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 provides good horizontal scalability and makes it easy to develop general application, low efficiency of task execution per node remains a challenge. Meantime, any improvement needs to be fully compatible to Hadoop, since many existed applications are developed based on Hadoop. In this paper, we describe several approaches to help address these challenges based on performance analysis. Based on these approaches, we introduce NativeTask, a high-performance, compatible, robust and extensible execution engine for Hadoop. We evaluate the performance of Hadoop based on NativeTask using representative workloads of HiBench. The results show that the performance of Hadoop can be improved by a factor of 10% to 160%. Our approaches show that it is possible to build a MapReduce-based system that is not only extensible and scalable, but also efficient, and in the long run, hardware acceleration can be applied to further improve the efficiency.**

***Keywords*-MapReduce; Hadoop; task; native; efficient; compatible**

I. Introduction

MapReduce [1][2] is a distributed programming model for processing massive datasets on clusters. Apache Hadoop [3] is the most popular open source implementation of the MapReduce model. Hadoop is especially popular in Internet companies, where users generate massive amount of data, ranging from user sessions data to application logs. For example, Baidu [4] processes about 20 petabytes of data per day on Hadoop clusters. The total number of Hadoop nodes exceeds 20000 and the largest cluster has nearly 4000 nodes.

As the demand for big data processing continues to grow, the scale of Hadoop cluster becomes larger and larger. In order to handle large scale clusters, people strive to try out to improve the efficiency of the whole cluster, and build cluster using more powerful machines. Although major design changes are being introduced at schedule and management of Hadoop [4], the task execution engine used in new Hadoop are still inefficient for running typical workloads with todays hardware configuration. While much emphasis is placed on horizontal scalability, the inefficiency of task engine is left unconsidered.

Especially single node performance of Hadoop is very pool. For the HiBench Wordcount workload, it may take one Hadoop task more than 150 seconds to handle 1GB input data compressed by snappy codec. This means the throughput of task is only 6 MB/s, which is far below maximum in theory.

There are lots of architectural and implementation causes for the performance gap between Hadoop and the ideal system. While MapReduce framework address these problems from task scheduling, monitoring and management, we focus on the data processing of tasks because tasks consume most of the cluster execution time and hardware resources.

As CPU cores, memory size and attached disks in one node increases, many workloads change from IO bound to CPU bound. Merge-sort stage of Hadoop task incurs high CPU cost, and current implementation suffers from cache locality problem. Moreover, we find that serialization and deserialization, stream abstraction, primitive type boxing and unboxing in Hadoop Java task can lead to lots of inevitable object creation and buffer copies.

In this paper, we use performance counters to study the behavior of workloads running on Hadoop clusters. We explain the reasons behind poor performance of Hadoop tasks. Meantime, we address the performance issues while still maintaining compatibility so as to improve performance of Hadoop without any modification to applications. We implemented NativeTask, a high-performance, fully compatible, lightweight, robust and extensible execution engine for Hadoop. We also report our experience on how NativeTask helps standard benchmark HiBench to efficiently improve performance.

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 accelerate the identified bottlenecks, while still maintaining the 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 gives plans for future.

II. Our Goals and Non-Goals

Our goal is to design and implement a framework that efficiently improves Hadoop performance, especially task execution performance. In particular, we want our framework to be broadly applicable to many different applications and systems based on Hadoop. Three design goals result from these requirements.

- High performance: Our framework should be more efficient than Hadoop and fully utilize resources 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 allowed to fall back to Hadoop automatically in case tasks fail to execute several times.

Two non-goals need to note:

- We dont specifically optimize the framework to support more efficient shuffle. Of course shuffle phase of job can be improved because of faster map tasks.
- Our goal is not to optimize job or task scheduling. We focus on task instances running on a node after 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 enhancing robustness.

A. Approach to Efficiency

We use HiBench as the workloads and try to analyze performance on a node. 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, four Western Digital HDD SATA 500GB is used. Namenode, Datanodes, Jobtracker and Tasktrackers are deployed on the same node. Each node is configured to 16 map slots and 8 reduce slots. HDFS block size is configured to 64MB. Input data, map output data and final output data is all compressed. The size of input data per task is 800MB uncompressed and io.sort.mb is set to 1GB. We perform profiling with YJP and OProfile tools.

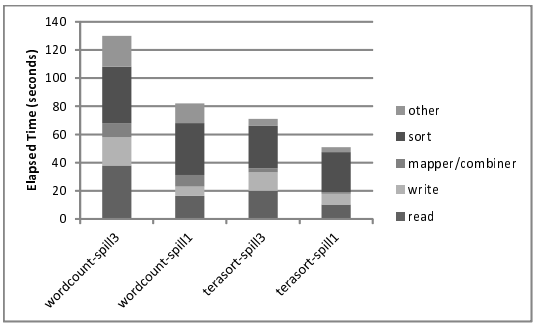

Figure 1. The Time Breakdown of Map Task in Hadoop.

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 contents to disk. Each time the memory buffer reaches the spill threshold, a new spill file is created, so after the map task has written 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.

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 map task is from computational operation instead of I/O operations. That means I/O bound workloads may become 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 the most cost, as shown in Fig. 1.

2) Cache-aware to cache-oblivious: We continue identifying the cause of 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 64% and 79% prospectively. Large LLC doesnt help on the performance.

Sorting algorithm used by Hadoop is usually a two-way recursive algorithm such as Quick Sort or Merge Sort. [4], [5] investigate the effect that caches have on the performance of sorting algorithms both experimentally and analytically. Sorting buffer decided by io.sort.mb is larger than LLC,

leading to poor cache performance results. Because Hadoop workloads operate on massive data sets and service a large number of concurrent tasks, both the dataset and the per-task data are orders of magnitude larger than the available on-chip cache capacity. On the other hand, [6] shows that a LLC that captures the working set achieves nearly the same performance as a LLC with double or triple the capacity.

To address the performance problems that high cache miss penalties introduce, we choose to restructure Hadoop sorting algorithm in order to improve cache locality.

Suppose that α is the number of elements, β is the size of cache line, and γ is the number of lines in a cache. For two-way recursive algorithm, at every level all α elements are processed, which means all α elements are loaded into cache. If loaded one line at a time, for every β elements loaded into cache, there is a cache miss. That means $1/\beta$ amortized 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 [7] whose intuition is to recursively lay out a K-way merge with smaller funnels. But it also introduces much more memory management duties than Quick Sort and lots of calculations to keep track of buffers and how full they are, so the actual implementation of Funnel Sort often performed 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 and Hadoop which sorts the single map output buffer. Because map output buffer is partitioned in map task, we can sort each partition buffer individually. Furthermore, we divide each partition into multiple small chunks. We sort each small chunk by Quick Sort individually while sorting a partition, then we merge sort all these chunks belong to one partition by heap sort which is an $O(\alpha)$ sorting algorithm. We implement parallel sorting 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 will fit in cache. Moreover, suppose that δ is the number of chunks in parallel tasks on one node, γ is much more than δ. For example, when 16 map slots, 1GB io.sort.mb, 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 utility of different LLC capacity for Wordcount workloads. Here sorting time doesnt including final heap sorting. 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.

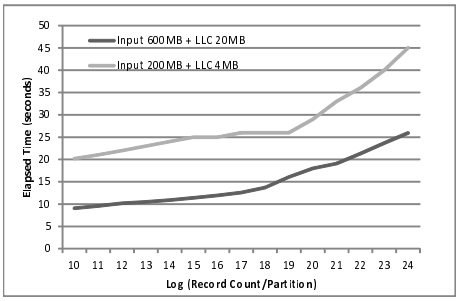

Figure 2. Partition sort time with different cache sizes.

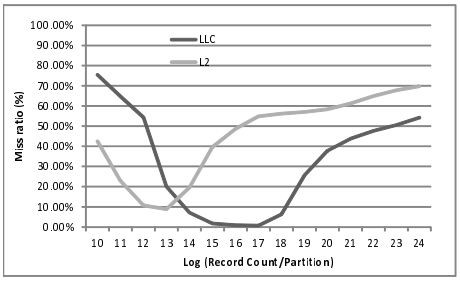

Figure 3. Cache Miss Rates of Partition Sort.

As shown in Fig. 2, we find sorting time increases with the increase of records per partition. For 20MB LLC, sorting performance declines more quickly when chunk size exceeds LLC capacity. It is the same as that with 4MB LLC. As shown in Fig. 3, when sorting small chunk, more cache references occur at L2 cache and less references at LLC, so the miss rate of LLC is high. When sorting large chunk, more references are at LLC and miss rate of LLC becomes lower if chunk size is less than LLC capacity. When the scale of data set exceeds LLC capacity, miss rate becomes high. Because L2 cache is smaller than LLC, 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 which are more difficult to realize in C or C++. For example, it is difficult to do dynamic optimizations such as lock coarsening, virtual function inlining in C++. But there are some optimization factors essential for performance, and more suitable to run in a native runtime.

The first optimization is compression and decompression. Nearly all 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 are some overheads crossing JNI boundary, especially when decompression speed is fast. And some techniques like lazy decompression, direct operations on compressed data cannot fit in bulk processing.

The second is instrument optimization. Currently Hadoop can use JNI to leverage SSE/SIMD optimization such as CRC checksum. But it is not a general solution and perfor-

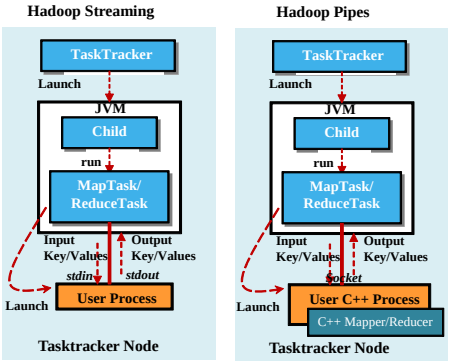

Figure 4. Hadoop Streaming and Pipes.

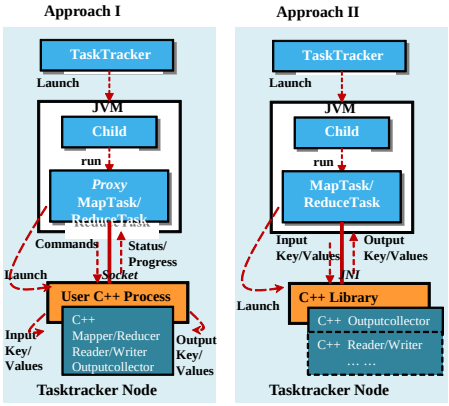

Figure 5. Two kinds of Hadoop exection frameworks.

mance speedup is limited.

The third is compilation. One of our objectives is to provide a native runtime to support high level query execution engine, where some compiler such as LLVM will be used. Because LLVM is a C++ library, so it is more suitable to develop execution engine in C++.

The fourth is memory copy. A lot of serialization and deserialization in Hadoop result in high overhead. In order to achieve maximum throughput, it is better to abandon serialization, or to introduce new serialization method that can operate directly on serialized data, avoiding object creation and memory copy. 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 which includes CRC checksum, decompression, reader, mapper/reducer, writer, compression and CRC checksum, is in 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. Aproach to Compatibility

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

There are two frameworks in Hadoop for multi-language support, Hadoop Streaming [8] and Hadoop Pipes [9], as illustrated in Fig. 4. Hadoop Streaming allows user 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. 5. 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++.

We can implement compatible framework with approach II, but using compatible interfaces will cause performance losses due to cross-language data serialization and memory copy. One of our objectives is to build high level analysis tools or libraries on NativeTask in future. In this case, the compatibility should be constrained in a higher level like query language, while permitting more flexibility in the lower framework. So we design NativeTask to support both two approaches.

NativeTask consists of two major components, java engine and native engine. Java engine is responsible to bypass normal java data flow and delegate the data processing to native side. The actual computations run on native engine. Java and native engine communicate with each other by 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 at all time. In order to make Hadoop patched NativeTask more robust,

we design a mechanism to deal with unknown exceptions. We separate two task jars from MapReduce jar package. When deploying MapReduce system, we put two task jar packages in related path. One is the traditional Hadoop task jar and the other is NativeTask jar. If system runs normally, TaskTracker load NativeTask jar and create task process. When task execution failures exceed a certain limit, system will choose Hadoop task jar to run instead of NativeTask jar, and failure information is still written into 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, framework allow user to change to original Hadoop mode. Users can resubmit their uncompleted jobs with a configuration option or add an option in configuration file of client.

IV. NativeTask

A. Implementation of Task Delegation

We introduce an interface, task delegation 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 job configuration file.

NativeTask supports Java mapper, reducer, reader and writer, which leads to full compatibility with Hadoop applications. In addition to this, task delegation also supports another kind of dataflow, which is involved with native mapper and reducer with native reader and writer. The former is compatible with existing components, e.g. input format, output format, reader, and writer. Key and value pairs are passed to or from native side in batch mode. In latter mode, reading and writing data directly by native reader and writer can yield better performance and flexibility. However input format and output format still exist 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 job configuration 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. A typical application based on framework usually consists of several dynamic libraries, as shown in Fig. 6.

NativeTask uses template to realize a simple equivalent 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 library. The library libnativetask.so is NativeTask runtime and also served as a class library with some predefined

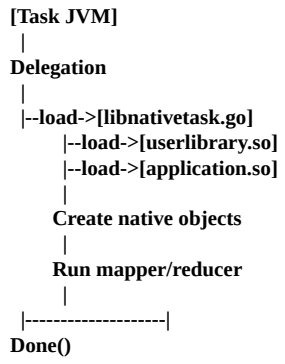

Figure 6. Diagram of dynamic class loading.

mapper, reducer, partitioner, reader and writer. The dataflow and main logics 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 side and native side, the serialized key and value pairs are transferred in a block based pattern rather than a record based pattern in order to decrease JNI calls overhead. The block size is set 32KB to 128KB, which is smaller than L2 cache.

To minimize buffer copy, two light-weight I/O buffers read buffer and append buffer are introduced. Instead of decorator pattern based Java and Hadoop I/O streams, NativeTask focuses on code efficiency, e.g. frequently invoked methods are implemented as inline mode, and meantime we make best effort to avoid buffer copy while supporting compression and decompression. We only uses decorator based stream in batch mode, such as file read, write and CRC checksum. It is easier to add a compression codec in native code, and snappy, lz4 and gzip already have 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 partitioner. Map output collector puts key and value pairs into a partition bucket. Partition bucket has two arrays, blocks vector used by this bucket and offsets vector which maintains the starts and offsets of key and value pairs in memory pool.

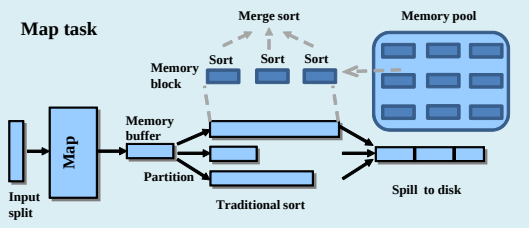

Figure 7. Buffers of map task in NativeTask.

As mentioned above, partition bucket has an array of memory blocks to hold key and value pairs. If memory blocks are used up, another 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 holds the buffer which is set to io.sort.mb, and tracks buffer usage. This buffer only reserves virtual memory instead of RSS. If not actually accessed, the memory will not be allocated in NativeTask instead of the actual memory allocation while initializing arrays in Java. Fig. 7 illustrates memory management and data processing in map task.

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

The block size is involved with cache size and memory usage. Usually the minimum block size equals 32K, and the maximum size equals 1M.

The efficiency will decrease when partition number and key and value size are large. In case that the ratio of io.sort.mb and partition is too small, we can use memory pool directly with disabling memory block.

V. Evaluation

We have conducted extensive evaluation on NativeTask framework. We use HiBench to evaluate typical applications, and show that NativeTask performs more efficiently than Hadoop. We also analyze several factors which affect job performance, including difference between CPU bound and I/O bound workloads, relationship between task and job improvement, the efficiency of JVM reuse and trade-off between partial NativeTask and full NativeTask.

A. CPU-bound and IO-bound Workloads

The experiments with various Hadoop benchmarks and applications were analyzed [10].

Fig. 8 shows high CPU utilization, representing less amounts of I/O examples. Wordcount which counts the number of words in the data listed is an example. This was based on a 1TB file created by Randomtextwriter. The output file size of Wordcount is very small because the final

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
Networks	1GbE
Disk	7 SATA 500GB

Table I
Cluster and Job Configuration

results are aggregated by key. Also, the size of map and reduce output files compared to the size of the initial input was very small, and I/O was rarely occurred. Pagerank is a link analysis method using Hadoop for big data based on PageRank algorithm. Its map is CPU bound and reduce is I/O bound. This is a benchmark machine learning library through Mahout. The number of pages is 500M and total input size is 481GB uncompressed.

Both Hive Join and Aggregation queries are adapted from the query examples in Pavlo et.al [12] and their inputs are defined in [12]. 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 the both the average and sum for each group by joining two different tables. Hive Join is CPU bound workload, and map of Hive Aggregation is CPU bound and reduce 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 cannot improve more efficiently.

Fig. 9 shows several high I/O utilization examples. Accordingly, for I/O bound workloads, there is small gap between the performance of the Hadoop and NativeTask. DFSIO is a test to extract the Hadoop I/O performance and throughput. DFSIO write 1TB data and read 1TB. Sort benchmark is sorts the data in ascending order based on the key. K-Means is a simple but well known algorithm for grouping objects, clustering. Again 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 identify. For K-Means, Iteration stage is CPU bound and classification stage is I/O bound. After the 380GB input data is generated from log, it was classified in five groups. Sort divides binary data into key/value pairs, then sorting them. 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

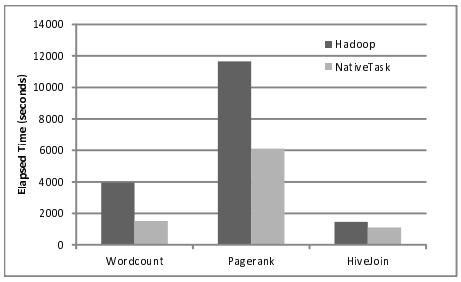

Figure 8. HiBench Workloads (CPU bound).

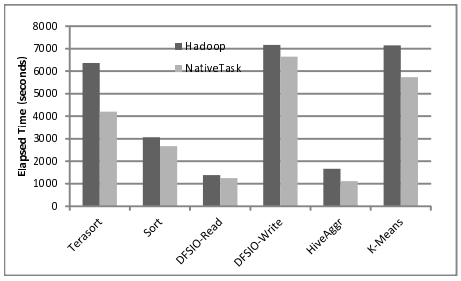

Figure 9. HiBench Workloads (I/O bound).

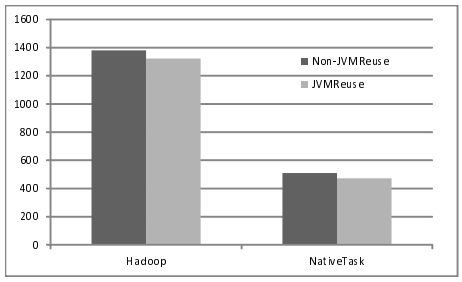

Figure 11. Impact of JVM reuse on Wordcount.

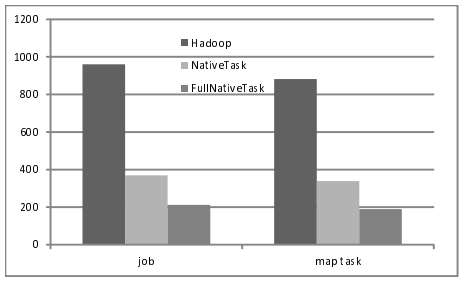

Figure 12. Wordcount performance on NativeTask and FullNativeTask.

writing the list of keys into HDFS. For TeraSort, map is CPU bound and reduce is I/O bound. Compared with 110 minutes running time on Hadoop, it only takes NativeTask 70 minutes to sort 1TB Teradata.

B. Relationship between Task and Job Improvement

In Fig. 10, we compares the improvement factor by NativeTask between job and map tasks for HiBench workloads. For Wordcound, DFSIO-Write and K-Means, improvement of map task almost equal to that of job, because data handled by reduce task is very little or job is map-only. But for K-Means, classification in reduce task takes more time so that NativeTask cannot improve its performance dramatically. For other workloads, map improvements are higher than job improvements, because these jobs have heavy reduce task except for map tasks. For PageRank and TeraSort, map tasks cost lots of CPU of job, so map optimization can improve job dramatically.

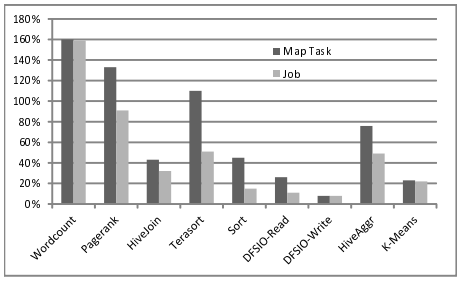

Figure 10. Improvement factor of job and map for HiBench Workloads.

C. Efficient of JVM Reuse

While we enabling JVM reuse, Hadoop can save compiling and launching time, as shown in Fig. 11. We can get 4.5% improvement for Hadoop and 8% improvement for NativeTask, because NativeTask running time is shorter.

D. Full and Patial NativeTask

As mentioned above, we implement the compatible NativeTask and full NativeTask. For latter all task execution components are implemented in C++ space, so latter can get highest performance but lose compatibility. Compared with NativeTask, full NativeTask can run 2x faster further, as shown in Fig. 12. Users can choose compatibility mode or high performance mode by their requirements.

VI. Related Work

There are some research efforts to enhance MapReduce framewok. In [11][12], the authors compared MapReduce with a traditional Parallel Database. The authors also speculate about possible architectural causes for the performance gap between the two systems. For example, record parsing is recognized to introduce performance overhead. [13] suggests that MapReduce users to avoid using text format and prefers protocol buffers for encoding and decoding binary structured records. [14] is similar with our non-sort data flow, but we build these on native side and get higher efficiency. [15] contributes to dynamic adjustment of job configurations rather than optimization of data processing logic .

We also notice the work in [16] published in 2011. This work figures out four factors that affect the performance of data processing, including I/O mode, indexing, parsing

and sorting. Actually it is different with our work. First, it suggests that direct I/O and streaming I/O are helpful. But due to light-weight compression and decompression, instead of I/O, CPU is the bottleneck in real world cluster. Second, it is reasonable to save I/O bandwidth by indexing technology. Third, decoding and encoding optimization in Java is not optimal, some serializations and memory copies are not avoided in Hadoop and reconstructing data flow is necessary. Fouth, it adopts fingerprints comparison strategy reduces the cost of comparing two keys which are fingerprinted to different values. This is useful for some scenario, but not suitable for small key-value which is the common case in real world cluster. Partition based sorting outperforms fingerprints comparison sorting because of optimizing the complexity of the algorithm and cache missing.

VII. Conclusion and Future Work

This paper studies the behavior and barrier of MapReduce task on its open source implementation, Hadoop. We figure out three factors that affect the performance of MapReduce task, especially map task. We investigate alternative optimization strategies for each factor. The papers main insight is that resource utilization of the whole cluster can be improved significantly by optimizing task, and native implementation of task is essential for both lower-level computing framework and upper-level data warehouse.

In this paper, we discussed our experience in designing and implementing NativeTask, which is an open source C++ data processing engine and API for Hadoop. It is completely compatible with Hadoop and supports existing applications to migrate without any modification. Finally, we evaluate the performance of NativeTask by benchmark and real world applications. The experimental results have shown that HiBench workloads can be improved by a factor of 1.3 to 2.6.

In the future, we plan to extend our future research in three areas. First, we will continue contributing to NativeTask for higher efficiency, such as native shuffle, reduce merge and so on. Second, we hope to integrate NativeTask with upper level data warehouse e.g. Hive or Tez for optimizing the whole infrastructure. Third, we want to run NativeTask in multi-core coprocessor environment like GPU or Xeon Phi, in order to explore more parallel possibility of framework.

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] 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.

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

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

[8] 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.

[9] 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.

[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] 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.

[12] 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.

[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.