NativeTask: A Hadoop Compatible Framework for High Performance

Dong Yang*, Xiang Zhong*, Dong Yan*, Fangqin Dai*, Xusen Yin*,
Cheng Lian†, Zhongliang Zhu†, Weihua Jiang*, Gansha Wu*
**Intel Corporation*
Beijing, China
Email: {kenny.yang, xiang.zhong, dong.yan, fangqin.dai, xusen.yin, weihua.jiang, gansha.wu}@intel.com
†*Email: {cheng.lian, zhongliang.zhu}@ciilabs.org*

***Abstract*—Although Hadoop MapReduce provides good programming abstractions and horizontal scalability, it's often blamed for its poor single node performance. In the meantime, MapReduce has already achieved a large install base, thus any performance improvement should keep the compatibility. In this paper, we address the challenges via several approaches guided by low-level performance analysis. And we materialize the approaches via NativeTask, a high-performance, fully compatible MapReduce execution engine. We evaluate its performance with representative HiBench workloads. The results show that the speedup NativeTask achieves ranges from 10% to 160%, and it paves the way for a better MapReduce that excels on both single node performance and scalability. In the future, hardware acceleration can also be applied to further improve the system's efficiency.**

***Keywords*-Hadoop; high performance; compatibility; CPU-bound application; cache-oblivious sort; C++ implementation**

I. Introduction

MapReduce [1][2] is a distributed programming model for massive data processing on large clusters, and Apache Hadoop MapReduce[3] is its most popular open source implementation. Hadoop (we use Hadoop or MapReduce interchangeably in the rest of the paper) is particularly popular among Internet companies that possess massive amount of data, ranging from user generated data to application logs. For example, Baidu [4] processes about 20PB data on a daily basis on its Hadoop clusters, which consists of more than 20,000 nodes. The largest cluster consists of nearly 4000 nodes.

As the demand for big data processing continues to grow, the scale of Hadoop clusters becomes larger and larger. In order to manage large scale clusters, people strive to improve the cluster level efficiency, and build cluster using more powerful nodes. While much emphasis [5][6] is placed on horizontal scalability, the inefficiency of the task engine is left unconsidered. According to our experiments, the task execution engine itself is still inefficient on modern hardware configurations.

Unfortunately the single node performance of Hadoop is very poor. For the HiBench Wordcount workload, it takes more than 150 seconds for one Map task to process 1GB input data compressed with the Snappy codec. This indicates that the throughput of single task is merely 6 MB/s, which is far below the theoretical maximum value.

Behind the performance gap between Hadoop and the ideal system, there lies lots of architectural and implementational issues. While the MapReduce framework addresses these issues from the perspective of task scheduling, monitoring and management, we focus on the per task data processing, because tasks consume most of the cluster execution time and hardware resources.

As CPU core number, memory size and hard disk number per node increase, many workloads change from I/O bound to CPU bound. Compression/decompression of Hadoop tasks incurs high CPU cost, and current implementation of sort further suffers from cache locality. Moreover, we found that serialization and deserialization, stream abstraction, primitive type boxing and unboxing in Hadoop Java tasks lead to significant object creation and buffer copying.

In this paper, we use low-level performance counters to study the behavior of workloads running on Hadoop clusters. We explain the reasons behind the poor performance of Hadoop tasks. Meantime, we overcome the performance issues without sacrificing compatibility, as such the performance of existing Hadoop applications can be boosted with zero effort. We implemented NativeTask, a high-performance, fully compatible execution engine for Hadoop. We also report our experiences on how NativeTask accelerates representative benchmarks in the HiBench suite.

Our primary contributions include:

- A thorough characterization and analysis of task data processing with bottlenecks identified;
- A modified MapReduce implementation with new design of Java-native interaction to remove identified bottlenecks, while still maintaining compatibility; this also opens the opportunities for hardware acceleration in the future;
- An extensive evaluation of the implementation on a cluster, including comparisons to a standard MapReduce implementation.

The rest of this paper is organized as follows. We describe our goals and non-goals in Section II. In Section III, we analyze the Hadoop performance issues and describe our approaches. In Section IV, we present the implementation of NativeTask. In Section V we evaluate the performance of NativeTask. Section VI surveys the related work. Finally, Section VII concludes the paper and describes future work.

II. Our Goals and No-Goals

Our goal is to design and implement a framework that significantly improves Hadoop performance, especially task execution performance. In particular, we want our framework to be broadly applicable to all MapReduce applications. Three design goals lead to the following requirements.

- High performance: Our framework should be more efficient on resource utilization on a single node.
- Compatibility: Our framework should not require any modifications to the cluster runtime, middleware, messages, or applications.
- Robustness: Our framework should be able to fall back to Vanilla Hadoop implementation automatically in case that the framework fails to execute certain tasks for several times.

Two no-goals need to be noted:

- We do not specifically optimize the framework to support more efficient shuffle. But the shuffle phase of a Hadoop job can be improved as a side product of faster task engine.
- Our goal is not to optimize job or task scheduling. We focus on task instances running on a node after the scheduling.

III. Issues and Approaches

Our approaches focus on improving task execution on each node in the cluster, which further improve the whole performance of jobs. The overall optimization process consists of three main aspects: improving efficiency, keeping compatibility and guaranteeing robustness.

A. Approach to Efficiency

We use HiBench as the target workloads and try to analyze its performance on a node. Its configuration is as follows: Intel(R) eight-Core E5-2680 CPU 2.70GHz, 512KB L1, 2MB L2, 20MB LLC, 32GB RAM, Linux Fedora14, Hadoop 1.0.3, Sun Java 1.6u31, and four Western Digital HDD SATA 500GB. We perform profiling with YJP and OProfile tools.

1) I/O-bound to CPU-bound: Depending on specific Hadoop applications, computation may be bound by I/O, memory, or CPU resources. Sufficient memory capacity is critical for high utilization of servers in a Hadoop cluster, because more map and reduce tasks can be carried out on one node simultaneously.

Each map task has a memory buffer to which it writes the output. The buffer size can be tuned by changing the io.sort.mb property. When the content of the buffer reaches a certain threshold size, a background thread starts to spill the content to disk. Each time the memory buffer reaches the spill threshold, a new spill file is created. After the map task finishes the writing of its last output record, there could be several spill files. Before the task is finished, the spill files are merged into a single partitioned and sorted output file. So the higher spill frequency, the more I/O operations and the worse job performance.

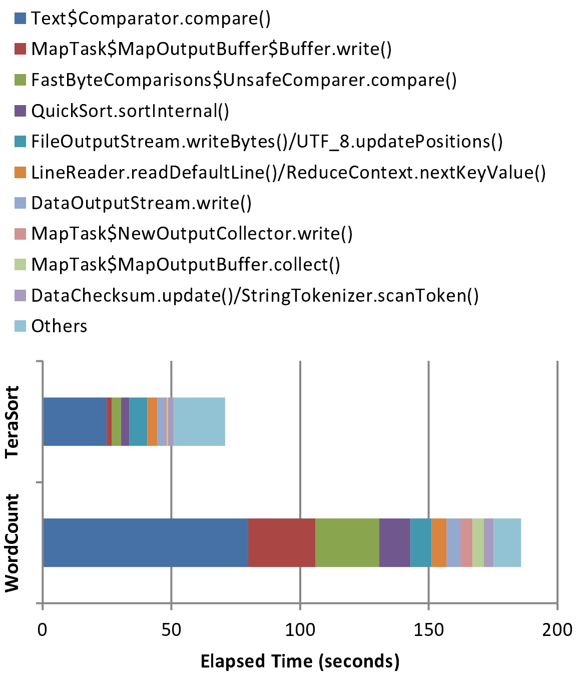

Figure 1. The time breakdown of map task for TeraSort and WordCount.

As RAM memory per node increases, more memory can be allocated for each map or reduce task. We tune up the map task buffer size to make sure that spill operation performs only once. In this case, the biggest cost in the map task is from the computation rather than I/O operations. That means I/O bound workloads may trend to be CPU bound.

Although we decrease the spill frequency to reduce the overhead of read and write stages, benchmarks still dont perform efficiently. Next, we profile the process of task execution and find that sort stage occupies most time, as shown in Fig. 1.

2) Cache-aware to cache-oblivious: We continue identifying the cause of the long-running sort. The processors feature a three-level cache hierarchy where the last-level cache (LLC) is a large-capacity cache shared among all cores. We investigate the cache behavior of Wordcount and find that L2 and LLC cache miss rates are 79% and 64% prospectively. Large LLC does not help on the performance.

The sort algorithm used by Hadoop is usually a two-way recursive algorithm such as Quick Sort or Merge Sort. [4], [7] investigate the cache effect on sorting algorithms both experimentally and analytically. Because Hadoop workloads operate on massive data sets and launch a large number of concurrent tasks, both the data sets and the per-task data are orders of magnitude larger than the available on-chip cache capacity. So double or triple the capacity of LLC does not bring obvious improvement [8]. Tuning sorting buffer size specified by io.sort.mb does not help. Large io.sort.mb leads to high cache miss rate, while small io.sort.mb is even worse since it increases spill cases and incurs more I/O operations.

To address the performance penalty by high cache miss rates, we choose to restructure Hadoop sorting algorithm in order to improve cache locality.

Suppose that α is the total number of elements, β is the size of a cache line, and γ is the number of lines in a cache. For two-way recursive algorithms, at every level all α elements are processed, which means all α elements are loaded into cache. If loading one line at a time, for every β elements loaded into cache, there is a cache miss. That means $1/\beta$ amortizes cache miss per element, so $O(\alpha/\beta * (\log \alpha)/(\log 2))$ is cache miss bound. While doing an $O(\gamma)$-way Merge Sort, we try to get a cache aware sorting algorithm with $O(\alpha/\beta * (\log \alpha)/(\log \gamma))$ cache miss.

We get the idea from Funnel Sort [9] whose intuition is to recursively lay out a K-way merge with smaller funnels. But it introduces much more memory management overhead than Quick Sort, and it also requires lots of calculations to keep track of buffers and how full they are, so the actual implementation of Funnel Sort often performs poorly. Moreover, parallelized Funnel Sort is not necessary for MapReduce because there are already multiple Hadoop tasks running on a node.

Our approach is different from Funnel Sort which recursively lays out a K-way merge. It also differs from Hadoop which sorts the single map output buffer. Because the map output buffer is partitioned in the map task, each partition buffer is sorted individually. Furthermore, each partition is divided into multiple small chunks. We sort each small chunk by Quick Sort individually. While sorting a partition, we merge sort all these chunks of the partition by heap sort which is an $O(\alpha)$ sorting algorithm. Finally, parallel sorting is achieved by parallel tasks on one node. In our design, chunk is small enough so that sorting fits into cache. Even for parallel tasks, the sorting chunk per task fits in cache. Moreover, suppose that δ is the number of chunks in parallel tasks on one node, γ is much larger than δ. For example, when 16 map slots, io.sort.mb of 1GB, 1MB chunk, 64bytes cache line, and 20MB*4 LLC are used, γ is 400K and δ is 16K. In this case, our algorithm is close to cache miss bound.

We investigate the impact of LLC capacity on the Wordcount workload. Here sorting time does not including final heap sort. We have two data sets and two nodes. The first data set is 200MB and the key length is 15 bytes, the other is 600MB and the key length is 20 bytes. One node configuration includes Intel(R) quad-Core i5-650 CPU 3.20GHz, 512KB L2, 4MB LLC, 4 GB RAM, and the other includes Intel(R) Xeon(R) quad-Core E5-2680 CPU 2.70GHz, 2MB L2, 20MB LLC, 64GB RAM.

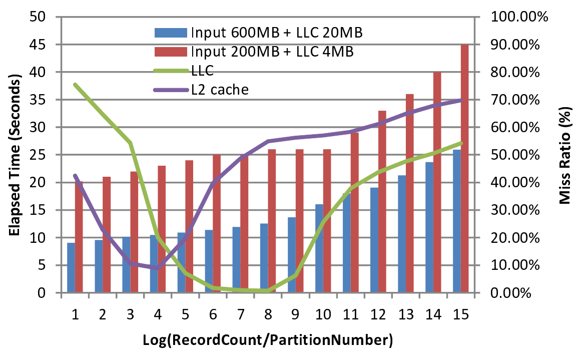

Figure 2. Sort time and cache miss rates with different cache sizes.

As shown in Fig. 2, we find sorting time increases when the number of records per partition becomes larger. For 20MB LLC, sorting performance declines more quickly when the chunk size exceeds the LLC capacity. It is the same as the case with 4MB LLC. As shown in Fig. 2, when sorting small chunks, more cache references occur at L2 cache and less references at LLC, so the miss rate of LLC is high. When sorting large chunks, more references are at LLC and the miss rate of LLC becomes lower if the chunk size is less than the LLC capacity. When the scale of data set exceeds the LLC capacity, miss rate becomes high. Because L2 cache is smaller than LLC, the miss rate of L2 cache decreases earlier.

3) Java to C++: Actually Java is efficient for MapReduce tasks, and Java has some runtime optimizations techniques that are more difficult to realize in C or C++. For example, it is difficult to do dynamic optimizations, such as lock coarsening and virtual function inlining, in C++. But there are some still optimization factors essential for performance, and more suitable to run in a native runtime.

The first optimization is compression and decompression. Nearly the fastest compression algorithms are written in native code. Currently Hadoop uses JNI to call these libraries in a bulk processing manner, but still there is significant overhead crossing JNI boundary, especially when decompression speed is fast. And for some other techniques like lazy decompression, direct operations on compressed data cannot fit in bulk processing.

The second is vectorization optimization. Currently Hadoop uses JNI to leverage SIMD (i.e. SSE/AVX) optimizations such as CRC checksum. But it only touches a small portion of the stack and performance speedup is limited.

The third is the efficiency of compiler generated code. One of our objectives is to make Hive running on MapReduce have competitive performance as high level query execution engines written in native code. However, when comparing some microbenchmarks, the LLVM compiled C++ code is

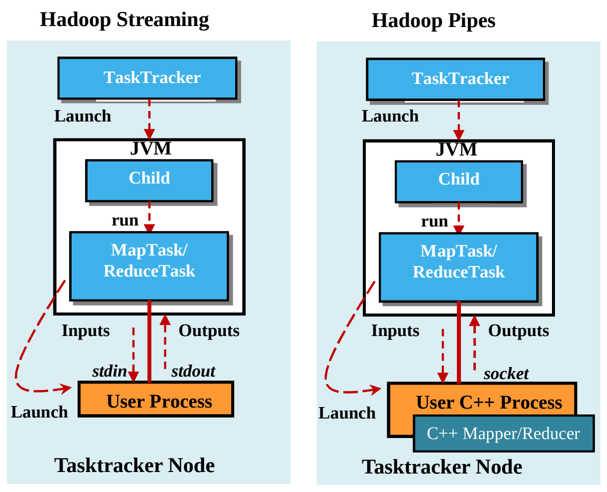

Figure 3. Hadoop Streaming and Pipes.

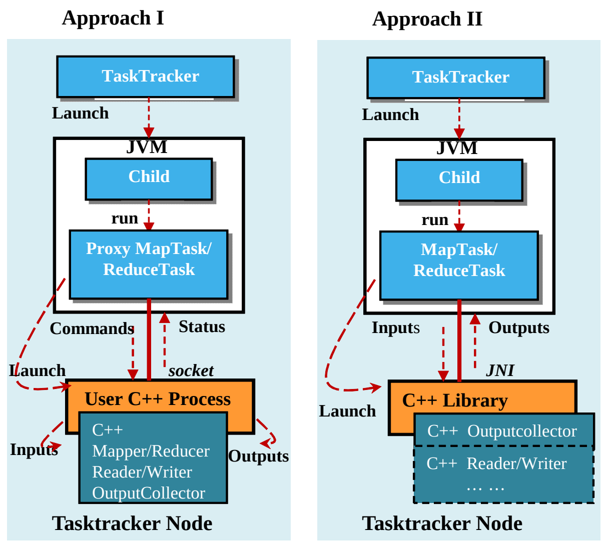

Figure 4. Two kinds of Hadoop execution frameworks.

much faster than the Java JIT'ed code.

The fourth is memory copying. A lot of serialization and deserialization result in high overhead. In order to achieve maximum throughput, it is better to abandon serialization, or introduce a new serialization method that can operate directly on serialized data, avoiding object creation and memory copying. These are hard and not user-friendly in Java, but convenient and straightforward in native code, using c-struct like data representation. Moreover, when the whole data flow, from CRC checksum, decompression, reader, mapper/reducer, writer, compression to CRC checksum, is in the native space, it has even better opportunity to eliminate small memory copies. We do our best to design the interface and the underlying processing flow to eliminate most memory copies.

B. Approach to Compatibility

In order to keep complete compatibility, we should keep intact the user programming interface, configuration settings, logging pattern, monitoring tools and visualization interface.

There are two frameworks in Hadoop for multi-language support, Hadoop Streaming and Hadoop Pipes, as illustrated in Fig. 3. Hadoop Streaming allows users to create and run MapReduce jobs with any executable or script as the mapper and/or the reducer. Both the mapper and the reducer are executables that read the input from stdin and emit the output to stdout. Hadoop Pipes allows C++ code to use Hadoop DFS and MapReduce. The primary approach is to split the C++ code into a separate process that does the application specific code. In many ways, this approach is similar to Hadoop streaming, but using Writable serialization to convert the types into bytes that are sent to the process via a socket.

Following Streaming and Pipes, we choose to move some Hadoop execution components from Java to native space. There are two different approaches on how to implement cross-language communication. As illustrated in Fig. 4. Approach I moves all execution components into C++ space, including mapper, reducer, reader, writer and collector. In this case, Hadoop task is a proxy, actual data processing stages are all in C++ space. Approach II moves non-user-defined components to C++, except mapper, reducer, partitioner, combiner, reader and writer. In this case, input data is read in from Java side, and then transferred into C++ side by JNI. When combiner is invoked, data is also transferred from Java to C++.

Approach II has better compatibility, but it suffers from performance loss due to cross-language data serialization and memory copying. One of our objectives is to build a high performance data warehouse to support ad-hoc queries, and it is impossible to write the native mapper/reducers for all queries. So we design NativeTask to support both two approaches.

NativeTask consists of two major components, Java engine and native engine. Java engine is responsible for bypassing normal Java data flow and delegating the data processing to native side. The actual computations runs in the native engine. Java and native engine communicate with each other using JNI, in a synchronized and block based batch processing way. This is different from other IPC mechanisms used in Hadoop Streaming and Pipes. Sockets and pipes are fast enough for data processing, but they consume lots of CPU and introduce some issues such as multi-thread programming and asynchronous processing.

C. Approach to Robustness

Even if NativeTask is well tested, as a new framework/library, we cannot assure its reliability for all time.

In order to make NativeTask more robust, we design a mechanism to deal with unknown exceptions. We separate two task jars from MapReduce jar package. When deploying the MapReduce system, we put two task jar packages in the related path. One is the traditional Hadoop task jar and the other is NativeTask jar. If system runs normally, TaskTracker loads NativeTask jar and creates task process. When task execution failures exceed a certain limit, system chooses Hadoop task jar to run instead of NativeTask jar, and failure information is still written into the job log. In this way, we complete failover implicitly and enhance system robustness.

Another situation is in case jobs on NativeTask become slower, it may take a long time to complete jobs, but users need results in time. At this time, the framework allows users to change to original Hadoop mode. Users can resubmit their uncompleted jobs with a configuration option or add an option in the client configuration file.

IV. NATIVETASK

A. Implementation of Task Delegation

We introduce a task delegation interface to bypass normal Java data flow. At the beginning of map task and reduce task, a delegator is used to run the task bypassing the original logic, if it is configured in the job configuration file.

NativeTask supports Java mapper, reducer, reader and writer, which leads to full compatibility with Hadoop applications. Key/value pairs are passed to or from native side in batch mode. In addition to this, task delegation also supports another kind of data flow, which involves native mapper/reducer with native reader and writer. In this case, native reader/writer directly reads and writes the data, which yields better performance and flexibility. However it still resorts to the Java code when dealing with input and output formats for input split and output commit.

For reduce task, shuffle and merge sort are still done on Java side. A native version of shuffle and merge will be implemented in future.

Due to lack of object reflection in C++, it is difficult to set mapper, reducer, combiner, reader and writer components in the job configuration file on client side and create them dynamically on server side. Rather than static linking by Hadoop Pipes, NativeTask chooses a more dynamic method which loads class libraries based structure.

NativeTask uses templates to implement an equivalent version of instance reflection in Hadoop. Considering .so library as class library just like .jar files, every .so library has an entrance function to create C++ objects of the classes in the library. The library libnativetask.so is NativeTask runtime and also serves as a class library with some predefined mapper, reducer, partitioner, reader and writer. The data flow and main logic of these components are almost the same as the original implementation. One difference is that native implementation tends to be more compact and easy to improve. In addition, the mapper, reducer, reader and writer APIs are designed to make zero copy possible.

B. Implementation of Cross-language Communication

Between Java and native side, the serialized key and value pairs are transferred in a per-block pattern rather than a per-record one so as to decrease JNI calls overhead. The block size ranges from 32KB to 128KB, which is smaller than L2 cache.

To minimize buffer copying, two lightweight I/O buffers, namely read buffer and append buffer, are introduced. Java and Hadoop I/O streams used the decorator pattern extensively which introduces complex class hierarchy. But NativeTask emphasizes more on code efficiency, e.g. frequently invoked methods are implemented with the inline mode, and meantime we make best effort to avoid buffer copying while supporting compression and decompression. We only use the decorator pattern for stream I/O in the batch mode, such as file read, write and CRC checksum. It is easier to add compression codecs in the native code, and snappy, lz4 and gzip have already been integrated into framework.

The JNI based batch processing is both implemented on Java and C++ sides and we encapsulate them into two classes, on which other components can operate, without incurring the complexity of JNI.

C. Implementation of Memory Management

Basically, map output collector maintains a partitioned buffer which stores key and value pairs. Mapper emits key and value pairs, and a partition number is generated by the partitioner. Map output collector puts key and value pairs into partition buckets. Each partition bucket has two arrays, memory blocks vector used by this bucket and offsets vector which maintains the starts and offsets of key and value pairs in memory pool.

Partition bucket has an array of memory blocks to hold key and value pairs. If memory blocks are used up, a new memory block is allocated from memory pool. If there is not enough memory in memory pool, a spill operation will be activated.

Memory pool reserves a memory space with a size of io.sort.mb, and tracks its usage. If not actually accessed, the memory will not be allocated in NativeTask The actual memory allocation happens only when a block is requested and released to a bucket. Fig. 5 illustrates memory management and data processing in the map task.

Memory blocks backed by memory pool are used by partition bucket and designed to be CPU cache friendly. When sorting large indirectly-addressed key and value pairs, sorting time is dominated by RAM random reads. Memory chunk is used to help each bucket get relatively continuous memory.

The block size is determined by cache size and memory usage. Usually the minimum block size equals to 32K, and the maximum size equals to 1M.

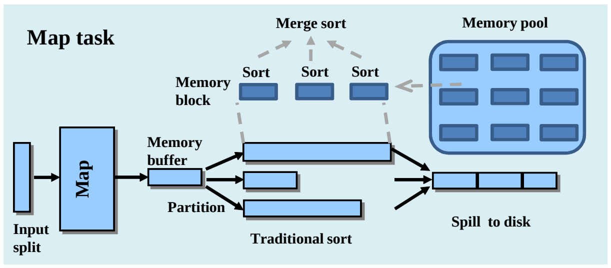

Figure 5. Buffers of map task in NativeTask.

Job	Wordcount
Compression	Enabled (Snappy)
dfs.block.size	256MB
io.sort.mb	1GB
Cluster size	4
Hadoop version	1.0.3
slots	Map: 3*32+1*26 = 122 Reduce: 3*16+1*13=61
CPU	Intel(R) Xeon(R) 8-Core E5-2680 2.70GHz
L1 cache	512KB
L2 cache	2048KB
LLC (L3 cache)	20MB
Memory	64GB, 3 DDR3 channels
Disk	7 SATA 500GB

Table I
CLUSTER AND JOB CONFIGURATION

The efficiency decreases when the partition number and key/value sizes become large. In cases that the ratio of io.sort.mb to the partition number is too small, we can use memory pool directly with memory blocks disabled.

V. EVALUATION

We have conducted extensive evaluation on the NativeTask framework. We evaluated 9 representative applications in HiBench. The results show that NativeTask performs more efficiently than the vanilla MapReduce. We also analyzed several factors that may affect job performance, including differences between CPU bound and I/O bound workloads, relationship between task and job speedup, and the efficiency of sorting optimization and trade-offs between partial NativeTask and full NativeTask.

A. CPU-bound and IO-bound Workloads

[10] conducted experiments with various HiBench benchmarks. Fig. 6 and Fig. 7 show the speedup for CPU bound and I/O bound workloads respectively.

Wordcount counts the number of words in an 1TB data file created by Randomtextwriter. The output file size of Wordcount is very small because the final results are aggregated by words as keys. That is, the ratio of map/reduce output file size to the initial input size is very small, so I/O operations are negligible. Pagerank is a link analysis algorithm. Its map is CPU bound and reduce is I/O bound. The number of pages is 500M and total input size is 481GB if uncompressed. NativeTask achieves about 50% improvements for both workloads, as measured by the job performance.

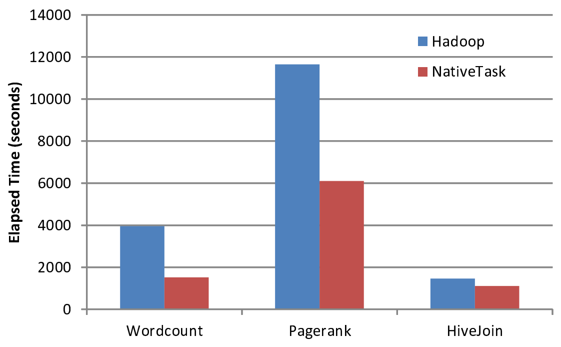

Figure 6. HiBench workloads (CPU bound).

Both Hive Join and Aggregation queries are adapted from the query examples in Pavlo et.al [11] and their inputs are defined in [11]. They are intended to model complex analytic queries over structured tables. Hive Aggregation computes the sum of each group over a single read-only table, while Hive Join computes both the average and sum for each group by joining two different tables. Hive Join is CPU bound. For Hive Aggregation, its map phase is CPU bound and reduce phase is I/O bound. We use 5GB user visits log and 860GB page as input data. Because most of computational logics are implemented in Hive codes, NativeTask speedup ranges from 10% to 30%.

Generally speaking, for I/O bound workloads, the performance improvement of NativeTask is smaller. TeraSort is a standard MapReduce benchmark, which samples the input data and uses map/reduce to sort the data into a total order. This algorithm generates the sample keys by sampling the input before the job is submitted and then writes the list of keys into HDFS. For TeraSort, map is CPU bound and reduce is I/O bound. Compared with the 110 minutes running time on MapReduce, NativeTask only takes 70 minutes to sort 1TB data. The Sort benchmark sorts the data in ascending order. NativeTask is 10% faster. DFSIO is a test on Hadoop I/O performance and throughput. DFSIO writes 1TB data and also reads 1TB data. NativeTask's improvement is also around 10%. K-Means is a simple but well known algorithm for grouping objects, a.k.a. clustering. Here all objects need to be represented as a set of numerical features. In addition the user has to specify the number of groups he wishes to cluster into. For K-Means, Iteration stage is CPU bound and classification stage is I/O bound. After the 380GB input data is generated from logs, it is classified in five groups. For this workload, NativeTask is 17% faster.

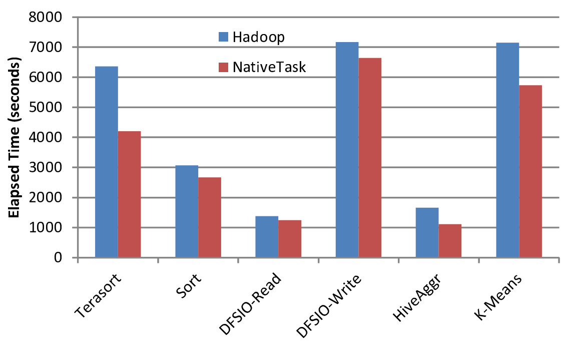

Figure 7. HiBench workloads (I/O bound).

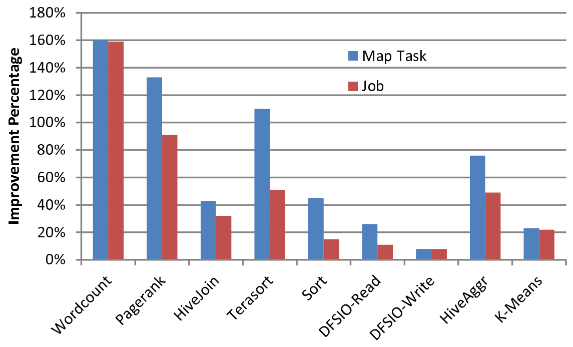

Figure 8. Improvement factor of job and map for HiBench Workloads.

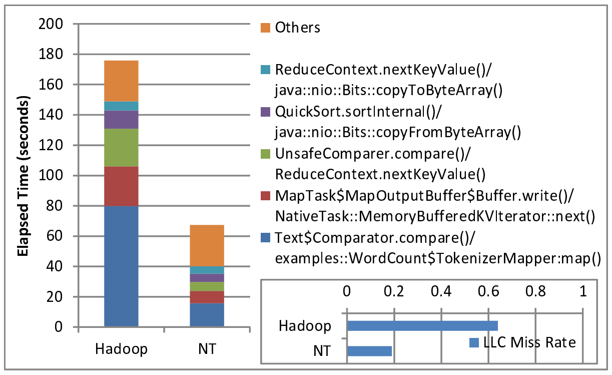

Figure 9. Improvement factor of cache-oblivious sorting for WordCount.

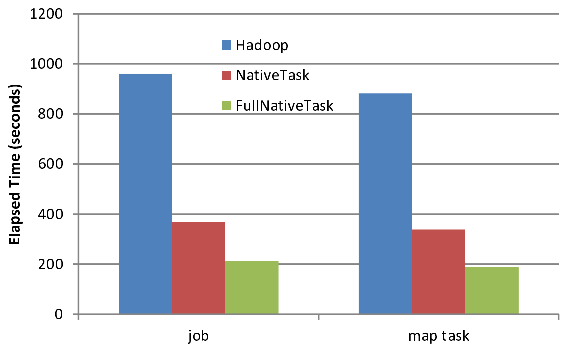

Figure 10. Wordcount performance on NativeTask and FullNativeTask.

B. Relationship between Task and Job Improvement

In Fig. 8, we compare the NativeTask speedups between job and map tasks for HiBench workloads. For Wordcount, DFSIO-Write and K-Means, the speedup of map task almost equals to that of job, because the data handled by reduce task is very little or jobs are map-only. But for K-Means, classification in the reduce task takes more time so that NativeTask's speedup is not so significant. For other workloads, map improvements are higher than job improvements, because these jobs have heavy duty reduce tasks in addition to map tasks. For PageRank and TeraSort, map tasks cost lots of CPU cycles, so the map optimization improves the performance dramatically.

C. Efficiency of Sorting Optimization

As shown in Fig. 9, the task execution time of WordCount exceeds 150 seconds and the LLC miss rate is 64%. We profiled the task execution cycles and found the sorting stage costs more than 60%. Using cache-oblivious sorting method and NativeTask optimization framework, we reduce task execution time to about 1 minute and achieve a much lower LLC miss rate of 19%.

D. Full and Patial NativeTask

As mentioned above, we implemented the compatible NativeTask and full NativeTask. For the latter case, all task execution components are implemented in C++, so it achieves highest performance at the cost of compatibility. Compared with compatible NativeTask, full NativeTask gets 2x more speedup, as shown in Fig. 10. Users can choose compatibility mode or high performance mode per their requirements.

VI. Related Work

There are some research efforts to enhance the MapReduce framework. In [12][11], the authors compared MapReduce with a traditional parallel database. The authors also speculated about possible architectural causes for the performance gap between the two systems. For example, record parsing is recognized to bring about performance overhead. [13] suggests MapReduce users to avoid using text format and prefer ProtocolBuffer for encoding and decoding binary structured records. [14] is similar to our unsorted data flows, but we build these on the native side and get higher efficiency. [15] tries to dynamically adjust job configurations rather than optimizing data processing logic.

[16] in 2011 figures out four factors that affect the performance of data processing, namely I/O mode, indexing,

parsing and sorting. Our work is different in the following aspects. First, it suggests that direct I/O and streaming I/O are helpful. However, due to the lightweight compression and decompression, CPU is the bottleneck in real world clusters. Second, it is reasonable to save I/O bandwidth via indexing technology. Third, decoding and encoding code in Java is not optimal, some serialization and memory copying operations can be avoided by reconstructing the data flow. Fourth, their work adopts a fingerprint comparison strategy to reduce the cost of comparing two keys that have different fingerprints. But fingerprinting doesn't always differentiate two keys effectively. Partition based sorting outperforms fingerprint comparison based sorting because of smaller computational complexity and lower cache miss rate.

VII. Conclusion and Future Work

In this paper, we studied the characterizations and bottlenecks of MapReduce based on its open source implementation, Hadoop. We figured out three factors that affect the performance of MapReduce tasks, especially the map tasks. We investigated a variety of optimization strategies for each factor. The main insight we achieved is that resource utilization of the whole cluster can be improved significantly by optimizing the task engine.

We also discussed our experiences in designing and implementing NativeTask. As an open source C++ data processing engine, it is completely compatible with Hadoop and supports existing MapReduce applications without any modification. Finally, we evaluated the performance of NativeTask. The experimental results show that HiBench workloads can be improved by a factor of 10% to 160%.

In the future, we intend to extend our research in three areas. First, we will continue improving NativeTask for higher efficiency, such as native shuffle, reduce merge and so on. Second, we hope to integrate NativeTask with upper level data warehouses such as Hive or Tez for optimizing the whole infrastructure. Third, we want to run NativeTask with hardware accelerators and heterogeneous computing environments such as Xeon Phi, in order to explore more optimization possibilities of the framework.

We acknowledge the feedbacks from the reviewers. Binglin Chang from VMWare also has significant contributions to NativeTask.

References

[1] J. A. Stuart and J. D. Owens, "Multi-gpu mapreduce on gpu clusters," in *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, ser. IPDPS '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 1068–1079.

[2] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: a warehousing solution over a map-reduce framework," *Proc. VLDB Endow.*, vol. 2, no. 2, pp. 1626–1629, Aug. 2009.

[3] A. Abouzeid, K. Bajda-Pawlikowski, D. J. Abadi, A. Rasin, and A. Silberschatz, "Hadoopdb: An architectural hybrid of mapreduce and dbms technologies for analytical workloads," *PVLDB*, vol. 2, no. 1, pp. 922–933, 2009.

[4] A. LaMarca and R. E. Ladner, "The influence of caches on the performance of sorting," in *proceedings of the seventh annual ACM-SIAM symposium on discrete algorithms*, 1997, pp. 370–379.

[5] B. Chattopadhyay, L. Lin, W. Liu, S. Mittal, P. Aragonda, V. Lychagina, Y. Kwon, and M. Wong, "Tenzing a sql implementation on the mapreduce framework," in *Proceedings of VLDB*, 2011, pp. 1318–1327.

[6] R. Chaiken, B. Jenkins, P.-Å. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou, "Scope: easy and efficient parallel processing of massive data sets," *PVLDB*, vol. 1, no. 2, pp. 1265–1276, 2008.

[7] A. Maus, "Sorting by generating the sorting partition, and the effect of caching on sorting," in *NIK'2000. Norwegian Informatics conference (ISBN 82-7314-308-2)*, 2000, pp. 19–30.

[8] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[9] G. S. Brodal, R. Fagerberg, and K. Vinther, "Engineering a cache-oblivious sorting algorithm," 2006.

[10] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang, "The hibench benchmark suite: Characterization of the mapreduce-based data analysis," in *Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on*, 2010, pp. 41–51.

[11] M. Stonebraker, D. Abadi, D. J. DeWitt, S. Madden, E. Paulson, A. Pavlo, and A. Rasin, "Mapreduce and parallel dbmss: friends or foes?" *Commun. ACM*, vol. 53, no. 1, pp. 64–71, Jan. 2010.

[12] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker, "A comparison of approaches to large-scale data analysis," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, ser. SIGMOD '09. New York, NY, USA: ACM, 2009, pp. 165–178.

[13] J. Dean and S. Ghemawat, "Mapreduce: a flexible data processing tool," *Commun. ACM*, vol. 53, no. 1, pp. 72–77, Jan. 2010. [Online]. Available: http://doi.acm.org/10.1145/1629175.1629198

[14] A. Verma, B. Cho, N. Zea, I. Gupta, and R. Campbell, "Breaking the mapreduce stage barrier," *Cluster Computing*, vol. 16, no. 1, pp. 191–206, 2013.

[15] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu, "Starfish: A self-tuning system for big data analytics," in *In CIDR*, 2011, pp. 261–272.

[16] D. Jiang, B. C. Ooi, L. Shi, and S. Wu, "The performance of mapreduce: an in-depth study," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 472–483, Sep. 2010.