chunks in advance and efficiently schedule them to the HW compression cores. At high throughputs, we observe that such predictor is a major source of bottlenecks utilizing 32.7% of the total CPU resources and up to 23.7% of the total memory bandwidth.

4.3 Resolution of Table Caching Overhead

Observation #4. Neither host CPU/memory-only approaches nor accelerator-only approaches can provide scalable data reduction table caching. As the data reduction table management is a key component in the CPU utilization overhead, a naive solution is to fully offload the table cache management to an accelerator. However, such approach is not scalable to PB-scale capacity and Tbps-scale throughput. Accelerators usually have small on-board memory and limited PCIe bandwidth (e.g., 64 GB DRAM and 16 GB/s PCIe bandwidth in a VCU1525 FPGA board [47]). This limits both the table cache size and the rate of serving table cache misses. Therefore, both host-only approach and accelerator-only approach severely limit the supported storage capacity and throughput.

We characterized CPU utilization and memory capacity requirement of the components in data reduction table cache management (Table 2). We observed that 68.8% of the CPU overhead is consumed by operations for small sized data structures. These operations are for tree-structure cache indexing (with less than 3 GB tree for 500 GB table cache) and the SSD software stack used for cache line replacements (with KB-size control queues). The operations that directly require 10-100s of GB table cache content (i.e., scanning the cached content of a table bucket) only have 6.3% CPU overhead. Our characterization suggests that to provide scalable caching of data reduction tables, the CPU and the large host memory should be used for table cache content while offloading the indexing and management operations to a HW accelerator.

Table 2: Normalized CPU utilization and the memory capacity requirements of the components in table caching.

Component	CPU util (normalized)	Memory data structure	Memory capacity	Best place to run
Table cache tree indexing	43.9%	Tree nodes	Below 3 GB	Accelerator
Table SSD access	24.7%	IO control queues	KB-MBs	Accelerator
Table cache content access	6.3%	Table cache content	10-100s GB	Host
Table cache item replacement management	1.0%	LRU and free lists	MBs	Host or accelerator

4.4 Design Goals

Based on the challenges and our observations, to achieve a multi-Tbps, PB-scale end-to-end data reduction system, we have the following design goals:

- **Scalable accelerator scheduling.** To address the CPU- and memory-intensive task scheduling that state-of-the-art HW data reduction suffers from.
- **Scalable accelerator management.** To address huge host-side memory bandwidth utilization during client data buffering and data forwarding across accelerators, NIC, and SSDs.
- **Scalable data reduction table caching.** To address limited throughput scalability of straightforward table caching (CPU bottleneck in CPU-based caching and accelerator memory size or IO bandwidth bottleneck in accelerator-only caching).

5 FIDR: SCALABLE FINE-GRAIN INLINE DATA REDUCTION SYSTEM

5.1 Key Ideas

We design FIDR with three main ideas to achieve our design goals and ensure its high performance and capacity scalability. First, we significantly simplify the accelerator scheduling by offloading hashing from a dedicated accelerator to a NIC, and enable early detection of unique chunks. FIDR's in-NIC hashing makes the CPU- and memory-intensive the unique chunk predictor redundant, so we can remove such heavy-weight predictor from accelerator scheduling. In addition, by early detection of unique chunks, the NIC transfers only the unique chunks (instead of the total data chunks) to other devices which minimizes the PCIe bandwidth usage and improves the system scalability.

Second, we achieve scalable accelerator management by offloading the (client's request) data buffering from host memory to the NIC, and taking advantage of the PCIe peer-to-peer feature for direct data transfers across NIC, compression accelerators and data SSDs. All these operations almost completely bypass the host memory, which maximizes the performance scalability.

Third, we propose hybrid host CPU, memory, and FPGA acceleration which ensures scalable data reduction table caching. In this hybrid approach, we use the FPGA logic and FPGA-board memory for CPU-intensive operations that require low memory capacity (e.g., tree cache indexing). We use the host memory for holding large cached table content and the CPU for lightweight operations that directly access such cached content. With our proposed hybrid acceleration, the CPU overhead is mostly eliminated, and a form of near data operation for both CPU and FPGA is realized, which boosts the scalability of table caching.

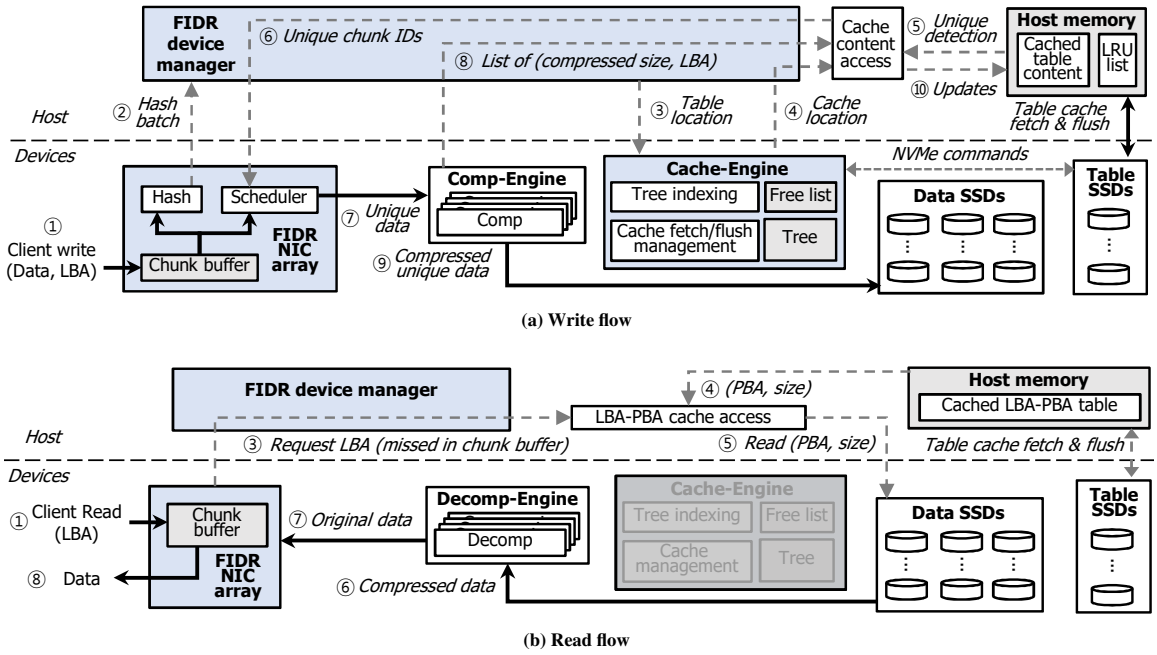


Figure 6: Our proposed system architecture.

5.2 FIDR's Key Components

FIDR has five main components: three hardware accelerator modules, FIDR software and SSDs. In the following, we briefly explain FIDR-specific components.

5.2.1 FIDR NIC. It is a NIC with added support for scalable data reduction. A FIDR NIC does in-NIC buffering of client requests, hashing of data (for write requests), and other tasks such as batch making (scheduling) of unique chunks for transferring to other dedicated accelerators.

5.2.2 FIDR Compression/Decompression Engine. FIDR uses a dedicated accelerator for compression (decompression), named FIDR Compression (Decompression) Engine. During write request handling, this accelerator receives batches of unique chunks for compression, and during read request handling, decompresses batches of compressed chunks.

5.2.3 FIDR Cache HW Engine. It consists of the hardware modules of FIDR that accelerate table caching. In particular, it includes hardware-based tree indexing and management of fetching and flushing of table cache lines.

5.2.4 FIDR Software. This is the host-software of FIDR that orchestrates the communication of FIDR HW devices, in addition to including the software modules of the FIDR table cache management (e.g., scanning the cached table content in host memory).

5.3 Simplified System-wide Operational Flow

Figure 6a shows the simplified write flow of FIDR. When the FIDR NIC receives a client's request (data, LBA), it buffers the request in the NIC (①), and immediately acknowledges write completion to the client, so that the client does not suffer from the long backed latency. When enough chunks are buffered, FIDR NIC calculates the hash value for each chunk and sends the hashes to the FIDR device manager (②), a module in FIDR software that manages communication between the hardware devices in FIDR. Based on the hash values, it calculates the table location information (i.e., bucket locations) and sends it to the FIDR Cache HW-Engine (③). FIDR Cache HW-Engine searches the cache (tree) index to find the location of the target table bucket in the table cache. If the hash value is not present, FIDR Cache HW-Engine fetches the respective table bucket from the table SSDs. After the host gets the cache location information (④), FIDR software scans the target cache lines in host memory and determines duplicate/unique status of each chunk (⑤). Then, FIDR NIC receives the unique/duplicate status of each chunk (⑥), schedules a batch of unique chunks and sends it to FIDR Compression Engines (⑦). After the total size of compressed chunks in a Compression Engine reaches a threshold (e.g., 4 MB), it sends the compressed sizes and the metadata associated with each chunk (e.g., LBAs) to the host (⑧). Finally, FIDR software selects an empty location of a specific data SSD and informs the SSD with the required metadata for writing the compressed data batch (metadata=total size of compressed batch, Compression Engine PCIe address, SSD destination address). Next, the specified data SSD fetches the compressed data from the memory of FIDR Compression Engine, and writes the data to its flash cells (⑨). Finally, the FIDR software updates the cache content for the newly written data (⑩).

The read request handling in FIDR has up to 8 steps but with less complexity than write requests (Figure 6b). Upon receiving a read request from a client (①), FIDR NIC first searches the in-NIC

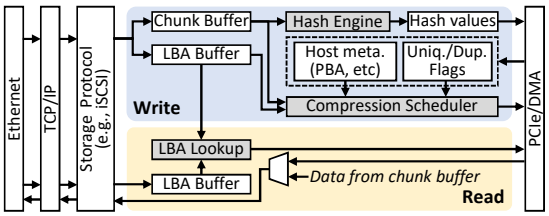


Figure 7: Proposed NIC microarchitecture to minimize the inter-device communication overhead.

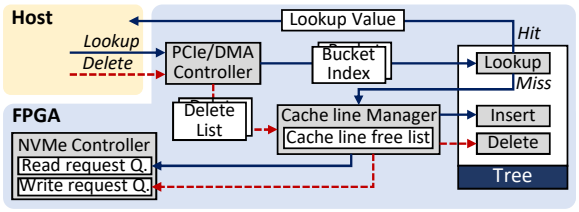


Figure 8: Proposed FIDR Cache HW-Engine's microarchitecture to accelerate operations for data-reduction table-cache.

buffer of the previously received write requests (②). If it finds the same LBA in the buffer, it sends the data to the client; otherwise, it forwards the LBA to the FIDR software (③). The host-side software searches the LBA-PBA table to get the PBA and the compressed size for the requested LBA (④). Then FIDR (device manager) software orchestrates the data transfer from data SSDs to FIDR Decompression Engines (without host memory involvement) (⑤, ⑥) and direct transfer of decompressed data from those FPGAs to the NIC (⑦). Finally, the NIC sends the requested data to the client (⑧). In the following sections, we provide the microarchitecture details of FIDR NIC and FIDR Cache HW Engine.

5.4 FIDR NIC Microarchitecture

FIDR NIC is a high performance NIC with added support for data reduction acceleration. When a client request arrives at a high performance NIC, the NIC usually does more tasks than just processing the ethernet packets. In this case, instead of sending the packet data to the host for processing, the NIC itself parses the content in deeper layers. For example, a high performance NIC also parses the TCP/IP packets and even decodes the client requests embedded as a certain storage protocol layer (e.g., iSCSI). FIDR NIC adds an additional layer of buffering and processing for data reduction support. Figure 7 shows FIDR NIC's specific operation flow for both write and read.

For write request handling, the FIDR NIC buffers data and LBAs in its respective in-NIC buffers, hashes each chunk of a batch of requests and sends the hash values to the host via the PCIe/DMA controller. After the host determines the status of each chunk (e.g., unique, duplicate), FIDR NIC receives such information (i.e., uniqueness flags) along with additional metadata containing the destination of each chunk. Next, our compression scheduler scans the list of unique/duplicate flags and prepares a batch of *unique* chunks. Finally, FIDR sends the batch of unique chunks along with corresponding metadata (e.g., PBA) to the Compression Engines through the PCIe/DMA controller.

For read request handling, FIDR NIC buffers LBAs of incoming read requests in the LBA buffer. Then *LBA Lookup* module scans the LBA buffer of write requests to find a possible match. If the lookup hits, FIDR NIC reads the associated data from its data buffer (i.e., chunk buffer), encodes it based on a storage protocol (e.g., iSCSI) and sends the encapsulated data to TCP/IP core, ethernet core, and finally the client. If the lookup misses, FIDR NIC sends LBAs through the PCIe/DMA controller to the host. Then, the host orchestrates the direct transfer of the requested compressed data from the SSDs to the FIDR Decompression Engine. After the Engine decompresses the data, FIDR software informs FIDR NIC to fetch the decompressed data from the Decompression Engine's memory. Finally, FIDR NIC loads the decompressed data, encodes them based on a storage protocol, and sends them to the client.

5.5 FIDR Cache HW-Engine Microarchitecture

FIDR Cache HW-Engine reduces the host-side pressure by using hardware-based tree indexing for Hash-PBN table caching and hardware acceleration for managing the cache line replacements. Note that Hash-PBN table is only accessed during write requests, thus FIDR Cache HW Engine is also only used during handling the client's write requests.

Figure 8 shows the microarchitecture of FIDR Cache HW-Engine. After the host calculates the target (bucket) location that should be accessed in the data reduction table, it sends a batch of such bucket indexes to the Cache HW-Engine. The Cache HW-Engine has two jobs (1) searching (and updating if necessary) the tree that holds (key, value) pairs of (table bucket index in SSD, its location in the host-side table cache) and (2) handling Table SSD accesses for fetching a missed cache line or flushing a dirty cache line to SSD upon replacement. After the batch of requests is handled, the Cache HW-Engine sends a batch of cache line locations to the host; therefore, FIDR host-side software can access the cache content for duplicate/unique chunk detection.

When a table bucket index does not exist in the tree structure (i.e., not in the host-side cache), FIDR Cache HW-Engine follows a few steps. It selects a free cache line from the free list, queues a read request to the Table SSD for bucket fetching and inserts the (key, value) pair of (bucket index, new cache line index) into the tree. To ensure the free list is never empty, FIDR Cache HW-Engine periodically requires to delete some cache lines and flush their content to Table SSD if they are dirty. As the host software accesses the table cache content, the cache LRU list is also kept in the host side. To minimize the interaction with the host, FIDR HW-Engine periodically receives batches of top LRU list items for deletions and handles the remaining required operations completely in the HW-Engine.

5.5.1 Optimization: concurrent HW-tree updates. Providing high throughput data reduction requires a high throughput tree indexing to efficiently support table cache line lookups and replacements. This requirement becomes more important when considering the low locality of Hash-PBN table accesses. However, it is very challenging for resource-limited FPGAs to have a fully concurrent tree for *update requests* (i.e., inserts and deletes) due to the chance of tree node conflicts (i.e., update requests may modify same nodes during

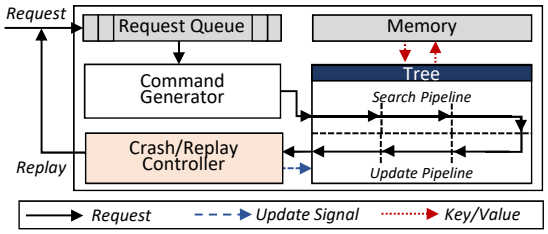


Figure 9: Proposed optimization for FIDR Cache HW-Engine to support concurrent updates on FPGA-based tree indexing.

merge/split). In this regard, some existing work significantly limited the degree of concurrent updates (e.g., no concurrent deletes [48]).

To handle concurrent tree update requests, we propose an improved hardware-based tree indexing. Our HW-based tree has a command generator to make proper control signals for a given request, a tree structure to manage entries, and a crash/replay controller to maintain the correct tree state (Figure 9). The HW-tree has two separate pipelines for search and update, and the number of stages of each pipeline is same as the tree height.

Our key idea to support concurrent update requests is *speculative execution and recovery*. As hash values are highly random and there are many stages in the tree (e.g., 14 for 100 GB cache indexing), the chance that multiple requests update the same node (in a short period of time) is extremely low (e.g., less than 0.1% for our workloads). Therefore, we speculatively issue multiple update requests and run recovery for the wrong speculation. A very low rate of miss-speculations makes the performance penalties negligible.

Our HW-tree guarantees the correctness of the tree for the concurrent update (especially, delete) requests as follows. For cache replacement, the command generator issues an update request, which flows through search and update pipeline in series. In the search pipeline, the request records the information of the traversed nodes, which will be visited in the update-pipeline in reverse order (i.e., from the leaf to the root). During this reverse-traversal, each request first checks whether prior requests have changed the same node or not (Algorithm 1). If such nodes exist, then the speculation was wrong and the changes made by the request should be invalidated. To ease this process, our HW-tree postpones applying the changes made by the request until the speculation is resolved (line 4).

Before committing the request, the crash/replay controller examines *is_crash* of the requests to check the state of speculation (Algorithm 2). If the speculation was correct, then the crash/replay controller sends the update signal to the tree, so that the tree applies the changes (line 4-7). Otherwise, the crash/replay controller ignores all changes made by the request and re-inserts it into the request queue for replaying (line 2). Note that, such replay incurs only negligible performance overhead because the replay rarely happens and has simple logic.

5.6 PCIe Bandwidth Considerations

To ensure the scalability of FIDR, we make two design choices to control the PCIe bandwidth utilization and PCIe transfer directions. First, we make multiple groups of three types of devices (NIC, Compression Engine, data SSDs). Each group is placed under a separate

Algorithm 1 Issue an update request at n^{th}-level of the tree

Require: Node to be updated *node*
Require: Request for the node *request*
Require: List of speculatively updated nodes *spec_updated_node*
1: **if** (*node* **or** *node.neighbor*) **in** *spec_update_node* **then**
2: *request.is_crash* ← 1
3: **else** // Speculative execution
4: *request.state* $[n]$ ← *update_node_state* $(node, command)$
5: *spec_updated_node.insert* $(node)$
6: **end if**
7: **return** *request*, *spec_update_node*

Algorithm 2 Commit an update request

Require: Request for the node *request*
Require: A list of nodes accessed by the request *nodes*
Require: List of pending requests *request_queue*
Require: List of speculatively updated nodes *spec_updated_node*
1: **if** *request.is_crash* **then** // Re-insert the request for replay
2: *request_queue.push_back* $(request)$
3: **else** // Commit the changes
4: **for** $i \leftarrow 1$ **to** *tree_height* **do**
5: *nodes* $[i]$ *.state* ← *request.state* $[i]$
6: *spec_updated_node.remove* $(nodes[i])$
7: **end for**
8: **end if**

PCIe switch which ensures maximum peer-to-peer PCIe bandwidth. Second, our choice of offloading only metadata cache management to Cache HW-Engine and keeping the table cache content in host memory guarantee scalable metadata caching. Communication with Cache HW-Engine requires negligible PCIe bandwidth (e.g., 200 MB/s for 100 GB/s data reduction considering 8 byte-cache index per 4 KB request). For table cache content, table SSDs need to access the content on host memory. Such access is through PCIe root complex which has massive bandwidth (e.g., 128 GB/s with AMD EPYC [7]). Note that depending on the table cache miss-rate, the required root complex bandwidth is set. In practice, with low cache miss-rate, FIDR incurs only a small PCIe bandwidth overhead (e.g., 10 GB/s PCIe bandwidth for 100 GB/s data reduction assuming 10% table cache miss-rate).

6 IMPLEMENTATION

6.1 Overall System

We implemented HW-accelerators of FIDR in three FPGA boards: one for our custom NIC, another for Compression Engine, and the other for Cache HW-Engine. FIDR Compression Engine is similar to the baseline HW-Engine except two parts: (1) we removed the hashing cores (offloaded to NIC) and (2) the FPGA transfers only the metadata (compressed data sizes, etc.) to the host and keeps the compressed data for direct transfer to data SSDs. We explain the implementation details of the FIDR NIC, along with FIDR Cache HW Engine in Section 6.2 and 6.3, respectively.

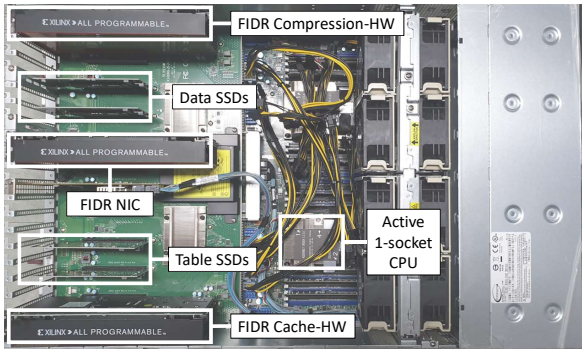


Figure 10: FIDR Prototype

For proper system-wide operations, we also implemented some software modules of FIDR in the Linux Kernel (in a device driver). We used software-based management for (NVMe) data SSDs and implemented a hardware-based management for (NVMe) table SSDs. Write requests to data SSDs for the compressed chunks are sequential, which have tolerable overheads. Therefore, for simplicity, NVMe submission/completion queues of data SSDs are in host memory (similar to default system) and only the data come from Compression Engines (instead of host memory). On the contrary, table SSD accesses are random small blocks which have significant management overhead on CPU. Therefore, we designed table SSD's submission/completion queues to be in the HW Cache Engine and modified the SSD driver to support this.

6.2 FIDR NIC

We implemented a FIDR NIC on an FPGA board with target throughput of 64 Gbps. The TCP offload engines in our implementation consist of two 32 Gbps instances and are optimized for large network packets (i.e., the common scenario for a storage environment that a client requests data blocks larger than 1 KB). We made a simplified protocol (instead of a complete protocol like iSCSI) to work as our storage access protocol. The protocol follows a basic flow from a client's point of view: read-wait-acknowledgment (with data) for read requests and write-wait-acknowledgment for write requests. The encoding mainly includes the operation type (i.e., read, write or acknowledgment), the requested address (i.e., LBA) and data. For hashing, we used instances of an open-source SHA-256 core [13].

6.3 FIDR Cache HW-Engine

The implementation of a FIDR Cache HW-Engine consists of three major components: tree indexing, cache free list management and NVMe SSD controllers. We relied on an existing FPGA-based balanced tree [48] as a core part of the tree indexing, but we applied two modifications to boost its capacity and throughput support. First, unlike the original paper [48] that uses maximum 2 keys per node in all stages, we use larger nodes (with 16 keys) for the leaf stage. The larger sized gap of leaf stage and non-leaf stages enables us to make a very large tree while placing all non-leaf stages in single-cycle accessible on-chip memory. For example, in our VCU1525 FPGA board, increasing the leaf node size allows indexing up to 100 GB cache size with a 13-level tree using FPGA's fast on-chip memory and only the leaf node being on the FPGA-board DRAM. The second modification is adding concurrent, pipelined updates as explained in Section 5.5.1.

For the cache free list management, we implemented a circular buffer which is in FPGA-board DRAM due to its size (for a large number of cache lines). While the free list is in DRAM, accesses to it are sequential and each entry has a small size. Therefore, a single access to the DDR controller with a 512-bit bus can receive many entries of the free list and have negligible DRAM access overhead.

Table 3: Summary of workloads.

Workload	Dedup. ratio	Comp. ratio	Table cache hit rate	Trace block	Total size
Write-H	High (88%)	Medium (50%)	High (90%)	2M IOs from Mail	176M IOs (704GB)
Write-M	High (84%)	Medium (50%)	Medium (81%)	11M IOs from Mail	176M IOs (704GB)
Write-L	Medium (43.1%)	Medium (50%)	Low (45%)	7.5M IOs from WebVM	180M IOs (720GB)
Read-Mixed	Half reads and half writes. Writes are same as write-H. Reads are random valid addresses.				

7 EVALUATION

7.1 Environment and Methodology

We evaluated our scheme using three methods. First, we use end-to-end testing involving two machines (one as a client and one as a storage server) to measure CPU utilization and host memory bandwidth utilization at a line-rate. With the network bandwidth limitation of our prototyping environment, we also use a single machine for measuring the performance of our proposed hybrid table cache acceleration. Third, we make a simulation model based on our previous measurements and simulate the end-to-end behavior for 1 Tbps per-socket target throughput.

Our baseline (CIDR [5] in Section 2.3) is the state-of-the-art HW-based data reduction system with added capability to operate on small chunks with software-based table caching. The (Hash-PBN) table cache has 4-KB cache lines corresponding to 4-KB table buckets. For mapping of bucket indexes to each cache line, we used an open-source high performing B+ tree [32][2] in the baseline.

Our server machine uses a single active socket with Intel E5-2650 v4 CPU, four Samsung Pro 970 1-TB SSDs (two for data SSDs and the rest for Table SSDs), and three VCU1525 Xilinx FPGA boards as FIDR NIC, FIDR Compression Engine and FIDR Cache HW-Engine (Figure 10). Note that for the total 2-TB unique compressed data storage in data SSDs, only 60 GB of Table SSDs are required (assuming 50% deduplication ratio and 50% compression ratio).

We evaluate our scheme on both read and write requests. However, we mainly focus on write-intensive workloads, because they have become more widespread in enterprise environments [10], and directly affect SSD's lifetime.

We generated four IO workloads based on real traces to enable testing various workload patterns. We extract portions of real traces[3] and use as our workload building blocks. We considered five factors

[2]The tree indexing is based on Intel PALM [33]

[3]Data reduction operation depends on data content and access patterns. Due to security concerns, there are no public IO traces that contain real data content. Example real IO

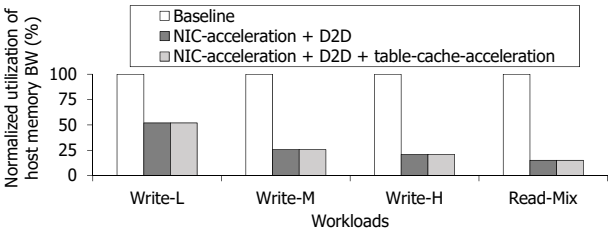


Figure 11: Reducing host memory BW utilization using FIDR.

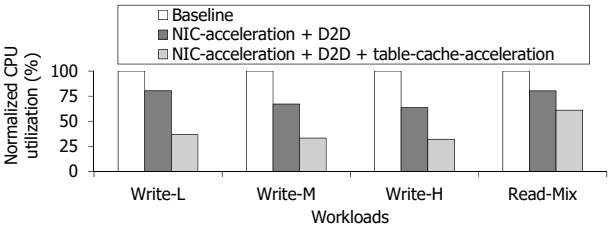


Figure 12: Using FIDR to eliminate CPU utilization bottleneck (memory/IO management overhead during data reduction).

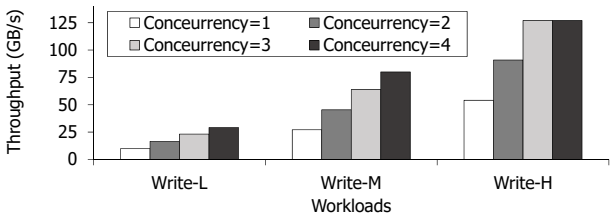


Figure 13: Boosting throughput using our proposed multi-update mechanism for FPGA-based tree indexing.

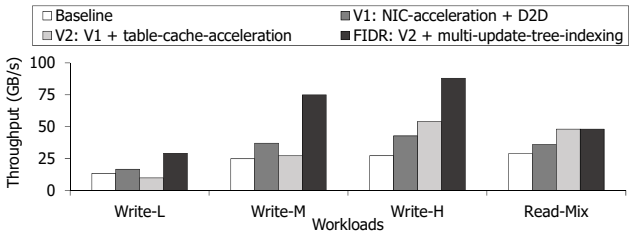


Figure 14: Overall throughput evaluation of different techniques in our scheme.

when generating the workloads: (1) We extracted a portion of real traces to achieve a target table cache hit-rate for a fixed small cache size. (2) We replicated the extracted trace many times to generate a large enough workload. (3) To prevent replication from resulting in a 100% duplicate workload, we applied minor systematic modifications to the trace content across each replicate. This makes the deduplication ratio of a large number of trace replicates same as a single replicate. (4) We set the compressibility to 50% by concatenating a 50% compressible string to all trace requests. (5) We set the reduction table size assuming 500 GB unique compressed storage, and use 2.8% in-memory table caching. The detail of each workload (three write-only and one read-write mixed) are shown in Table 3.

7.2 Memory Bandwidth Utilization

FIDR reduces the host-side memory bandwidth utilization by up to 79.1% in write-only workloads and 84.9% in the read mixed workload (Figure 11). Performing data reduction support tasks in a NIC, along with exploiting P2P capability of PCIe devices for direct NIC-FPGA-SSD transfers, enabled FIDR to eliminate the host-side memory bandwidth pressure. With higher table cache hit-rates (i.e., less memory bandwidth use in the table caching), FIDR becomes more effective in removing memory bandwidth pressure. For *Read-Mixed*, FIDR removes the pressure from the host-side memory for handling read requests and a large portion of the overhead for handling write requests. This resulted in a successful reduction of the memory bandwidth utilization by 84.9%.

7.3 CPU Utilization

FIDR reduces the CPU utilization by up to 68% in write-only workloads and by 39% in mixed read-write workloads. FIDR NIC-based early hashing removes the need for the unique chunk predictor of the baseline, and reduces the CPU utilization by 20-37%. By hardware offloading of selected memory/IO management tasks of table caching, FIDR further reduces CPU utilization by 19-44% points. When a workload has low table cache hit-rates (e.g., *Write-L*), the system requires more frequent tree updates and table SSD read/writes. Therefore, the baseline requires more CPU cores to achieve the same throughput. However, as Figure 12 shows, FIDR eliminates such increased overhead of memory management.

7.4 Table Cache Acceleration Throughput

FIDR's FPGA-based cache indexing and cache line allocation achieve up to 54 GB/s with a single-update tree and almost linear scalability with our multi-update optimization (Figure 13). For a workload such as *Write-M* that has a large number of cache line replacements (tree index updating), a single update tree indexing results in 27.1 GB/s throughput. Our proposed crash/replay optimization enables up to 4 concurrent updates in the HW-based tree structure, thus results in almost linear throughput scalability up to 63.8 GB/s. Such linear performance increase is assigned to negligible overhead (below 0.1% concurrent update crash/replays) of our proposed scheme. Workloads with high cache hit-rate (i.e., *Write-H*) also benefit from our optimization. but their throughput becomes limited to about 127 GB/s due to saturating the FPGA-board DRAM bandwidth.

7.5 Overall Throughput

FIDR achieves significant throughput improvement of up to 3.3× for write-only workloads and up to 1.7× for the mixed read-write workload (Figure 14). We build a basic simulation model based on our measured CPU utilization, memory bandwidth and the throughput of FIDR Cache HW-Engine. Then we project the system throughput assuming a high-end 22-core CPU (e.g., Intel Xeon E5- 4669

traces that are applicable to deduplication (but not compression) are FIU IO traces [24, 39], which include the hash values of data block writes along with the IO addresses.

Table 4: Resource utilization of FIDR custom NIC

Write-only workload			
-	Data reduction support	Basic NIC + TCP Offload	Total
LUTs	125 K (10.7%)	166 K (14.0%)	290 K (24.5%)
Flip flops	128 K (5.4%)	169 K(7.1%)	296K (12.5%)
BRAMs	95 (4.4%)	1024 (47.4%)	1119 (51.8%)
Mixed workload (50% read, 50% write)			
	Data reduction support	Basic NIC + TCP Offload	Total
LUTs	84 K (7.1%)	166 K (14.0%)	249 K (21.1%)
Flip flops	87 K (3.7%)	169 K(7.1%)	255K (10.8%)
BRAMs	75 (3.5%)	1024 (47.4%)	1099 (51.0%)

Table 5: Resource utilization of FIDR Cache HW-Engine

		All	Except table SSD access	
			Medium tree	Large tree
Size & BW	Table cache size	410 MB	410 MB	99,645 MB
	# Tree levels on-chip/off-chip	8/1	8/1	13/1
	Table SSD BW	2 GB/s	-	-
	Estimated max throughput (for Write-M)	10 GB/s	80 GB/s	64 GB/s
FPGA resource	LUTs	320 K (27.1%)	316K (26.7%)	348K (29.4%)
	Flip flops	160K (6.8%)	154K (6.5%)	137K (5.8%)
	On-chip BRAM	218 (10.1%)	202 (9.3%)	390 (18.1%)
	On-chip URAM	-	-	756 (78.8%)

v4 [20]). The result shows that our proposed NIC-based acceleration and direct data transfer across PCIe devices provide up to 1.6x speedup over the baseline. We also observe that adding the HW-based table cache management may result in throughput reduction in *Write-L* and *Write-M*. This throughput degradation is due to the low throughput of HW-based tree indexing with single update capability. By adding our optimization for multiple concurrent updates in the HW-based tree indexing, the throughput is boosted. Note that our optimization did not increase the throughput of *Read-Mixed* due to the inherent CPU utilization overhead of the data SSD software stack for handling read requests. We can also offload this NVMe software stack to FPGA, but we left it as future work.

7.6 Request Latency

7.6.1 Write Latency. FIDR does not affect the commit latency of write requests (i.e., same latency as a system with no data reduction). We achieve this by assuming a battery-backed system which enables non-volatile buffering in FIDR NIC. Thus, when the server buffers client's requests in such non-volatile buffer, the server immediately sends write completion to the client, hiding the backend operation latency. Note that many commercial storage servers hide write latency of the backend in a similar way (e.g., using NVRAM or a battery-backed system) [1, 19].

7.6.2 Read latency. FIDR's shorter datapath from SSDs to NICs improves the read latency over the baseline HW-based data reduction system. We measured the latency of a client's 4-KB read request that is served as a part of a batch of read requests. Our measurement shows that FIDR reduces the server-side latency (SSDs↔NICs) from 700 us to 490 us.

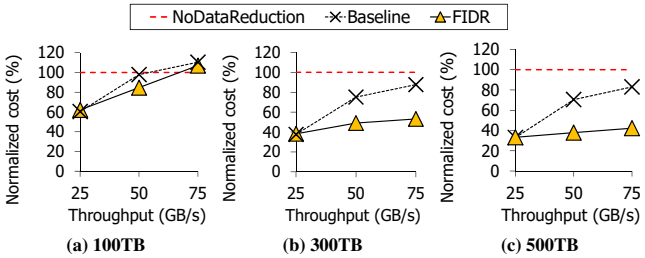


Figure 15: Scalability of FIDR to high throughputs while significantly reducing storage costs (lower values on y-axis are better).

7.7 FPGA Resource Utilization

7.7.1 FIDR NIC. Our added data reduction modules for a FIDR NIC consume negligible FPGA resources. Practically, only SHA2 hash cores and the DDR controllers (for data buffering) consume FPGA resources, which are minimal. For write-only workloads, our proposed architecture adds as little as 4.4% point BRAM and 10.7% point LUTs to the FPGA resource utilization (Table 4). With a mixed read-write workload, as less hashing is required, such resource utilization becomes as low as 7.1% LUTs, and 3.5% BRAMs. Considering that the default operations of a NIC for storage (ethernet packet handling+ TCP offload engine + iSCSI processing) can be implemented in a fixed ASIC, we expect that our added data reduction support can be easily implemented in even low-end FPGAs.

7.7.2 FIDR Cache HW-Engine. FIDR Cache HW-Engine achieves 80 GB/s with less than 26.7% LUT utilization for medium size tree indexing, and provides estimated 64 GB/s with indexing capability of 200x larger cache size (Table 5). Our workloads were suitable for a table cache size of 410 MB. Such cache size required 9 levels of tree indexing, which could fit completely in FPGA's on-chip memory. A PB-scale system may require a cache that is even 100 GB large (i.e., the tree becomes 14 levels deep). Thus to provide throughput numbers closer to PB-scale table caching, we keep the leaf node of our 9-level tree in the FPGA board DRAM (similar to 14-level deep tree). We observe that our medium-size tree indexing acceleration has 80 GB/s, while a 200x larger tree has estimated 64 GB/s. Such high performance was achievable by relying on FPGA's large SRAM cells (e.g., on-chip URAM), so that increasing tree depth up to 14 does not add any new off-chip memory stages.

7.8 Cost Analysis

We calculated the cost saving of FIDR considering the amount of data SSDs that are saved by data reduction (for 50% deduplication and 50% compression). We treat the cost as the remaining data SSDs after data reduction, and the added data reduction cost on CPU, FPGA, DRAM and table SSDs. Following prices on major online shops (e.g., Amazon) and vendor websites, we assume 0.5 $/GB for SSDs, 5.5 $/GB for DRAM, $7000 22-core CPU (e.g., Intel E5-4669V4 [20]), and $ 7000 high-end FPGA (e.g., Xilinx VCU9p). We scale the CPU and FPGA cost based on their resource utilization and assume only 70% of FPGA resources to be practically usable.

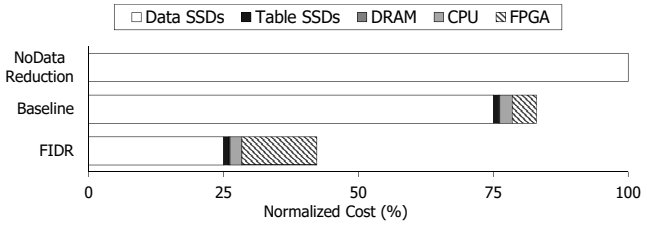


Figure 16: Cost-effectiveness of FIDR with breakdown of cost components (at 75 GB/s and 500 TB effective capacity)

FIDR significantly reduces the storage cost compared to a system with no data reduction, and scales to high throughputs (Figure 15). We observe that the benefits of SSD cost reduction outweigh the cost of added CPUs and FPGAs. As the throughput increases, FIDR overhead increases; however, because the target storage capacity usually increases with the performance, FIDR is still cost-effective at high capacities. For example at 500 TB target capacity, FIDR cost saving only changes from 67% at 25 GB/s to 58% at 75 GB/s. Note that FIDR has a similar cost to the baseline at low throughput, but FIDR can scale to higher throughput (e.g., over 25 GB/s per socket) that the baseline fails to achieve. Therefore, the baseline has to do *partial* data reduction to catch up with the throughput and this significantly increases its cost over FIDR (Figure 16).

8 DISCUSSION

In this paper, we only focused on the data reduction acceleration in a properly load-balanced single-workload environment. In multi-tenant environments or single-workload environments with skewed accesses, other problems such as resource contentions happen. Many existing studies focus only on such contentions. We can easily integrate existing solutions by modifying FIDR software. For example, to address table cache contention, instead of a basic LRU replacement policy, we may use a prioritized LRU policy that considers each workload's locality (similar to [44]). For imbalanced read accesses to the data SSDs, we can extend FIDR software and the LBA-PBA table to maintain frequently accessed blocks in main memory.

9 RELATED WORK

9.1 Existing Data Reduction Systems

Plenty of work focused on implementing or optimizing software-based data reduction [17, 38, 40, 45, 50]. Some of this work improves deduplication write throughput through sampling chunks and applying deduplication only on them [17]. Some aims to reduce table caching latency by exploiting content locality and adopting metadata prefetching [50]. Overall, most of these studies are for hard disk drives (HDDs) and backup storage environments. Furthermore, existing work has shown that software-based data reduction cannot provide high throughput for SSD arrays [5].

Therefore, recent work has focused on offloading compute-intensive data reduction operations from CPU to an accelerator. Examples include GPU- and FPGA-accelerated variable sized chunking [9, 28], FPGA-based compression [2, 16], and FPGA-based deduplication [6]. Some works also offload both hashing and compression to FPGAs [5, 21]. We adopt such existing work in our baseline and show their scalability limitations due to their memory overheads.

In-SSD deduplication [11, 22] and in-SSD compression [26] have been also proposed as another way to offload the CPU overhead to an IO device. However, as shown in CIDR [5], such in-SSD data reduction approaches suffer from low deduplication and compression opportunities in a large SSD array.

Lastly, some existing studies modify the table caching algorithm to improve its hit-rate [29, 43, 44]. For example, HANDS uses a statistical method to group requests and enables request prefetching to improve the table cache hit-rate [43]. These studies mostly aim to reduce the table access latency, as the table is kept in slow storage such as HDDs. Such techniques are orthogonal to ours and we can adopt them for a more scalable storage system.

9.2 Optimizing Data Movements

A few existing work has used P2P capability of PCIe devices for direct data transfers across a number of devices [3, 4, 25, 41]. We use such techniques as technology enablers to do direct data transfers across NIC↔FPGA↔SSD and adapt them to our data reduction environment. As an example of device-to-device transfers, GPUDirect Async [3] can be considered. It enables direct data transfer between GPUs and NICs. Other existing studies focus on improving performance by unifying or coupling different software stacks [23, 49]. For example, ReFLEX tightly couples the network and storage stacks to guarantee stable and fast accesses to SSDs in a remote node [23].

9.3 Smart NICs

Programmable NICs have become of interest in recent years [15, 27, 30]. Some of this work offloads the host networking stack to NICs (e.g., [15]). The other focuses on load balancing and efficient microservice execution on servers with smart NICs [30]. Unlike this work, we use NIC offloading for the scalable accelerator management and the accelerator scheduling in a data reduction system.

10 CONCLUSION

In conclusion, we proposed a scalable end-to-end storage system to enable fine-grain inline data reduction. We proposed a data reduction friendly NIC architecture and direct data transfer across NIC↔FPGA↔SSDs to minimize host-side DRAM bandwidth utilization. We also proposed hybrid CPU/FPGA acceleration of data reduction table caching. Our scheme significantly improved the memory management scalability and boosted the overall throughput.

ACKNOWLEDGMENTS

This work was partly supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT (MSIT) (NRF-2015M3C4A7065647, NRF2017R1A2B3011038), Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.1711080972), and Creative Pioneering Researchers Program through Seoul National University. We also appreciate the support from Samsung Advanced Institute of Technology (SAIT), and Automation and Systems Research Institute (ASRI), Inter-university Semiconductor Research Center (ISRC) at Seoul National University.

REFERENCES

[1] Deepstorage.net. 2012. Storage efficiency imperative: an in-depth review of storage efficiency technologies and the solidfire approach. http://www.deepstorage.net/NEW/reports/SolidFireStorageEfficiency.pdf.

[2] Mohamed S Abdelfattah, Andrei Hagiescu, and Deshanand Singh. 2014. Gzip on a chip: High performance lossless data compression on fpgas using opencl. In *Proceedings of the International Workshop on OpenCL 2013 & 2014*. ACM, 4.

[3] Elena Agostini, Davide Rossetti, and Sreeram Potluri. 2017. Offloading communication control logic in GPU accelerated applications. In *17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 248–257.

[4] Jaehyung Ahn, Dongup Kwon, Youngsok Kim, Mohammadamin Ajdari, Jaewon Lee, and Jangwoo Kim. 2015. DCS: a fast and scalable device-centric server architecture. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 559–571.

[5] Mohammadamin Ajdari, Pyeongsu Park, Joonsung Kim, Dongup Kwon, and Jangwoo Kim. 2019. CIDR: A cost-effective in-line data reduction system for terabit-per-second scale SSD arrays. In *High Performance Computer Architecture, 2019. HPCA-25. 25th Annual IEEE International Symposium on*. IEEE.

[6] Mohammadamin Ajdari, Pyeongsu Park, Dongup Kwon, Joonsung Kim, and Jangwoo Kim. 2018. A Scalable HW-Based Inline Deduplication for SSD Arrays. *IEEE Computer Architecture Letters* 17, 1 (2018), 47–50.

[7] AMD. 2017. AMD EPYC 7000 series. https://www.amd.com/en/products/epyc-7000-series.

[8] Anton Shilov. 2019. Samsung HBM2E 'Flashbolt' memory for GPUs: 16 GB per stack, 3.2 Gbps. https://www.anandtech.com/show/14110/samsung-introduces-hbm2e-flashbolt-memory-16-gb-32-gbps.

[9] Pramod Bhatotia, Rodrigo Rodrigues, and Akshat Verma. 2012. Shredder: GPU-accelerated incremental storage and computation.. In *FAST*.

[10] Robert Birke, Mathias Bjoerkqvist, Lydia Y Chen, Evgenia Smirni, and Ton Engbersen. 2014. (Big) data in a virtualized world: volume, velocity, and variety in cloud datacenters. In *Proceedings of the 12th USENIX conference on File and Storage Technologies*. USENIX Association, 177–189.

[11] Feng Chen, Tian Luo, and Xiaodong Zhang. 2011. CAFTL: A Content-Aware Flash Translation Layer Enhancing the Lifespan of Flash Memory based Solid State Drives.. In *FAST*.

[12] Chris M. Evans. Jan 2017. HPE 3PAR Adaptive Data reduction: A competitive comparison of array-based data reduction. https://www.hpe.com/h20195/v2/getpdf.aspx/4AA6-6256ENW.pdf.

[13] Java Doin. 2016. Open-source SHA-256 hardware core. http://opencores.org/project,sha256_hash_core.

[14] Ahmed El-Shimi, Ran Kalach, Ankit Kumar, Adi Ottean, Jin Li, and Sudipta Sengupta. 2012. Primary Data Deduplication-Large Scale Study and System Design.. In *USENIX ATC*.

[15] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. 2018. Azure accelerated networking: SmartNICs in the public cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. 51–66.

[16] Jeremy Fowers, Joo-Young Kim, Doug Burger, and Scott Hauck. 2015. A scalable high-bandwidth architecture for lossless compression on fpgas. In *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*. IEEE, 52–59.

[17] Fanglu Guo and Petros Efstathopoulos. 2011. Building a High-performance Deduplication System.. In *USENIX annual technical conference*.

[18] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, et al. 2018. Applied machine learning at facebook: A datacenter infrastructure perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 620–629.

[19] Hewlett Packard Enterprise. Nov 2018. HPE 3PAR StoreServ Architecture. https://www.hpe.com/h20195/v2/getpdf.aspx/4aa3-3516enw.pdf.

[20] Intel. 2016. Intel Xeon Processor E5-4669 v4. https://ark.intel.com/products/93805/Intel-Xeon-Processor-E5-4669-v4-55M-Cache-2_20-GHz.

[21] Kentaro Katayama, Hidetoshi Matsumura, Hiroaki Kameyama, Shinichi Sazawa, and Yasuhiro Watanabe. 2017. An FPGA-accelerated high-throughput data optimization system for high-speed transfer via wide area network. In *2017 International Conference on Field Programmable Technology (ICFPT)*. IEEE, 211–214.

[22] Jonghwa Kim, Choonghyun Lee, Sangyup Lee, Ikjoon Son, Jongmoo Choi, Sungroh Yoon, Hu-ung Lee, Sooyong Kang, Youjip Won, and Jaehyuk Cha. 2012. Deduplication in SSDs: Mo del and quantitative analysis. In *IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST), 2012*. IEEE, 1–12.

[23] Ana Klimovic, Heiner Litz, and Christos Kozyrakis. 2017. Reflex: Remote flash local flash. *ACM SIGPLAN Notices* 52, 4 (2017), 345–359.

[24] Ricardo Koller and Raju Rangaswami. 2010. I/O deduplication: Utilizing content similarity to improve I/O performance. In *File and Storage Technologies (FAST), 8th Usenix Conference on*. Usenix.

[25] Dongup Kwon, Jaehyung Ahn, Dongju Chae, Mohammadamin Ajdari, Jaewon Lee, Suheon Bae, Youngsok Kim, and Jangwoo Kim. 2018. DCS-ctrl: a fast and flexible device-control mechanism for device-centric server architecture. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*. IEEE Press, 491–504.

[26] Sungjin Lee, Jihoon Park, Kermin Fleming, Jihong Kim, et al. 2011. Improving performance and lifetime of solid-state drives using hardware-accelerated compression. *IEEE Transactions on consumer electronics* 57, 4 (2011).

[27] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. 2017. Kv-direct: High-performance in-memory key-value store with programmable nic. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 137–152.

[28] Dongyang Li, Qing Yang, Qingbo Wang, Cyril Guyot, Ashwin Narasimha, Dejan Vucinic, and Zvonimir Bandic. 2015. A Parallel and Pipelined Architecture for Accelerating Fingerprint Computation in High Throughput Data Storages. In *FCCM*.

[29] Bin Lin, Shanshan Li, Xiangke Liao, Jing Zhang, and Xiaodong Liu. 2014. Leach: an automatic learning cache for inline primary deduplication system. *Frontiers of Computer Science* 8, 2 (2014), 175–183.

[30] Ming Liu, Simon Peter, Arvind Krishnamurthy, and P. Mangpo Phothilimthana. 2019. E3: Energy-Efficient Microservices on SmartNIC-Accelerated Servers.. In *USENIX annual technical conference*.

[31] PureStorage. 2019. PureStorage Purity Reduce. https://www.purestorage.com/products/purity/purity-reduce.html.

[32] Ran Xian and Runshen Zhu. 2016. Reposity of an open-source PALM tree. https://github.com/runshenzhu/palmtree.

[33] Jason Sewall, Jatin Chhugani, Changkyu Kim, Nadathur Satish, and Pradeep Dubey. 2011. PALM: Parallel architecture-friendly latch-free modifications to B+ trees on many-core processors. *Proc. VLDB Endowment* 4, 11 (2011), 795–806.

[34] SK Hynix. 2019. HBM Products. https://www.skhynix.com/chat/product/dramHBM.jsp.

[35] SmartIOPS. 2016. Flash Summit 2016 Product Video. http://www.smartiops.com/.

[36] SmartIOPS. Feb. 2018. World's Fastest SSDs. http://www.smartiops.com/worlds-fastest-ssds/.

[37] Solidfire. 2019. How Solidfire data efficiencies work. https://www.netapp.com/us/media/ds-solidfire-data-efficiencies-breif.pdf.

[38] Kiran Srinivasan, Timothy Bisson, Garth R Goodson, and Kaladhar Voruganti. 2012. iDedup: latency-aware, inline data deduplication for primary storage.. In *FAST*.

[39] Storage Networking Industry Association. IOTTA trace repository. 2008. FIU Traces. http://iotta.snia.org/.

[40] Vasily Tarasov, Deepak Jain, Geoff Kuenning, Sonam Mandal, Karthikeyani Palanisami, Philip Shilane, Sagar Trehan, and Erez Zadok. 2014. Dmdedup: Device mapper target for data deduplication. In *Ottawa Linux Symp.*

[41] Hung-Wei Tseng, Qianchen Zhao, Yuxiao Zhou, Mark Gahagan, and Steven Swanson. 2016. Morpheus: creating application objects efficiently for heterogeneous computing. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 53–65.

[42] Tung, Liam . 2018. Intel: This 'ruler' SSD is world's densest, so you can cram 1PB in single 1U rack. https://www.zdnet.com/article/intel-this-ruler-ssd-is-worlds-densest-so-you-can-cram-1pb-in-single-1u-rack/.

[43] Avani Wildani, Ethan L Miller, and Ohad Rodeh. 2013. Hands: A heuristically arranged non-backup inline deduplication system. In *ICDE*.

[44] Huijun Wu, Chen Wang, Yinjin Fu, Sherif Sakr, Kai Lu, and Liming Zhu. 2018. A differentiated caching mechanism to enable primary storage deduplication in clouds. *IEEE Transactions on Parallel and Distributed Systems* 29, 6 (2018), 1202–1216.

[45] Wen Xia, Hong Jiang, Dan Feng, Lei Tian, Min Fu, and Zhongtao Wang. 2012. P-dedupe: Exploiting parallelism in data deduplication system. In *Networking, Architecture and Storage (NAS), 2012 IEEE 7th International Conference on*. IEEE, 338–347.

[46] Xilinx. 2016. DMA Subsystem for PCI Express (Vivado 2016.3) - Performance Numbers. https://www.xilinx.com/support/answers/68049.html.

[47] Xilinx. March 2019. VCU1525 Reconfigurable Acceleration Platform. https://www.xilinx.com/support/documentation/boards_and_kits/vcu1525/ug1268-vcu1525-reconfig-accel-platform.pdf.

[48] Yi-Hua Edward Yang and Viktor K Prasanna. 2010. High throughput and large capacity pipelined dynamic search tree on FPGA. In *Proceedings of the 18th annual ACM/SIGDA international symposium on Field programmable gate arrays*. ACM, 83–92.

[49] Jie Zhang, David Donofrio, John Shalf, Mahmut T Kandemir, and Myoungsoo Jung. 2015. Nvmmu: A non-volatile memory management unit for heterogeneous gpu-ssd architectures. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 13–24.

[50] Benjamin Zhu, Kai Li, and R Hugo Patterson. 2008. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System.. In *Fast*, Vol. 8. 1–14.