SSDStreamer: Specializing I/O Stack for Large-Scale Machine Learning

Jonghyun Bae
Seoul National University

Hakbeom Jang
Sungkyunkwan University

Jeonghun Gong
Seoul National University

Wenjing Jin
Seoul National University

Shine Kim
Seoul National University

Jaeyoung Jang
Sungkyunkwan University

Tae Jun Ham
Seoul National University

Jinkyu Jeong
Sungkyunkwan University

Jae W. Lee
Seoul National University

***Abstract*—This article presents SSDStreamer, an SSD-based caching system for large-scale machine learning. By using DRAM as stream buffer, instead of an upper-level cache, SSDStreamer significantly outperforms state-of-the-art multilevel caching systems on Apache Spark, while requiring much less DRAM capacity.**

■ Iterative algorithms are at the core of many machine learning (ML) algorithms, which update estimates of the exact solution through an iterative process. Examples include optimization problems that search for the value of an argument to maximize or minimize a function. For these algorithms, in-memory caching is widely adopted, such as Spark MLlib[1] and Ignite ML,[2] to reuse the intermediate results by later computations and provide much higher throughput and lower latency than the

Digital Object Identifier 10.1109/MM.2019.2930497
Date of publication 23 July 2019; date of current version 10 September 2019.

conventional disk-based framework. DRAM device scaling is the key enabler of this technology, allowing hundreds of gigabytes of memory to be installed on a single node at an affordable cost. Thus, the success of this paradigm in the future counts heavily on the continuation of DRAM scaling.

However, DRAM scaling faces serious challenges, and SSDs are a promising alternative to substitute or augment DRAM-based caching. Figure 1 shows the scaling trends of DRAM and SSD bit cost over the past four decades we gathered from various public sources.[3,4] The figure shows a significant slowdown of DRAM cost reduction to be only 6% per year, while SSDs show much healthier cost scaling curve during the same period. Furthermore, an 8-lane PCIe Gen3 channel offers 6.4-GB/s peak bandwidth, which will soon double with the arrival of PCIe Gen4. This bandwidth will be comparable to that of DDR4-1600 DIMM, which was the mainstream around 2015.

This article presents SSDStreamer, a novel SSD-based caching system that retains the benefits of fast DRAM caching at a fraction of its cost, targeting large-scale ML frameworks.

A common approach to building an in-memory cache is to use DRAM as a primary cache backed by SSDs as a secondary cache. However, there is a serious performance drop when the working set outgrows the DRAM size as most requests are served by a slower SSD. Serialization adds an additional latency penalty to the slow disk access. Also, the heavyweight kernel I/O stack, both synchronous and asynchronous, incurs significant CPU overhead, penalizing both latency and throughput. Thus, an ideal caching system must provide both low average latency and high SSD bandwidth utilization with a small DRAM footprint.

This article presents SSDStreamer, a novel SSD-based caching system that retains the benefits of fast DRAM caching at a fraction of its cost, targeting large-scale ML frameworks. SSDStreamer uses DRAM as a *stream buffer* for coarse-grained prefetching from a large SSD cache built on top of a lightweight user-space I/O stack. The stream buffer generates a stream of coarse-grained prefetch requests to better utilize SSD bandwidth and hide its long latency as only the first request in a stream misses, while the subsequent ones hit via effective prefetching. Besides, SSD latency is further reduced by employing a custom serializer.

We successfully evaluate SSDStreamer on Apache Spark (MLlib),[1] a popular in-memory analytics framework using ML workloads from HiBench.[5] Spark on SSDStreamer with a 0.76-GB stream buffer outperforms the state-of-the-art multilevel cache with 192-GB DRAM by 37.8% whose working set is larger than available DRAM. If we assume the same capacity of 192 GB, SSDStreamer reduces the cost per gigabyte of the caching system by a factor of 30.2× over the DRAM-only cache.

Our contributions can be summarized as follows.

- We analyze the performance of multiple in-memory caching systems to identify major performance bottlenecks in disk-based caches.
- We propose SSDStreamer, an SSD-based caching system, which eliminates those bottlenecks by specializing the I/O stack for large-scale ML processing.
- We provide detailed evaluation of Spark on SSDStreamer to demonstrate superior performance over state-of-the-art multilevel caches, while using much less DRAM.

BACKGROUND AND Motivation

In-memory processing frameworks cache frequently reused data in memory to access them fast. This eliminates a large portion of the disk I/Os, thus, enabling high-throughput, low-latency ML processing. However, as the effective working set size outgrows memory capacity, performance is often bottlenecked by slow disk I/Os.

Figure 2 compares the per-task execution time of two ML workloads on Spark (NWeight and SVM) using three caching models. In Spark, memory-only (M) stores all intermediate data, called resilient distributed data sets (RDDs), to memory as long as there is free space. Memory-only discards cached data when memory is full and recomputes them when they are reused. As shown in Figure 2, memory-only performs very well when the

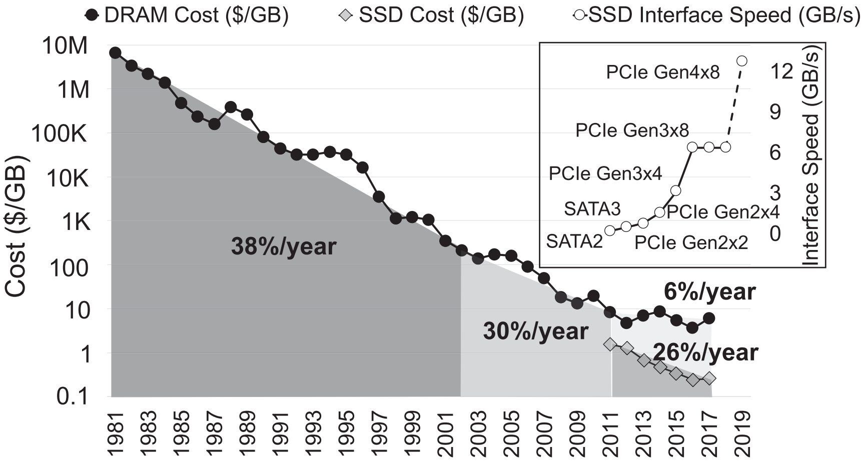

Figure 1. Scaling trends of DRAM and SSD.

working set fits in memory (labeled “Small” in Figure 2) but poorly when the working set outgrows memory size due to frequent recomputation (hence not shown in the figure).

Another caching model named memory-and-disk (M+D) caches data in memory, but when there is no free space, the data are stored (or migrated) to the disk instead. This model naturally has the same performance to memory-only for small data but can achieve reasonable performance for large data. Finally, disk-only (D) always caches data to the disk without using memory at all. While this has lower performance potential than the M+D (see SVM with large input in Figure 2), it can achieve better performance when the working set greatly exceeds the memory size (see NWeight with large input) due to excessive capacity misses at DRAM cache.

The message of Figure 2 is simple. Spark wastes most of the time on managing disk I/Os (i.e., disk accesses and serialization) when the working set is large, regardless of the choice of caching model. And with the trend of ever-increasing data size, this will become an even more serious problem in the future.

SSDStreamer DESIGN

Overview

SSDStreamer is a lightweight user-space I/O stack exposing a stream interface. Through the effective use of SSD and small DRAM as a stream buffer, SSDStreamer aims to retain the performance benefits of in-memory caching, while saving the cost of DRAM for large-scale ML processing. Our strategies to achieve this goal are as follows.

- *Do not cache but stream.* Stream buffers[6] are well suited for coarse-grained caching patterns of large-scale ML frameworks. By rightsizing the request granularity, we can utilize SSD bandwidth efficiently while minimizing the DRAM footprint.
- *Substitute the general-purpose kernel I/O stack with a specialized user-space I/O stack.* A lightweight user-space I/O stack allows us to reduce the average latency by avoiding the penalty of page cache management, context switching, and interrupt handling.
- *(De)serialize it fast.* Our analysis shows the cost of serialization in terms of CPU cycles can be significant. A customized serializer alleviates this problem.

Figure 3(a) overviews the SSDStreamer system stack, whose details are to be discussed in the rest of this section.

Stream Interface

To support data object streaming, we introduce a *stream* data type. Typically, a stream is

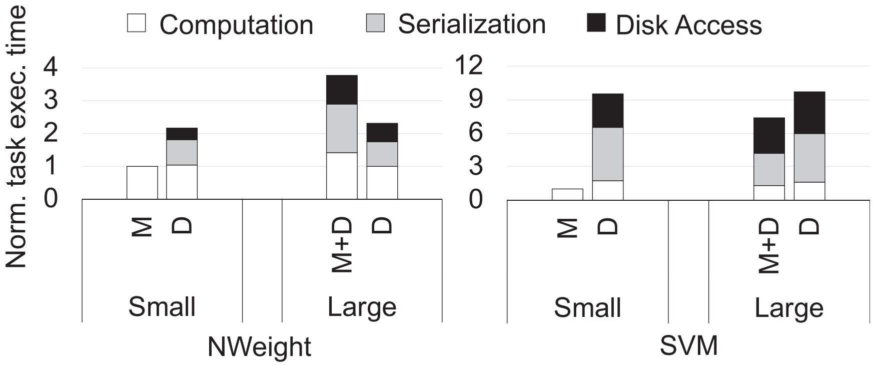

Figure 2. Task execution time breakdown for NWeight and SVM.

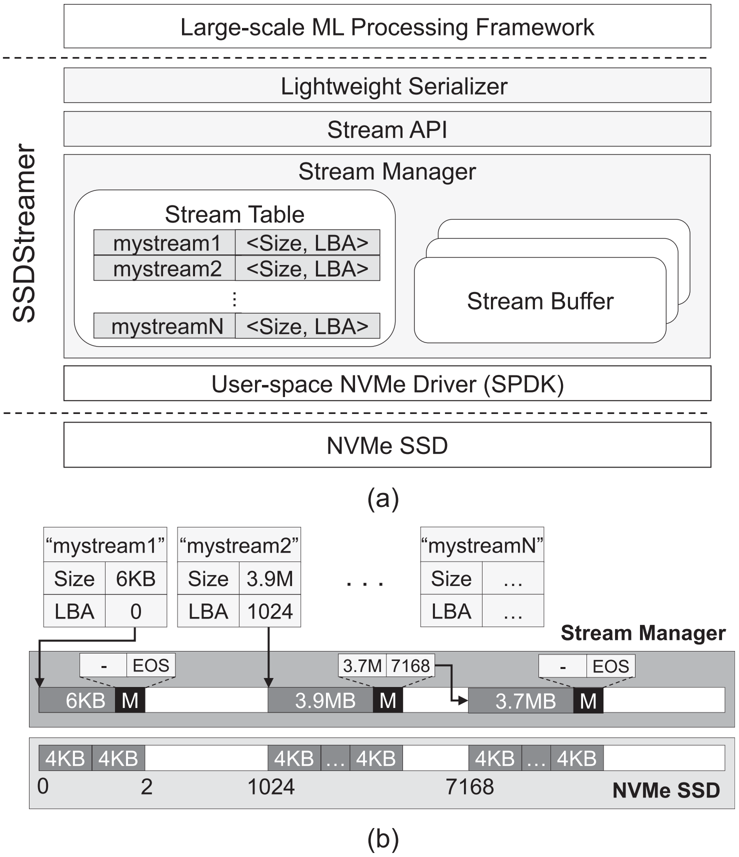

Figure 3. (a) SSDStreamer system stack. (b) Stream metadata structure and sblock chaining.

allocated to each cached object (e.g., RDD partition in Spark). A stream consists of a sequence of multiple, fine-grained *stream blocks*, or *sblocks*, which serve as the basic unit of caching. The concept of stream roughly corresponds to a file in the conventional file system but is optimized for a sequential access pattern with effective coarse-grained prefetching.

Stream API. SSDStreamer exposes a simple, easy-to-use API to control streams. Figure 4 shows a simple code example using Stream API to (a) create a stream and (b) read it back to display on stdout. Since we intentionally design the API in a similar style to the file API of a POSIX-compliant OS, most of the functions are self-explanatory [similar to open(), close(), read(), and write()].

Stream Manager

Stream manager is responsible for receiving requests for a stream and translating them to device read/write requests handled by a user-space NVMe SSD driver [i.e., Storage Performance Development Kit (SPDK)].[7] The stream manager object is created and attached to an NVMe SSD by calling init_stream_manager() [line 2 in Figure 4(a)]. The object is a global structure shared by all threads.

Managing Sblock metadata. Stream manager maintains per-sblock metadata, which consist of a 4B starting logical block address (LBA) and the *effective size* of the sblock (4B). The starting LBA is necessary because adjacent sblocks are not necessarily placed in contiguous LBA space although each sblock occupies a contiguous LBA space. In addition, the effective size of an sblock should be maintained as the actual amount of data within the sblock may be smaller than the predefined static sblock size (e.g., 4 MB is used for our evaluation). As shown in Figure 3(b), the stream manager maintains this metadata in memory (i.e., LBA and the size) only for the very first sblock of the stream. The lifetime of sblock metadata spans that of the stream itself [i.e., from create_stream() to destroy_stream()]. Metadata for other sblocks in the stream are attached to the end of the preceding sblock's data. This is possible because sblocks in a stream are accessed sequentially. We call this technique *sblock chaining*. Thus, only 8B metadata per stream needs to be maintained in memory instead of 8B metadata per every single sblock.

Managing DMA buffers. When a stream is created or opened for read, the stream manager allocates a *stream buffer* to the stream from the buffer pool. A stream buffer is a circular FIFO queue, which is sized to house multiple sblocks. The number of sblocks contained in a stream buffer is equivalent to the prefetch depth. During stream creation, this buffer is a place where data passed by a write_stream() call are buffered before it gets flushed to the disk. During stream read, a stream buffer houses prefetched sblocks. When the stream is closed, this buffer is freed. The stream buffer is necessary only for the duration of stream creation or read operation itself.

Creating a stream. When create_stream() is called which returns a handle (sd) for the stream [line 3 in Figure 4(a)], followed by the first write [write_stream() in Line 8], a set of contiguous logical blocks in the SSD are retrieved from the free block pool. Also, its starting LBA and the first sblock's effective size are logged to the in-memory metadata of the stream manager. In addition, a stream buffer is allocated for this stream, and the payload data for current write_stream() request are buffered in the stream buffer. When the next block of data is written to this stream (with another write_stream() call), another set of contiguous logical blocks are retrieved from the free block pool, and its starting LBA and effective sblock size are appended to the previous sblock's data. Then, the previous sblock's data (appended with current sblock's metadata) are flushed to the disk. At the same time, the data for the current sblock are buffered in the stream buffer. When the stream is closed by close_stream() [line 10 in Figure 4(a)], an end-of-stream token is appended to the current sblock's data and flushed to the disk.

Reading a stream with stream buffer. To read data from a stream, open_stream() is invoked [line 1 in Figure 4(b)]. Then, the stream manager allocates a stream buffer to this stream and retrieves the starting LBA and the size of the first sblock from the stream table. With this metadata, the stream manager starts reading the very first sblock of this stream from the disk and places it to the corresponding stream buffer. Once it finishes reading the sblock, it reads the LBA and effective size for the next sblock, which are appended to the current sblock's data, and initiates disk access. This process is continued as long as there is free space in the stream buffer. When read_stream () is called [line 5 in Figure 4(b)], the stream manager returns the oldest sblock from the corresponding stream buffer and frees the space this sblock was occupying. In a steady state this initiates another disk read for the next sblock that needs to be fetched. With this mechanism read_stream() typically returns immediately because the requested data are already buffered in the stream buffer. Essentially, this mechanism enables disk accesses to overlap with computation and can completely hide disk access latency when the computation is sufficiently long.

```
 1 size_t sblock_size = 1<<22    // 4MB
 2 void *manager = init_stream_manager(sblock_size, nvme_id);
 3 int sd = create_stream(manager, "mystream");
 4 void *large_array;
 5
 6 for(i = 0; i < array_size; i += sblock_size) {
 7   compute(large_array + i, …)   // User computation
 8   write_stream(sd, large_array + i, sblock_size);
 9 }
10 close_stream(manager, sd);
```

(a)

```
 1 int sd = open_stream(manager, "mystream");
 2 size_t sblock_size;
 3 void *buffer;
 4
 5 while((buffer = read_stream(sd, sblock_size)) {
 6   compute(buffer, …);                // User computation
 7   write(1, buffer, sblock_size);   // Write to stdout
 8 }
 9 close_stream(manager, sd);
10 destroy_stream(manager, "mystream");
```

(b)

Figure 4. Stream API usage example. (a) Creating a stream. (b) Reading a stream.

Destroying a stream. A stream is destroyed by calling destroy_stream() [line 10 in Figure 4(b)]. Then, the stream manager first allocates a temporary 4-KB buffer to hold the data stored in the last LBA of the sblock containing the metadata. A stream manager computes the last LBA of the first sblock using the metadata in the stream table [i.e., starting LBA+ceil(effective sblock size/logical block size)-1] and reads it into the 4-KB buffer. Then, the stream manager returns the list of LBAs allocated to the first sblock to the free list. By inspecting the last LBA of the sblock just freed, the stream manager can retrieve the starting LBA of the next sblock and free the list of LBAs allocated to the sblock. This process continues until an end-of-stream token is encountered. Stream API also provides read_and_destroy_-stream(), which performs destroy_stream() while reading out data from the stream, which is especially useful with a static data access pattern that can pinpoint precisely to the last use of the stream.

Lightweight Serializer

To store an object in an object-oriented language to the disk, the system must *serialize* it into a sequence of bytes. For this purpose a serializer needs to copy its fields to the serialized byte stream. Besides, if the object has a reference, the referenced object should be serialized and included in the serialized byte stream as well. As shown in Figure 2, (de)serialization accounts for a substantial portion of task execution time for the M+D cache.

To reduce this overhead, we *specialize* the Kryo serializer[8] used in Spark and make it more lightweight. In particular, we apply the following two major changes. First, whenever possible, the serializer replaces slow byte-by-byte copy with fast object-granularity copy. In the context of caching intermediate data (like RDDs in Spark), a serialized object is always produced and consumed in the same execution context. This relaxes some of the portability concerns such as byte ordering, memory layout, and data structure representations. Thus, strict byte-by-byte copy to preserve compatibility across different platforms is unnecessary.

Second, SSDStreamer progressively reads and writes a serialized object (stream) to the disk at a fixed-size granularity of sblock. The original Kryo serializer initially allocates 4-KB buffers for (de) serialization, and it can grow in size unboundedly depending on the actual object size. This results in an additional heap allocation and copy whenever the buffer is full. In contrast, the specialized serializer writes the contents of the serialized data directly to the off-heap stream buffer to eliminate this overhead.

EVALUATION

Methodology

System configurations. We use Spark 2.1.2[1] on five Dell R730 nodes with one master and four workers. Each node consists of two Intel Xeon CPU E5-2640v3, two 1TB SAS SSDs for HDFS, and two 1.6-TB Samsung PM1725 NVMe SSDs for caching. All nodes are connected by 40-Gb/s Infiniband.

Caching models. SSDStreamer (SST) is compared with five models, including the M+D and disk-only (D) discussed in the “Background and Motivation” section.

- Alluxio (ALX): Alluxio[9] offers a tiered storage including off-heap DRAM and SSDs.
- Disk-only+Prefetching (PREF): This model uses the asynchronous kernel I/O library (libaio) to quantify the benefits of prefetching, but with no other optimizations.
- Disk-only+UserIO (USERIO): This model uses the user-space NVMe driver (SPDK) to quantify the benefits of the lightweight I/O stack, but with no other optimizations.

M+D and ALX run with 256-GB DRAM per node: 64 GB for execution and 192 GB for caching. D, PREP, USERIO, and SST use one-fourth of the DRAM capacity per node (64 GB). In this setup, we allocate 60 GB for execution, and the remaining is allocated for page cache (4 GB in D), prefetching (4 GB in PREP), buffers for user-space I/O (0.38 GB in USERIO), and stream buffers (0.76 GB in SST). We have carefully tuned the partition size to minimize GC overhead as done in the article by Bae *et al.*[10]

Benchmarks. We use 11 ML workloads from the latest Intel HiBench 7.0.[5] Although the benchmark suite has 14 ML applications, three of them are excluded as they do not use the cache. We classify the applications into two categories. Six applications spend a dominant portion of total execution time computing on cached data and are classified as *cache-intensive*. The other five are classified as *cache-light*. Note that the working set of cache-light workloads fits in memory, whereas that of cache-intensive workloads do not.

Performance

Overall. Figure 5(a) shows the normalized execution time. SSDStreamer (SST) performs the best among all the caching models being considered. SST reduces the total execution time by 33.2% and 37.8% compared to the M+D and Alluxio (ALX), which represent the state-of-the-art multilevel caching organizations. Once the working set outgrows the DRAM capacity, the multilevel caching performs poorly due to a long storage access time and the promotion/demotion time of cached RDD partitions. SSDStreamer also outperforms the disk-only (D), disk-only+Prefetching (PREF), and disk-only+UserIO (USERIO) by 25.5%, 20.9%, and 18.2%, respectively.

Execution time breakdown. Figure 5(b) breaks down the task execution time into three

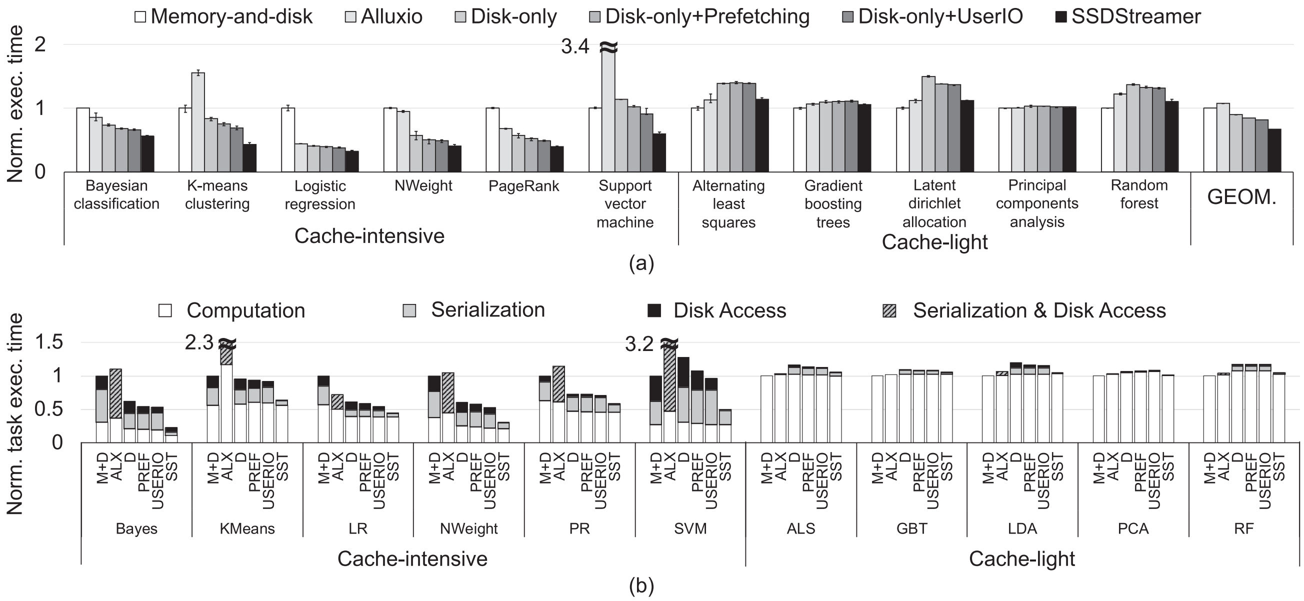

Figure 5. (a) Normalized execution time. (b) Normalized task execution time breakdown.

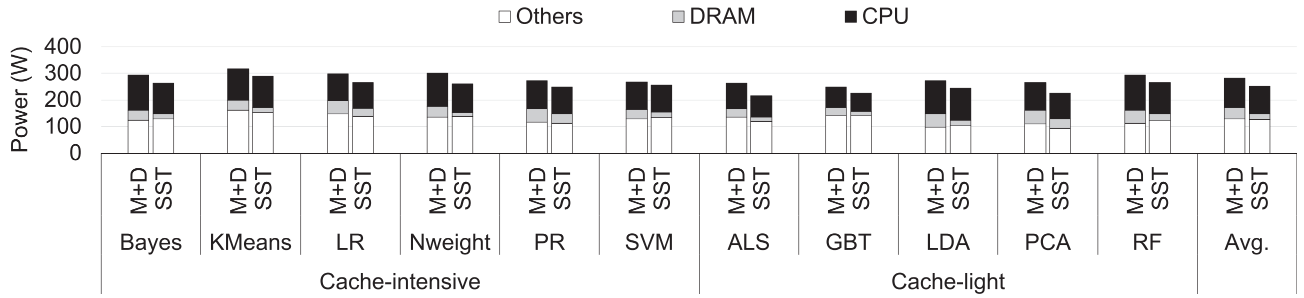

Figure 6. Per-node power breakdown of memory-and-disk (M+D) and SSDStreamer (SST).

portions: computation (in white), serialization (in gray), and disk access (in black).

SST effectively reduces the overhead of disk access by 82.3% on average compared to D. By comparing the disk access time between USERIO and SST, we can quantify the effectiveness of prefetching in the user-space I/O stack. The performance impact of employing a user-space I/O stack is captured by the difference of the disk access time between PREF to SST. SST reduces the overhead of disk access by 72.2% and 76.4% on average compared to USERIO and PREF, respectively.

Finally, the proposed lightweight serializer shows a significant reduction in SST by 59.6% on average compared to D (similar reduction compared to PREF and USERIO). To summarize, the performance gains for SST are attributed to effective overlap of computation and disk I/O using stream buffers and lightweight (de)serialization.

For the cache-light applications, SSDStreamer slightly increases the total execution time by 7.4% compared to M+D. This is attributed to the smaller cache footprint, allowing more execution memory to be opportunistically allocated to JVM heap in M+D (using 256 GB per node) than SSDStreamer, which only uses 60 GB per node for execution.

Power Consumption

To quantify the advantage of SSDStreamer in power efficiency, we measure the node-level power consumption using Dell's Remote Access Controller (iDRAC8). At the same time, we measure CPU and DRAM power consumption using Intel's Running Average Power Limit.

Figure 6 shows the breakdown of power consumption for the M+D and SSDStreamer (SST). SST reduces the node-level power consumption by 10.9% on average compared to M+D. DRAM and CPU power consumption decreased by 47.4% and 7.2%, respectively. SST has only one-fourth of the DRAM compared to M+D to significantly reduce DRAM power consumption. In addition, the lightweight I/O stack bypasses the heavyweight kernel I/O to reduce CPU utilization (and hence power).

SSDStreamer is an SSD-based caching system specialized for large-scale ML processing. To achieve both low latency and high bandwidth with a minimal DRAM footprint, SSDStreamer uses DRAM as stream buffers for a large SSD cache managed in a user space.

CONCLUSION

SSDStreamer is an SSD-based caching system specialized for large-scale ML processing. To achieve both low latency and high bandwidth with a minimal DRAM footprint, SSDStreamer uses DRAM as stream buffers for a large SSD cache managed in a user space. SSDStreamer also introduces a lightweight serializer to significantly reduce the (de)serialization latency. The resulting design delivers competitive performance to a large DRAM cache at a fraction of its cost. SSDStreamer is the first to successfully demonstrate the feasibility of replacing the DRAM cache with a cost-effective SSD cache for large-scale ML processing.

ACKNOWLEDGMENTS

This work was supported in part by the Research Resettlement Fund for the new faculty of Seoul National University; in part by a research grant from Samsung Electronics, National Research Foundation of Korea grant funded by the Ministry of Science, ICT & Future Planning (PE Class Heterogeneous High Performance Computer Development, NRF-2016M3C4A7952587); and in part by the Competency Development Program for Industry Specialists of the Korean Ministry of Trade, Industry and Energy (MOTIE), operated by the Korea Institute for Advancement of Technology (KIAT) (No. N0001883, HRD Program for Intelligent Semiconductor Industry).

REFERENCES

1. "Apache Spark," 2017. [Online]. Available: https://spark.apache.org/
2. "Apache Ignite," 2019. [Online]. Available: https://ignite.apache.org/.
3. "Newegg.com," 2017. [Online]. Available: https://www.newegg.com/
4. M. John C., "Price and Performance Changes of Computer Technology with Time," 2017. [Online]. Available: http://www.jcmit.net/
5. "Intel HiBench," 2018. [Online]. Available: https://github.com/Intel-bigdata/HiBench/
6. N. P. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," in *Proc. 17th Int. Symp. Comput. Architecture*, 1990, pp. 364–373.
7. "Storage Performance Development Kit," 2018. [Online]. Available: http://www.spdk.io/
8. "Kryo Serializer," 2018. [Online]. Available: https://github.com/EsotericSoftware/kryo/
9. "Alluxio," 2018. [Online]. Available: https://www.alluxio.org/
10. J. Bae *et al.*, "Jointly optimizing task granularity and concurrency for in-memory mapreduce frameworks," in *Proc. IEEE Int. Conf. Big Data*, 2017, pp. 130–140.

Jonghyun Bae is currently working toward a PhD in computer science and engineering at Seoul National University. His research areas include big data processing and distributed systems. Contact him at jonghbae@snu.ac.kr.

Hakbeom Jang has a PhD in electrical and computer engineering from Sungkyunkwan University, Seoul. His research interests include computer architecture, parallel programming, and big data processing. Contact him at hakbeom@skku.edu.

Jeonghun Gong is currently working toward an MS in computer science and engineering at Seoul National University. His current research focuses on hardware accelerator design. Contact him at jh.gong@snu.ac.kr.

Wenjing Jin is currently working toward a PhD in computer science and engineering at Seoul National University. Her research interests include operating systems and accelerator architectures. Contact her at wenjing.jin@snu.ac.kr.

Shine Kim is currently working toward a PhD in computer science and engineering at Seoul National University. His research areas include computer architecture and storage systems. Contact him at postshine@snu.ac.kr.

Jaeyoung Jang is currently working toward a PhD in electrical and computer engineering at Sungkyunkwan University, Seoul. His research interests include computer architecture and big data processing. Contact him at jaey86@skku.edu.

Tae Jun Ham is currently a postdoctoral researcher with Seoul National University. He has a PhD inelectrical engineering from Princeton University. His current research focuses on hardware–software codesign for emerging applications and computing systems. Contact him at taejunham@snu.ac.kr.

Jinkyu Jeong is currently an assistant professor with the College of Information and Communication Engineering, Sungkyunkwan University, Seoul. He has a PhD in computer science from the Korea Advanced Institute of Science and Technology (KAIST). His research interests include real-time systems, operating systems, virtualization, and embedded systems. Contact him at jinkyu@skku.edu.

Jae W. Lee is currently an associate professor with the Department of Computer Science and Engineering, Seoul National University. He has a PhD in computer science from the Massachusetts Institute of Technology (MIT). His research areas include computer architecture, VLSI design, parallel programming, and computer security. He is the corresponding author. Contact him at jaewlee@snu.ac.kr.