BigCache for Big-data Systems

Michel Angelo Roger, Yiqi Xu, Ming Zhao
Florida International University, Miami, Florida
{mroge037,yxu006,mzhao}@fiu.edu

***Abstract*— Big-data systems are increasingly used in many disciplines for important tasks such as knowledge discovery and decision making by processing large volumes of data. Big-data systems rely on hard-disk drive (HDD) based storage to provide the necessary capacity. However, as big-data applications grow rapidly more diverse and demanding, HDD storage becomes insufficient to satisfy their performance requirements. Emerging solid-state drives (SSDs) promise great IO performance that can be exploited by big-data applications, but they still face serious limitations in capacity, cost, and endurance and therefore must be strategically incorporated into big-data systems. This paper presents BigCache, an SSD-based distributed caching layer for big-data systems. It is designed to be seamlessly integrated with existing big-data systems and transparently accelerate IOs for diverse big-data applications. The management of the distributed SSD caches in BigCache is coordinated with the job management of big-data systems in order to support cache-locality-driven job scheduling. BigCache is prototyped in Hadoop to provide caching upon HDFS for MapReduce applications. It is evaluated using typical MapReduce applications, and the results show that BigCache reduces the runtime of WordCount by 38% and the runtime of TeraSort by 52%. The results also show that BigCache is able to achieve significant speedup by caching only partial input for the benchmarks, owing to its ability to cache partial input and its replacement policy that recognizes application access patterns.**

I. INTRODUCTION

Big data is an important computing paradigm that becomes increasingly used by many science, engineering, medical, and business disciplines for knowledge discovery, decision making, and other data-driven tasks based on processing and analyzing large volumes of data. To support such applications, big-data systems are typically built upon programming frameworks that can effectively express data parallelism and exploit data locality (e.g., MapReduce [1]) and storage systems that can provide high scalability and availability (e.g., Google File System [2], Hadoop HDFS [3]). A variety of higher-level data services (e.g., BigTable [4], HBase [5], GraphLab [6]) can be further built upon such frameworks.

Big-data systems have been relying on traditional hard-disk drive (HDD) based storage to provide the *volume* for storing large amounts of data required by big-data applications. They are optimized for large, sequential IOs which HDDs excel at [2]. However, modern big-data applications have rapidly growing *velocity*, the speed of storing and processing data, and *variety*, the types of data and their processing methods. HDDs alone become insufficient to satisfy the challenging demands of such big-data applications.

The emerging solid-state drive (SSD) based storage offers excellent IO performance that is substantially better than HDDs and can be employed to meet the performance requirements of big-data applications. However SSDs still face several serious limitations compared to HDDs. First, SSDs come with much smaller capacity and are much more expensive per unit size, which makes it difficult to provision enough volumes for big-data storage at a reasonable cost. Second, SSDs wear out by writes, which raises concerns about the durability of data as well as the additional cost for hardware maintenance. Therefore SSD-based storage needs to be strategically incorporated in existing big-data systems instead of completely replacing HDDs.

Recognizing the above mentioned constraints of HDDs and SSDs, the central research problem studied by this paper is how to effectively incorporate both types of storage devices into big-data systems in order to satisfy the volume, velocity, and variety requirements from diverse big-data applications. Our solution is BigCache, an SSD-based distributed caching layer that allows seamless integration of SSDs with existing HDD-based big-data systems and enables transparent acceleration of the different types of application data accesses, thereby exploiting both the performance of SSDs and the capacity of HDDs for big-data applications.

Although caching is a classic technique commonly used in computer systems design, BigCache is architected to address the unique challenges to providing caching for big-data systems. *First*, to support highly distributed big-data applications, BigCache is designed as a distributed SSD-based caching layer with distributed cache management across the networked datanodes in a big-data system. *Second*, to make effective use of the data cached in SSDs, the management of BigCache is coordinated with the job management of the big-data system in order to support locality-driven job scheduling which is critical to the performance of big-data applications. *Third*, to support a wide variety of big-data applications including legacy code, the integration of BigCache in a big-data system is completely transparent to the applications, without requiring any change to the existing APIs that the applications are familiar with.

A prototype of BigCache is created on Hadoop for MapReduce applications. Experiments based on representative benchmarks show that the BigCache can substantially improve the performance for applications with large input such as WordCount and applications with both large input and output such as TeraSort. The overhead of BigCache is unnoticeable as even with cold caches and buffering disabled, BigCache still achieves up to 14% runtime reduction. When the caches are warm, the runtime reduction increases to up to 27%. When the buffering is enabled in BigCache, the improvement grows even higher, especially for an application such as TeraSort that has intensive output. With both warm caches and buffering, the total improvement for WordCount is 38% runtime reduction and for

TeraSort is 53%. The results also show that BigCache delivers speedups even when the application's working set cannot fit entirely in the caches. The ability of caching partial input in BigCache is valuable for large big-data applications and consolidated big-data systems.

To the best of our knowledge, BigCache is a first SSD-based caching and buffering solution for big-data systems. Related work [7] has considered main-memory-based caching for MapReduce applications, which has only limited capacity to cache small applications and does not consider the opportunity of caching partial input. Related study [8] has also accessed the benefits of employing SSDs in big-data systems for storing temporary and permanent data, but it does not consider the limitations of SSDs which prevent them from replacing HDDs entirely. In comparison, BigCache is the first integrated solution that enables big-data applications to transparently benefit from the performance of SSDs and the capacity of HDDs.

The rest of the paper is organized as follows. Section II introduces the background and related work. Section III presents the design and implementation of BigCache. Section IV presents the experimental evaluation. Section V concludes the paper.

II. Background and Related Work

A. Big-data Systems

Typical big-data systems are built upon a highly scalable and available distributed storage system. For example, Google File System (GFS) [2] and its open-source clone Hadoop Distributed File System (HDFS) [3] provide fault tolerance while storing massive amounts of data on a large number of *datanodes* built with inexpensive commodity hardware, where MapReduce [1] applications are executed in a data-parallel fashion on the datanodes where their data is stored. Higher-level data services such as databases and data mining tools (e.g., [4][5][6]) can also be built upon such a big-data computing framework.

Both the map and reduce phases of a MapReduce application can spawn large numbers of map and reduce tasks, depending on the size of the input, on the datanodes of a big-data system to process data in parallel. Data stored on the distributed file system (DFS), e.g., GFS/HDFS, is typically replicated at least three times across the datanodes to tolerate failures. MapReduce applications often have complex but well-defined IO phases. A map task is always preferably scheduled to the nodes where its input data is stored, unless there is no *slot*—the unit used for CPU allocation, which typically maps to a CPU core or a portion of it—available on these nodes. It reads the input from the HDD storage through GFS/HDFS, either via the local file system if the input is local or across the network otherwise. The output of the map task, which is part of the intermediate result of the application, is first buffered in memory and then spilled onto the HDD storage directly via the local file system. A reduce task starts by copying and shuffling its input from all the map tasks' intermediate results, either stored locally from the map tasks on the same host or across the network from the remote map tasks. It then merges the copied inputs, performs the reduce processing, and generates the final output, which is stored and replicated on the HDD storage through GFS/HDFS.

B. Caching for Big-data Systems

Related work on PACMan [7] has studied the use of main-memory-based caching of map task inputs in order to improve application performance and cluster efficiency. There are several key differences in the approach taken by BigCache. *First*, BigCache is based on SSD devices which have much larger capacity and lower cost than main memory. At the same time, the limited main memory is often heavily used by big-data applications, e.g., for a map task to buffer the intermediate result and for a reduce task to buffer the shuffling data. Therefore, BigCache can provide more effective caching and buffering for large big-data applications and for highly consolidated big-data systems. *Second*, PACMan assumes an All-of-Nothing cache management property, i.e., a MapReduce application cannot achieve any performance improvement from caching unless its input is completely cached. This property applies to only small applications that can finish with a single wave of map tasks, but does not hold for large applications. In contrast, BigCache supports the caching of partial input while still delivering speedups to applications, as demonstrated in our experimental results (Section 4). Therefore, BigCache offers more opportunities for diverse big-data applications to improve performance from the use of caching. *Third*, PACMan does not consider buffering, since native MapReduce already uses main memory for buffering. BigCache provides big-data applications with both caching and buffering using SSDs and both are shown to be important to application performance.

The potentials of SSD-based caching has motivated several related solutions for distributed storage systems in general. For example, dm-cache [9] supports SSD-based caching for a block-level distributed storage system such as the virtual machine (VM) storage commonly used in cloud computing systems. Mercury [10] provides a block-level SSD cache in the hypervisor of a storage client, in order to provide caching to the VMs hosted on the client over a variety of networked storage protocols. Different from these solutions, BigCache provides SSD caching at a higher level, e.g., HDFS, of the storage hierarchy of a big-data system. It is therefore able to exploit useful semantics, e.g., HDFS files, specific to the applications and distributed file system to make better cache management decisions, e.g., per-application cache allocation and per-HDFS-file cache replacement, for the sake of application performance and resource utilization.

The related studies [11][12] also explored various design choices, including write policy, persistency, consistency, and flash-RAM integration, for SSD caching employed by cloud systems. BigCache learns from the insights of these studies and chooses designs that are specifically optimized for big-data applications, which are discussed in detail in the next section.

III. Design and Implementation

A. Architecture

To address the performance limitations of HDDs and the capacity and endurance limitations of SSDs, the approach taken by BigCache is to incorporate SSDs into the existing HDD-based big-data storage architecture as a caching layer, in order to exploit the limited capacity of SSDs to hold only the working sets of big-data applications and transparently accelerate the IOs for their various phases. *First*, when an application's map tasks read the input from GFS/HDFS, the requested blocks are cached by BigCache, so that when the application runs again the map tasks can be accelerated from the cached input. Note that a MapReduce application often processes the same dataset when

it executes repeatedly, although the knowledge that it tries to discover or the algorithm for the knowledge discovery may differ across runs. For example, a data mining application that tries to discover a user's interests from a particular social network (e.g., Twitter) often processes the same set of data (e.g., tweets), which may evolve slowly, while the user of interest may change and the algorithm for mining may also change.

Second, when the application's map tasks generate their intermediate results, they are buffered by BigCache if they are spilled out of the main memory, to speed up the operations of map output. *Third*, when the application's reduce tasks start to copy and shuffle the intermediate results from the map tasks, they can make faster progress when they find the map output in BigCache, instead of from the HDD storage, no matter whether the reduce task is getting the map output from the same host or across the network. *Fourth*, when the data that a reduce task has retrieved is spilled out of the main memory, it is buffered by BigCache to speed up the operations of shuffling.

Finally, the results generated by the reduce tasks, which are also the final output of the MapReduce application, are cached by BigCache, while being stored and replicated on the HDD storage. The caching of application output is useful when there is a chain of MapReduce applications where one's output is the input of the next. For example, a query to a data warehouse (e.g., [13]) can be processed as a series of MapReduce jobs with data interdependencies among their inputs and outputs. For such a workload, caching the output of one MapReduce application can speed up the input of the next one. However, BigCache does not delay the final output in the SSD caches, due to the concern for data reliability, so that the output is always stored reliably on the HDDs using the replication scheme typically employed by the big-data system. In contrast, the intermediate results buffered in the SSD caches will be cleaned up at the end of the application execution, in the same way how intermediate results stored on the HDDs are cleaned up by the native big-data system.

B. Cache Management

An important decision that the cache manager on each datanode needs to make is how to replace cached blocks when the cache size is not sufficient to hold the application's entire working set. Considering that when a map task reads its assigned range of input, it always reads it sequentially, we choose Most Recent Used (MRU) as the cache replacement policy instead of the commonly used Least Recently Used (LRU). When the map task reads a sequential sequence of blocks, LRU always evicts the earliest accessed blocks, if the entire sequence does not fit the cache. Consequently, when the same map task runs again, it cannot generate any hit in the cache as it repeats the sequence from the beginning. In contrast, for the same situation, MRU keeps the earliest accessed blocks in the cache which will be all hits when the map task repeats the sequence of block accesses. The portion of the sequence of blocks that does not fit the cache will be retrieved from the underlying HDD storage, which can be either local or remote.

Similarly to the above discussion, BigCache also employs MRU to buffer the output of a map task—the intermediate result—which is a sequential sequence of block writes, so that when the SSD gets full, the remaining map output will be directly spilled onto the HDD storage. On the contrary, if BigCache uses the LRU policy, the new block writes will evict the earlier ones from the SSD and cause unnecessary writes to both the SSD and the HDD, because the intermediate result does not really have to be stored on the HDD. These unnecessary writes are detrimental to both the performance of the application and the endurance of the SSDs used for buffering.

If there are multiple big-data applications running on the system concurrently, they also need to share the capacity of the SSD caches. BigCache supports proportional sharing of the cache capacity, where each application gets a certain share from each datanode's SSD cache. With each application's share, the capacity is equally divided for caching and buffing in our current BigCache implementation. The sharing ratio can be determined by system administrator based on the knowledge of applications' data size and priority. More intelligent cache and buffer partitioning and automatic cache allocation methods will be studied in our future work on BigCache.

C. Cache-locality-driven Job Scheduling

Locality-driven job scheduling is a fundamental principle of big-data systems, and a key differentiator from traditional distributed computing systems. Traditional systems typically adopt a remote data access model where data is shipped from the remote storage to the compute nodes where the application's tasks run. Big-data systems take a completely different model which ships computing to data by scheduling an application's tasks to the datanodes where their data is stored. Such data-locality-driven job scheduling avoids the expensive transfers of large volumes of data required by big-data applications, and is key to the success of big-data computing systems.

By employing SSD-based caching, BigCache introduces another level of locality into a big-data system, as a task may have its data local in the SSD cache of the node that it is scheduled to, independently from whether it has the data local in the HDD storage of the node. However, if a job scheduler (e.g., JobTracker in Hadoop) is unware of the existence of cache locality, it will schedule an application's tasks only based on the data locality on the HDD storage. But because data is commonly replicated across different datanodes' (at least three for triple modular redundancy) HDD storage, the exact nodes that a job scheduler chooses to run the application may differ across different runs. Consequently, the nodes that have cached data for the application—warmed up during the previous runs of the application—may not be the same nodes used for the next run. Therefore, the use of caching may become useless (while still paying for the performance and endurance costs) and the opportunity of accelerating the application by using the cached data may be wasted.

To address the above issue, BigCache enables new cache-locality-driven job scheduling by coordinating distributed cache management with centralized job management typically used by big-data systems. As data gets inserted into the SSD caches, the cache managers on the datanodes will periodically communicate with the centralized job scheduler to report the list of data blocks that are cached locally. Based on the knowledge about the cached blocks on all the datanodes, when the job scheduler decides the placement of an application's task, it will preferably consider the nodes that have the task's data in their local caches, before considering the nodes that have the data in their local HDD storage. In this way, the cached data can be effectively reused to improve the performance of big-data applications.

Table 1. IO demands of WordCount and TeraSort

Benchmark	Map input	Reduce input	Map spill & Reduce spill	Reduce output
WordCount	28GB	40GB	71GB	248MB
TeraSort	28GB	84GB	112GB	28GB

D. Hadoop-based Prototype

In our BigCache prototype, the SSD caching and cache management are fully integrated into Hadoop in order to assure low overhead of the implementation. At the same time, the modifications made to Hadoop do not require any change to the existing MapReduce API and are hence entirely transparent to MapReduce applications. Similarly to the use of HDD storage in Hadoop, the SSD storage used for caching is employed through a mounted local file system on the SSD. The path to the cache is specified as an additional entry in the standard Hadoop configuration file hdfs-site.xml.

The BigCache Controller class is the core component of our SSD caching layer. It runs on every datanode to manage the IO flow between the Hadoop DataXceiver class which issues HDFS IOs and the SSD cache and HDD storage. It also manages the cache allocation and replacement based on the chosen polices. The Controller caches data at the granularity of HDFS file chunks (by default 64MB per chunk), the unit used by JobTracker for assigning the range of input to be processed by a map task. Therefore, even though DataXceiver works at the granularity of smaller data blocks (e.g., 512KB), the BigCache Controller will always cache a chunk entirely because the map task will always process it entirely. Per-chunk caching allows BigCache to cache partial HDFS files and saves the overhead from managing large numbers of smaller blocks.

The BigCache Controller also periodically reports the list of locally cached data blocks to the Hadoop NameNode through remote procedure calls (RPCs), if there is any change to the list since the last report. The NameNode is a centralized component in Hadoop which keeps track of the data distribution on the HDDs, and is used by the Hadoop JobTracker to decide job scheduling. The frequency of report is chosen to be the same as the native heartbeat messages (by default every 3 seconds) between each datanode and the JobTracker, which are used by Hadoop to update node status and slot availability. This frequency can be tuned by the system administrator in order to make the overhead affordable for the size of the system.

When the native JobTracker considers the scheduling of a map task, it will first consider *node locality*, i.e., the nodes that have the required data on their local HDD storage, then *rack locality*, i.e., the nodes that are on the same rack of those with local data, and finally the other nodes without any locality in the system. Note that JobTracker may not always be able to schedule a map task to the nodes with locality, because the slot availability is dynamic as tasks come and go and at the time of the scheduling the nodes with locality may happen to have no slot available. But by following the aforementioned locality preferences to schedule the map task, JobTracker first tries to eliminate any data transfer and then tries to eliminate cross-rack data transfer. BigCache introduces another level of locality, *cache locality*, into the system, and the new cache-locality-aware JobTracker will give the nodes that have the required data in their SSD caches the highest preference, before considering the other nodes with lower levels of locality. This cache-locality-driven job scheduling ensures the reuse of cached data.

IV. Evaluation

A. Testbed and Benchmarks

This section presents an experimental evaluation of BigCache on a typical Hadoop setup using representative MapReduce benchmarks. The experiments were conducted on a cluster of eight nodes, each with two six-core 2.4GHz AMD Opteron CPUs, 32GB of RAM, one 120GB SSD, and two 500GB 7.2K RPM SAS HDDs, interconnected by a Gigabit Ethernet switch. All the nodes run the Debian 4.3.5-4 Linux with the 3.2.20-amd64 kernel. The HDDs use EXT3 as the local file system, and the SSDs use EXT4 as the local file system. Seven of these nodes are used as Hadoop datanodes and the other is used run the Hadoop JobTracker and NameNode.

The experiments consider two commonly used MapReduce benchmarks, WordCount and TeraSort, which have different data access patterns. We used similar input size for both benchmarks. The input for WordCount was downloaded from the United State patent website [14] and replicated multiple times to reach the 28GB of size. The input for TeraSort was created using TeraGen. Despite of the same input size and the same setup for execution, the two benchmarks differ significantly in their IO demands. Table 1 summarizes the total IO sizes of the various IO phases of these two benchmarks when they run on native Hadoop with HDD-based storage. TeraSort produces much more intermediate results—the map output, as reflected by the amount of data read by the reduce tasks and the amount of data spilled out of main memory by the map tasks and the reduce tasks. Note that the map tasks first buffer their output in memory and then spill it out to HDD storage. The reduce tasks read all the map output, which is also first buffered in memory and then spilled out to HDD storage.

All the experimental results reported in this section are from multiple runs of the benchmarks and both the averages and standard deviations are reported. Each run of TeraSort involves 420 map tasks and 14 reduce tasks; and each run of WordCount involves 424 map tasks and 14 reduce tasks.

B. Caching and Buffering Speedups

In the first set of experiments, we study the performance improvement from BigCache by providing SSD caching and buffering to a benchmark, assuming that the SSD caches are sufficient to hold the entire working set of the benchmark. We compare the benchmark's performance on the native Hadoop (*No Cache*) to BigCache with four different setups: 1) only caching is enabled and the caches are cold (*Cold Cache w/o Buffer*); 3) only caching is enabled and the caches are warm (*Warm Cache w/o Buffer*); 4) both caching and buffering are enabled and the caches are cold (*Cold Cache w/ Buffer*); 4) both caching and buffering are enabled and the caches are warm (*Warm Cache w/ Buffer*).

When caching is enabled, BigCache will cache the input of the benchmark on the SSDs. Before the first run of the benchmark, the caches do not contain any data of the benchmark and are hence considered as "cold". After the first

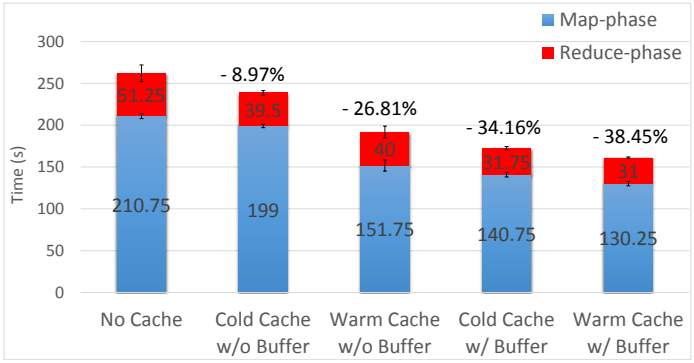

Figure 1. WordCount performance with SSD caching and buffering

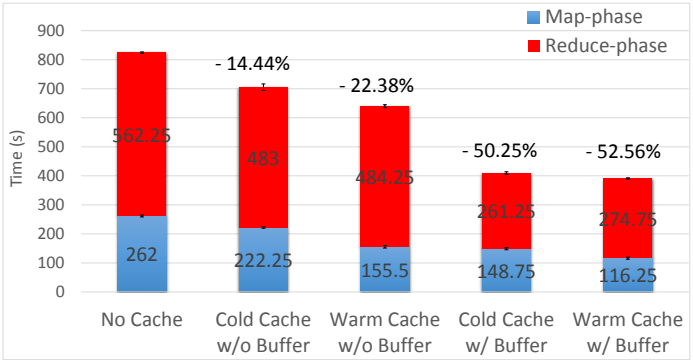

Figure 2. TeraSort performance with SSD caching and buffering

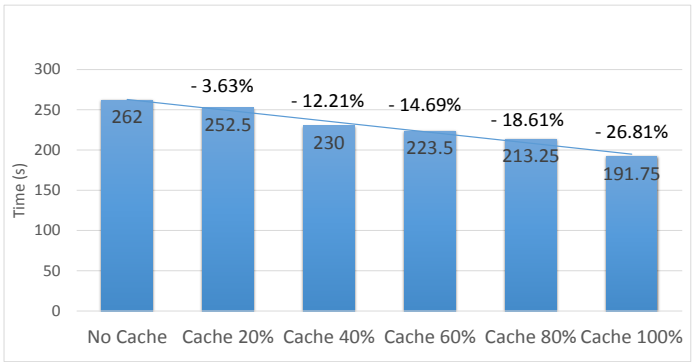

Figure 3. WordCount cache performance model

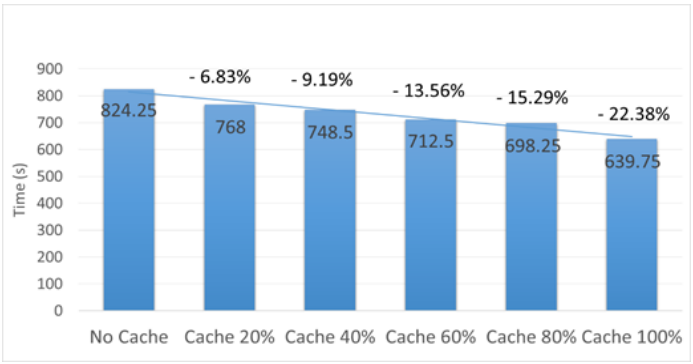

Figure 4. TeraSort cache performance model

run, the caches are "warm", and the cached data may be reused by the following runs of the benchmark and accelerate the map phase. When buffering is enabled, BigCache will buffer the output of the map tasks, which also becomes the input to the reduce tasks, on the SSDs, and can potentially improve both the map and reduce phases.

Figures 1 and 2 show the runtime of WordCount and TeraSort, respectively, under the five different setups. The total height of each bar represents the total runtime of the benchmark. The error bars on the bars represent the standard deviation. Note that the reduce time illustrated in the plots *do not* represent the total runtime of the reduce phase, because the reduce phase starts to copy the map output when a certain percentage (by default 5%) of map tasks finishes. So part of the reduce time is overlapped by the map time in the plots. The numbers on top of the runtime bars represent the reduction of runtimes achieved by BigCache over the native Hadoop.

First thing to notice is that the overhead BigCache is not visible, because even when the caches are cold and the buffering is disabled, which is the worst case scenario for BigCache, it in fact achieves performance improvement (9% runtime reduction for WordCount and 14% for TeraSort). This improvement is because there are already reuse of data blocks during the first run of the benchmark. When the caches are warm, BigCache achieves more substantial improvement (more than 20% runtime reduction for both benchmarks), because the entire input of the benchmark can be loaded from the SSDs.

When BigCache also enables buffering to use SSDs to store the intermediate results of the map phase, the performance of the benchmarks is further improved. However, the two benchmarks differ drastically in terms of the amount of improvement achieved by the SSD buffering. WordCount gets only another 11% reduction of runtime, whereas TeraSort's improvement is doubled. This difference is because TeraSort produces much more intermediate results from the map phase and therefore benefits much more than WordCount. As a result, the total runtime of TeraSort is cut by more than half when both SSD caching and buffering are employed, whereas the total runtime of WordCount is reduced by more than one third, which is also a considerable improvement.

The above results demonstrate that different applications may benefit differently from the use of SSD caching and buffering, which can be exploited to make better use of the limited SSD capacity shared by both caching and buffering. This observation will lead to our future work on more intelligent cache management for BigCache.

C. Cache Performane Models

The experiments discussed above assume that BigCache has enough capacity to hold the working set of the benchmarks. However, in a more realistic setting, the big-data applications may be more IO intensive than the benchmarks considered here, and there may be multiple applications running concurrently and contend for the cache capacity. Therefore, an application may have only part of its working set cached by the SSDs.

Related work [7] on main-memory-based caching assumes that a MapReduce application's input has to be entirely cached or it will not get any speedup. We believe this All-or-Nothing assumption does not hold for large MapReduce applications which have multiple waves of map tasks, and BigCache is designed to be able to cache the partial working set of an application. Our design choice is validated by the next set of experiments.

In these experiments, we enable only the caching (without buffering) in BigCache but vary the cache size to a certain percentage of the benchmarks' input size, from 20% to 100%. Figures 3 and 4 illustrate the performance improvement over the native Hadoop with different cache sizes. We can observe considerable improvement even when only 20% of the working set is cached, and the improvement becomes significant when the cache size grows to more than 40% of the working set size for both benchmarks. The speedup achieved by caching only partial input is also attributed to BigCache's use of MRU replacement policy which recognizes MapReduce applications' sequential read pattern and makes effective use of the limited cache space. These plots also show that the runtimes of both benchmarks fit nicely to simple linear models, which confirms the feasibility of creating accurate cache performance models for deciding an application's cache allocation according to its performance requirement.

V. Conclusions

Big data systems are important platforms for driving data sciences in many different disciplines. Such systems have been traditionally built upon HDD-based storage to support the large *volumes* of data required by big-data applications. However, with the increasing demand on *velocity*, the speed of data access, and *variety*, the types of data, of modern big-data applications, HDD-based storage alone becomes insufficient. Fortunately, another emerging technology that comes to rescue is SSD-based storage which offer excellent performance to both sequential and random IO accesses. However, SSDs still have serious limitations in capacity, cost, and endurance, and must be incorporated into the big-data system architecture strategically, instead of entirely replacing HDDs.

This paper presents BigCache, a first SSD-based caching and buffering solution for big-data systems. It can be seamlessly integrated into existing big-data systems and transparently accelerate the various phases of big-data applications, while still leveraging the size and cost of HDDs to provide the necessary volume and reliability. A prototype of the BigCache approach is created upon Hadoop for MapReduce applications. It is also evaluated experimentally using representative MapReduce applications, WordCount and TeraSort. The results show that BigCache can achieve considerable performance improvement (up to 14% runtime reduction) even when the SSD caches are cold and the SSD buffering is disabled. When the caches are warm, the improvement grows to up to 27%, and when the buffering is also enabled, it can achieve as high as 53% of runtime reduction. The results also show the ability of caching partial input and the choice of cache replacement policy in BigCache allows it to provide speedups to applications when their inputs cannot be entirely stored in the caches, which is important to the increasingly demanding big-data applications and consolidated big-data environments.

VI. Acknowledgement

The authors thank the anonymous reviewers. This research is sponsored by the National Science Foundation under grant CCF-0938045 and CAREER award CNS-125394 and the Department of Defense under grant W911NF-13-1-0157.

VII. References

[1] Jeffrey Dean and Sanjay Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," In Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation – Volume 6, OSDI'04, page 10, Berkeley, CA, USA, 2004.

[2] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in ACM SIGOPS Operating Systems Review, 2003, vol. 37, pp. 29–43.

[3] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), 2010,.

[4] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber, "Bigtable: A Distributed Storage System for Structured Data," In Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation – Volume 7, OSDI'06, pages 15–15, Berkeley, CA, USA, 2006.

[5] HBase. http://hbase.apache.org.

[6] Yucheng Low, Joseph Gonzalez, Aapo Kyrola, Danny Bickson, Carlos Guestrin, and Joseph M Hellerstein, "Graphlab: A New Framework for Parallel Machine Learning," arXiv preprint arXiv:1006.4990, 2010.

[7] G. Ananthanarayanan, A. Ghodsi, A. Wang, D. Borthakur, S. Kandula, S. Shenker, and I. Stoica, "PACMan: Coordinated Memory Caching for Parallel Jobs," in USENIX NSDI, 2012.

[8] Wei Tan, Liana Fong, and Yanbin Liu, "Effectiveness Assessment of Solid-state Drive used in Big Data Services," 21th IEEE International Conference on Web Services (ICWS), 2014.

[9] dm-cache, URL: http://visa.cs.fiu.edu/dmcache.

[10] S. Byan, J. Lentini, A. Madan, L. Pabon, M. Condict, J. Kimmel, S. Kleiman, C. Small, and M. Storer, "Mercury: Host-side flash caching for the data center," In Proceedings of the 28th IEEE Conference on Massive Data Storage, MSST'12, Pacific Grove, CA, USA, 2012.

[11] D. A. Holland, E. L. Angelino, G. Wald, and M. I. Seltzer, "Flash Caching on the Storage Client," In USENIX ATC'13 Proceedings of the 2013 USENIX conference on Annual Technical Conference. USENIX Association, 2013.

[12] D. Arteaga and M. Zhao, "Client-side Flash Caching for Cloud Systems," Proceedings of the 7th ACM International Systems and Storage Conference, June 2014.

[13] Apache Hive, URL: https://hive.apache.org.

[14] The NBER U.S. Patent Citations Data File, URL: http://www.nber.org/patents/Cite75_99.txt.