ShmStreaming: A Shared Memory Approach for Improving Hadoop Streaming Performance

Longbin Lai*, Jingyu Zhou†, Long Zheng‡§, Huakang Li§, Yanchao Lu§, Feilong Tang†, Minyi Guo§
*School of Information Security, Shanghai Jiao Tong University, Shanghai, 200240, China
†School of Software, Shanghai Jiao Tong University, Shanghai, 200240, China
‡School of Computer Science and Engineering, The University of Aizu, Aizu-wakamatsu, 965-8580, Japan
§Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, 200240, China

***Abstract*—The Map-Reduce programming model is now drawing both academic and industrial attentions for processing large data. Hadoop, one of the most popular implementations of the model, has been widely adopted. To support application programs written in languages other than Java, Hadoop introduces a streaming mechanism that allows it to communicate with external programs through pipes. Because of the added overhead associated with pipes and context switches, the performance of Hadoop streaming is significantly worse than native Hadoop jobs.**

We propose ShmStreaming, a mechanism that takes advantages of shared memory to realize Hadoop streaming for better performance. Specifically, ShmStreaming uses shared memory to implement a lockless FIFO queue that connects Hadoop and external programs. To further reduce the number of context switches, the FIFO queue adopts a batching technique to allow multiple key-value pairs to be processed together. For typical benchmarks of word count, grep and inverted index, experimental results show $20-30\%$ performance improvement comparing to the native Hadoop streaming implementation.

***Keywords*-Hadoop Streaming; Map-Reduce; Shared Memory;**

I. Introduction

In the era of big data [2], [9], Map-Reduce [4] has been widely used as the distributed programming model for data-intensive applications. Hadoop is one of the most mature and popular open-source Map-Reduce platform for data-intensive applications. To support legacy programs written in other languages, Hadoop provides a streaming mechanism, which allows mappers and reducers to be external programs. Because the data exchange of Hadoop streaming uses pipes, the performance of Hadoop streaming is significantly reduced [5].

Hadoop C++ Extension (HCE) [17] addressed the bottleneck of Hadoop Streaming by moving all map and reduce tasks to external C++ implementations. In HCE, Hadoop initiates the map-reduce tasks and then notifies C++ routines to actually launch map-reduce jobs written in C++ via sockets. Compared to Hadoop streaming, which only accomplishes the mapping and reducing portions, HCE is actually a reimplementation of Hadoop's Map-Reduce framework in C++. HCE only supports jobs written in C and C++ and introduces a large number of modifications to Hadoop.

In this paper, we design and implement **ShmStreaming**, a prototype utility, to improve the performance of Hadoop streaming with minimum changes to both Hadoop and streaming programs. We first analyze the performance overhead of the pipe mechanism used by Hadoop streaming to exchange `<key, value>` pairs between Hadoop and external processes, and find out it is the pipe that decreases overall performances. Furthermore, our micro benchmark results show that the system call overhead, locking, and address space checking are the main bottlenecks for pipes. To reduce the overhead associated with system calls, we adopt shared memory to transfer data pairs between Hadoop and external processes. An efficient queue is designed to guarantee exclusive access to the shared memory, and deal with synchronization issues within the involving processes. We have implemented extra interfaces in the Hadoop Streaming package in order to provide capabilities for Hadoop's Java routines to access the shared memory created by standard Linux IPC functions. The current ShmStreaming is implemented for C and C++. Our experimental results on three typical benchmarks show that the proposed ShmStreaming perform 20-30% better than the native Hadoop streaming.

Currently, we have not extended the implementation to languages other than C and C++. However, we note that shared memory operations are also supported by many other languages. For instance, Python [12], and Perl [16] both have official binding or customized SYSV IPC support. As a result, we see no technical difficulty in applying our approach to other languages. Furthermore, this approach can also be adopted to other settings besides Hadoop where there are interactions between Java and other languages. One latent drawback may be that our implementations can only be adopted when user has full access to the source code of the external program in order to change it and to use the shared memory.

This paper is organized as follows. Section II discusses the background of Hadoop streaming. Motivations are illustrated in Section III, in which we further explore the bottleneck of pipes. Section IV covers the overall design and the implementation of ShmStreaming. Section V evaluates the performance of ShmStreaming with typical applications.

Section VI discusses related work. Finally, Section VII concludes with paper.

II. Background

Hadoop Streaming is a set of extra utilities provided by Hadoop for developing applications in languages other than Java, such as C, C++, Python, Perl, and UNIX shells, which makes it easy to adapt legacy applications into a MapReduce-style execution model. Additionally, Hadoop Streaming provides alternatives for programmers who are not quite familiar with Java to develop Hadoop applications. Specifically, a streaming job launches external programs and communicates with external processes via two pipes, `clientIn_` and `clientOut_`, to accept data from external processes, and to send data outside, respectively.

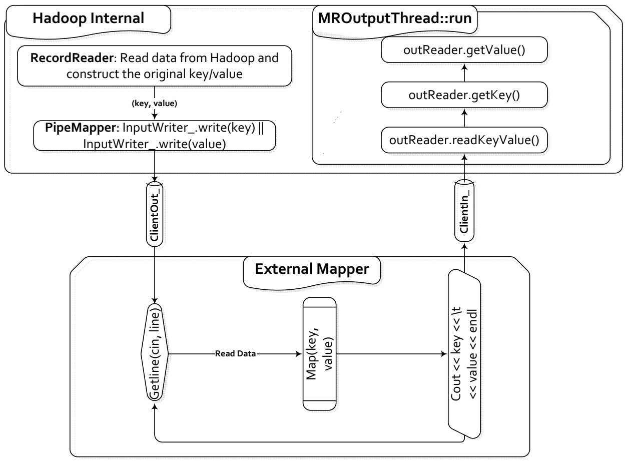


Figure 1. The mapper part of Hadoop Streaming.

Figure 1 illustrates the mapper part of Hadoop streaming. `PipeMapper` of the Hadoop streaming interface calls a map function to fetch original data records and transform them into `<key, value>` pairs, which are sent out to an external mapper via `clientOut_`. The external mapper simply fetches the pairs from standard input, processes the pairs sequentially, and emits output `<key, value>` pairs to `clientIn_`. The reduce part of Hadoop streaming works in a similar way.

Though flexible, Hadoop Streaming is often criticized for its poor performance. Ding et al. [5] have attributed the performance bottleneck to the pipe mechanism. They observed that the streaming performance becomes worse with increasing size of input data. While for computation-intensive tasks that don't have much data exchange between Hadoop and external processes, streaming performance was not so unacceptable, and even a bit better than the native Hadoop. However, they do not give a detailed treatment of why pipes can reduce the performance so remarkably. As one of our motivations in designing ShmStreaming, we will discuss this issue in the next section.

III. Motivation

Hadoop mainly targets data-intensive work. In a single streaming job, the number of `read` and `write` system calls to pipes is proportional to the number of `<key, value>` pairs, i.e., total data size divided by the average size of `<key, value>` pairs. Because the input data size is usually in the order of GB and TB (or even up to PB), and the average size of `<key, value>` pairs in typical applications is less than a few hundreds of bytes, the total number of `<key, value>` pairs becomes extremely large. As a result, Hadoop Streaming jobs issue a large number of `read` and `write` calls, which dramatically drag down the performance.

To measure the overhead of `read` and `write` system calls, we designed a micro-benchmark that works in a tight loop of writing 4 KB of data to a character device and then reading 4 KB data back. The device driver in the kernel simulates a UNIX pipe by maintaining an internal 4 KB buffer. The execution time is divided into three parts: system call overhead, memory copy, and the rest of the execution time. Let the execution time of `read` or `write` system call be t_u and the execution time of the corresponding handler of the device driver be t_k. The system call overhead is $t_u - t_k$, which includes the time for setting up system call parameters, switching from user mode to kernel mode, finding and calling the corresponding handler function, switching back to user mode, and returning results. t_k is composed of memory copy and some bookkeeping code. To be accurate, we use CPU ticks obtained via `rdtsc()` to determine the executing times.

The average results of 50,000 operations are shown in Figure 2. We found that the actual work of memory copy only accounts for about 10% of time, and the overhead of system calls takes up nearly half of all the execution time. Locking the kernel buffer, checking the validity of user space memory, and unlocking are responsible for 42% of the time. Results for `write` exhibit a similar pattern.

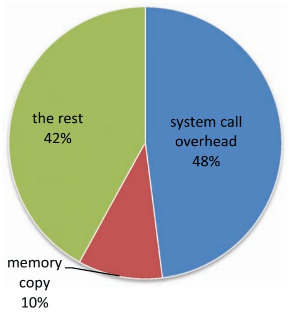


Figure 2. The breakdown of `read` system call time. Experiments are performed on a machine with Intel Core i5 2.3 GHz CPU, 4 GB memory.

To summarize, the `read` and `write` system calls have an overhead that is many times the cost of the real useful work, i.e., memory copy. For applications using Hadoop Streaming, this overhead aggregated over a large number of system calls can have significantly adverse impact on performance. To address this problem, we designed and implemented a shared-memory approach for Hadoop Streaming, thus avoiding the overhead associated with system calls.

IV. Design and Implementation

A. Design Requirement

We have identified the following requirements for our ShmStreaming approach.

- The changes to Hadoop source code should be minimal. This is to avoid introducing bugs into Hadoop and to make it easy for porting ShmStreaming to new releases of Hadoop.
- The modifications of external programs should also be minimal so that ShmStreaming can be easily adopted.
- ShmStreaming should be able to support different programming languages, such as C, C++, Python, Perl, and the UNIX shell.

B. ShmStreaming Interface

We have designed the interface for both Hadoop and external programs to use our ShmStreaming mechanism.

For Hadoop, programmers only need to change the job configuration file to specify using shared memory for streaming, i.e., set `stream.map(reduce).input=shm`, and `stream.map(reduce).output=shm`.

For external programs, instead of interacting with `stdin` and `stdout`, the programs need to instantiate a `SMStream` object and perform read and write operations with the object. Figure 3 illustrates the interface of class `SMStream`.

```
class SMStream {
public:
    // Initialize shared memory with buffer size
    SMStream(int bufSize=4096);

    ~SMStream();

    // write a buffer to shared memory
    int write(char *buf, int size);

    // read data from shared memory into a buffer
    int read(char *buf, int size);
};
```

Figure 3. SMStream interface.

C. Synchronization between Reader and Writer

Like many shared-memory applications, ShmStreaming needs to synchronize data accesses to shared-memory regions between Hadoop and external processes. Typically, critical areas like shared memory are guarded by locks, e.g., using semaphores. However, semaphore operations involve system calls and context switches. Because each access to shared memory needs to be protected with a lock and an unlock operations, there are twice as many system calls for semaphore operations as the number of accesses. Recall that each access only reads or writes one `<key, value>` pair. The performance overhead due to system calls is even larger than the pipe implementation.

By taking a closer look at streaming jobs, we find that each streaming communication follows a single-reader-single-write (SRSW) model, in which either Hadoop acts as a writer and outputs data to an external process, or vice versa. The FIFO queue [15], [3] is a natural fit for the SRSW model, as only two integer variables are needed to separately record the positions of the writer and the reader. Once the writer has pushed n bytes of data into the FIFO, one pointer moves forward by n. Similarly, the other pointer moves when the reader finishes a read operation. No locking is required here.

Algorithm 1 read and write using busy wait

```
 1: function READ(buf, n)
 2:     while $n > 0$ do
 3:         $read \leftarrow$ readFIFO($buf, n$)
 4:         $n \leftarrow n - read$;
 5:     end while
 6: end function
 7:
 8: function WRITE(buf, n)
 9:     while $n > 0$ do
10:         $written \leftarrow$ writeFIFO($buf, n$)
11:         $n \leftarrow n - written$;
12:     end while
13: end function
```

Although FIFO emancipates us from locks, synchronization is still required to notify the reader and the writer about the conditions of the buffer. The reader cannot receive any valid data until the buffer is not empty. Meanwhile, the writer will be blocked once the buffer is full. Algorithm 1 illustrates strawman implementations of `read` and `write` using busy wait. `readFIFO` and `writeFIFO` functions attempt to read or write certain bytes of data from or to the FIFO, and return the actual bytes completed. If the buffer is empty, the `readFIFO` returns zeros and the read operation will be retried immediately in a busy loop. The write operation is also in a busy loop when the buffer is full. Busy wait can be relatively efficient when reader and writer synchronize with each other well so that both empty and full buffer rarely happen. However, in Hadoop Streaming, external programs are usually much slower to process `<key, value>` pairs than Hadoop. As a result, the Hadoop mapper spends much time busy-waiting on writes, and the reducer frequently waits on reads, significantly reducing the performance.

To avoid busy waiting, we need a mechanism for blocking and later resuming execution. Whenever there is nothing to read, the reader is blocked until the writer has pushed some data into the FIFO. Conversely, the writer is blocked when the FIFO is full and resumes execution when the reader has consumed some data from the FIFO. A semaphore can be used here to synchronize readers and writers. However, because external mappers are usually slower than Hadoop, the FIFO buffer will frequently be in the full state, and the writer will issue many semaphore wait calls to block itself, which incurs high overhead. The reducing phase follows the same pattern, blocking the reader due to an empty buffer.

Algorithm 2 Read and write using semaphores to synchronize and batching to reduce overhead.

```
 1: // Global variables
 2: int $batch\_size = CONSTANT$
 3: semaphore $sem\_full(0)$, $sem\_empty(0)$
 4: // flag of FIFO's status
 5: bool $empty = 1$, $full = 0$
 6: // # of FIFO's is full/empty
 7: int $times\_full = 0$, $times\_empty = 0$
 8: // functions testing whether the FIFO is empty or full
 9: is_empty(), is_full()
10:
11: function READ_WAIT(buf, n)
12:     if not $compare\_and\_swap(empty, 0, is\_empty())$ then
13:         if $compare\_and\_swap(full, 1, 0)$ then
14:             $times\_full \leftarrow 0$
15:             $sem\_post(sem\_full)$ // wake up writer
16:         end if
17:         $sem\_wait(sem\_empty)$ // wait for writer
18:     end if
19:     $READ(buf, n)$
20:     if $full$ then
21:         $times\_full \leftarrow times\_full + 1$
22:         if $times\_full > batch\_size$ then
23:             $times\_full \leftarrow 0$
24:             $sem\_post(sem\_full)$
25:         end if
26:     end if
27: end function
28:
29: function WRITE_WAIT(buf, n)
30:     if not $compare\_and\_swap(full, 0, is\_full())$ then
31:         if $compare\_and\_swap(empty, 1, 0)$ then
32:             $times\_empty \leftarrow 0$
33:             $sem\_post(sem\_empty)$ // wake up reader
34:         end if
35:         $sem\_wait(sem\_full)$ // wait for reader
36:     end if
37:     $WRITE(buf, n)$
38:     if $empty$ then
39:         $times\_empty \leftarrow times\_empty + 1$
40:         if $times\_empty > batch\_size$ then
41:             $times\_empty \leftarrow 0$
42:             $sem\_post(sem\_empty)$
43:         end if
44:     end if
45: end function
```

Algorithm 2 presents an improved version of `read` and `write` functions using semaphores to synchronize readers and writers. More importantly, this algorithm batches reads and writes to reduce the number of semaphore calls.

The batching size is controlled by variable $batch\_size$ that can be configured by users. In practice, we recommend setting $batch\_size$ to be slightly less than $\frac{buffer\_size}{record\_size}$, where $record\_size$ is the estimated size of `<key, value>` pairs. This is to prevent the size of a batch from exceeding the total buffer size. Our evaluation in Section V-C shows that such a setting for $batch\_size$ yields the best performance.

Flags $empty$ and $full$ represent the empty and full status of the shared FIFO buffer, and $times\_empty$ and $times\_full$ serve as counters for the times that the writer/reader executes when the buffer is empty or full. These parameters are stored in shared memory to be accessed by both the reader and writer.

When the FIFO is empty, the $is\_empty()$ at line 12 returns one, the $compare\_and\_swap$ function assigns one to $empty$ and enters the *if* block from line 13 to 17, causing the reader to be blocked at line 17. The reader will not be woken up by the writer until $times\_empty$ exceeds the threshold of $batch\_size$ (line 38 to 44). Once woken up, the reader will have a number of `<key, value>` pairs to consume and can avoid immediately blocking after processing one pair. The blocking and waking up of the writer work in a similar fashion. In this way, the number of system calls can be greatly reduced. Note that we use the $compare\_and\_swap$ atomic function at line 12, 13, 30 and 31 to protect the modification of the FIFO-status parameters. Otherwise, there will be risks of deadlock.

Take line 12 as an example: the execution will be separated into two steps, first to judge whether the FIFO is empty, and then to assign values to the $empty$ variable. Suppose initially FIFO is empty and $empty$ has not been set to one yet. Just before setting the $empty$ variable to one, the operating system switches the process from the reader to the writer, leaving the $empty$ variable unset. Then the writer checks the $empty$ variable, and since the variable has not been set, the writer completes the operation. The writer continues writing until blocking itself due to a full FIFO. Finally, when the reader resumes execution to set the $empty$ variable and then conducts the P operation at line 17, both the reader and the writer become blocked, i.e., a deadlock condition. By using the $compare\_and\_swap$ atomic functions, we can avoid such deadlock scenarios.

D. Implementation

We have implemented the Hadoop Streaming interface with shared memory operations discussed above. Specifically, we have added three new classes to Hadoop and made changes to `PipeMapRed`, `PipeMapper`, and `PipeReducer`. We are careful to make the changes small — a few dozen lines. The new classes include `ShmInputWriter` and `ShmOutputReader`, implementing interfaces of `InputWriter` and

`OutputReader`, respectively. As a result, programmers can use our approach by only changing a configuration file.

Because Java does not directly support shared memory operations, we use Java Native Interfaces (JNI) to invoke shared memory operations and copy memory from shared memory to Java, and vice versa. With native library in the `HADOOP_HOME` directory, Hadoop automatically loads the library during startup.

For external programs, we implement a new class `SMStream` that provides a `read` and a `write` interface, similar to the system call interface. Typically, applications only need to change a few lines of code to take advantages of ShmStreaming.

V. Evaluation

A. Experiment Setup

All experiments are conducted in a cluster of eight nodes. Among them, one node is configured as namenode and the others are datanodes. Table I describes the detailed hardware and software configurations.

Table I
HARDWARE AND SOFTWARE CONFIGURATIONS FOR THE EXPERIMENT CLUSTER.

Item	Configurations
CPU	Intel Xeon E5405, quad-core, 2 GHz
Memory	8 GB
OS	CentOS5.5 64bit, Linux 2.6.18
GCC / G++	4.1.2 (Rat Hat)
Java	OpenJDK (build 1.6.0-b09)

Benchmarks used in the experiments are: (I) **word count**: counting words in input files; (II) **grep**: finding match for specified pattern from input files; (III) **inverted index**: generating inverted index for a bundle of small files. These three benchmarks are typical Map-Reduce applications that generate a large number of `<key, value>` pairs. The input files for all the tests except the $batch\_size$ study section are generated by randomly selecting words from the 10,000 most frequently used English words [18] and composing them into a large file with approximately 10 to 20 words a line. *word count* and *grep* have the same input files with sizes of 4, 6, 8, 10, and 20 GB. Input files for the *inverted index* are a number of text files, and each file has a size of 64 MB, equal to the size of an HDFS (Hadoop File System) block.

During the experiments, $batch\_size$ is set to 64 and $buffer\_size$ of the shared memory is set to 8192 for mapper and 16384 for reducer if not specially mentioned. We control $record\_size$ to guarantee high performance of ShmStreaming during the experiment. Experimental results are the average of 10 runs.

B. Overall Results

This experiments compare the performance of Hadoop, Hadoop Streaming, and ShmStreaming (with the optimal configurations) on three benchmarks. The results are shown in Figure 4. The native Hadoop presents the best performance in all benchmarks, followed by ShmStreaming. Hadoop Streaming performs the worst, with more than 100% overhead for *word count* and *grep*, and over 50% overhead for *inverted index*. ShmStreaming and Hadoop Streaming are slower than native Hadoop due to the extra overhead of the memory copy between Hadoop and external process. ShmStreaming outperforms Hadoop Streaming due to a reduction of context switches as well as an optimization of synchronizations.

The performance of ShmStreaming lies between native Hadoop and Hadoop Streaming. Compared to Hadoop Streaming, ShmStreaming improves performance by nearly 24% for *word count*, 24% for *grep*, and 22% for *inverted index*. This is mainly because ShmStreaming uses shared memory to avoid the extra cost of system calls.

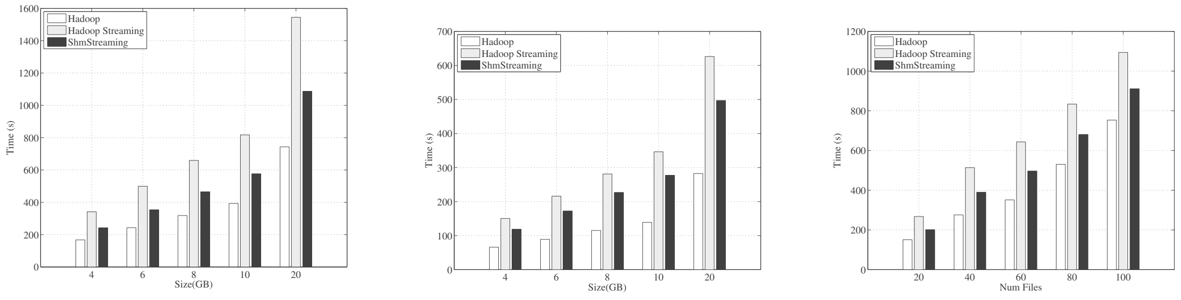
Figure 5 illustrates the extra overhead of Hadoop Streaming and ShmStreaming relative to native Hadoop with different input sizes. The extra overhead is approximately proportional to the size of the input. As a result, ShmStreaming achieves a higher performance improvement in absolute values as the input size increases.

C. Study on $batch\_size$

This experiment studies the impact of $batch\_size$ on the performance of ShmStreaming. When $batch\_size$ is one, ShmStreaming is similar to kernel pipes, which lock the buffer for each read or write call. When $batch\_size$ is larger, ShmStreaming allows reader or writer to perform several operations without being blocked.

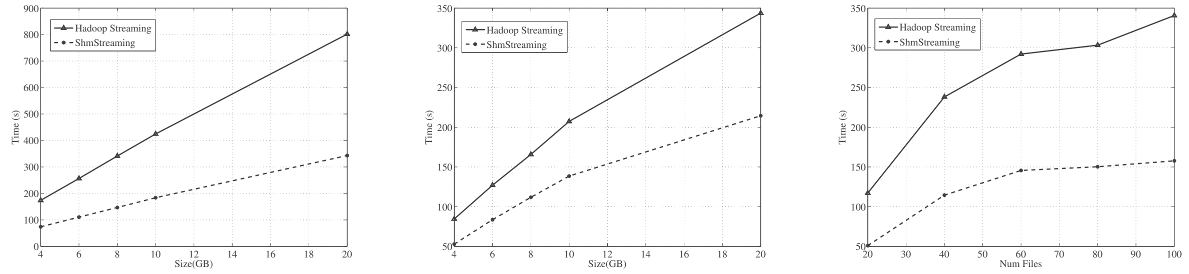
In this experiment, we focus on the mapper stage for *word count*. The $buffer\_size$ is set to 8192 and the test files (4GB each) are specifically generated with a fixed line size of 256 bytes. We start with an experiment using test files with fixed word size as 2, 4, 8, 16, 32, 64, respectively.

Figure 6(a) illustrates the performance improvement of ShmStreaming over Hadoop Streaming when changing $batch\_size$ with different $record\_size$. We can observe that when $batch\_size$ is 1 and 2, there is almost no improvement, because the overhead of JNI dominates the performance savings of shared memory. We can also observe the following pattern from Figure 6(a): the performance starts to drop significantly when $batch\_size$ is $\frac{buffer\_size}{record\_size}$. For example, when the $record\_size$ is 2, the performance starts to drop when $batch\_size$ is 4096, and when the $record\_size$ changes to 64, the performance decreases when $batch\_size$ is larger than 128. This is because when $batch\_size$ is larger, the other end of the shared memory buffer is forced to wait for some time, even though it may access the buffer. As a result, the overlapping period of reads and writes is greatly decreased, resulting in lower performance. The performance starts to drop earlier than $\frac{buffer\_size}{record\_size}$ is due to the fact that $record\_size$ is actually a little bit larger than word size,



(a) word count (b) grep (c) inverted index

Figure 4. Results of executing time for native Hadoop, Hadoop Streaming, and ShmStreaming on three benchmarks.



(a) word count (b) grep (c) inverted index

Figure 5. Extra overhead (lag) comparing to native Hadoop for Hadoop Streaming and ShmStreaming.

because each `<key, value>` pair contains an integer as the value.

We performed similar experiments on *grep* and *inverted index* with the same observations as *word count*. Figure 6(b) shows similar pattern for all the three benchmarks when the $record\_size$ is set as 16.

Table II
PERFORMANCE IMPROVEMENT WITH FIXED AND VARIED $record\_size$.

$batch\_size$	2	64	128	256	512
$record\_size$(fixed 16)	9.87	18.06	17.71	**17.99**	6.41
$record\_size$(varied 16.32)	9.49	19.14	18.12	18.95	**18.41**
$record\_size$(fixed 32)	6.17	18.48	**18.77**	8.67	6.26
$record\_size$(varied 32.32)	7.99	18.36	18.49	**18.79**	6.00
$record\_size$(fixed 64)	7.92	**18.68**	8.71	6.21	6.05
$record\_size$(varied 63.98)	5.43	18.9	**18.7**	10.24	8.21

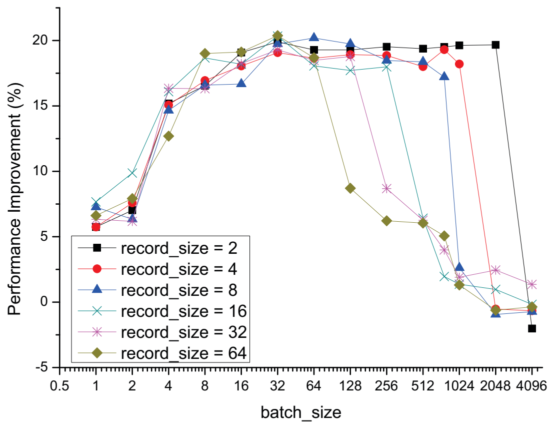
Because the $record\_size$ cannot be a fixed value in practice, we conduct another experiment to study the impact of varied $record\_size$. The test files contain either fixed-size or varied-size words, whose average size is approximately 16, 32 and 64. Table II illustrates the experimental results, the data represents the percentage of performance improvement, and the bold items represent the $batch\_size$ after which the performance drops. We can observe that the average $record\_size$ still dominates the optimal configurations of $batch\_size$ in all cases. For the varied-size cases, the performance always starts to drop at a larger size than that of the fixed-size cases. This experiment demonstrates that our approach can improve the performance for cases when word size varies.

D. Impact of $buffer\_size$

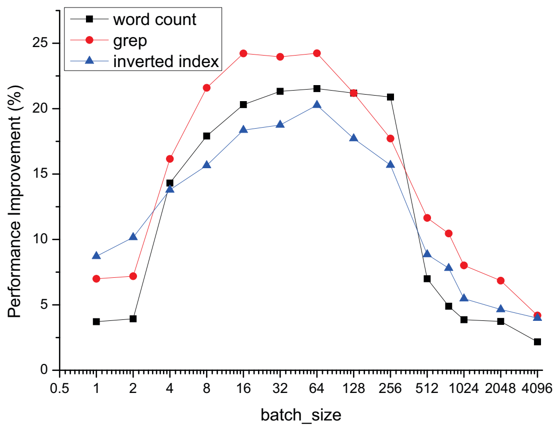
This experiment evaluates the impact of $buffer\_size$, which is one of the most significant configurations in this work. The test file is the same as the general test, and $batch\_size$ always guarantees optimal performance. As shown in Table III, the performance of all three benchmarks is not sensitive to the variation of $buffer\_size$. Note that $buffer\_size$ exceeding 16,384 is not supported in current prototype implementation of ShmStreaming, because the JNI interface we adopted fails to allocate a shared memory with size larger than 16,384. We expect our approach to work well with larger sizes of shared memory.

Table III
THE IMPACT OF $buffer\_size$.

$buffer\_size$	4096	8192	12288	16384	≥20480
word count	18.40	18.24	18.39	18.83	N/A
grep	22.28	22.40	22.61	22.15	N/A
inverted index	18.30	18.72	18.51	18.17	N/A



(a) *batch_size* and *record_size* for word count.



(b) *batch_size* and *record_size* for three benchmarks

Figure 6. The performance improvement of ShmStreaming on configuration of *batch_size* varied from 1 to 4096 and *buffer_size* is set to 8192.

Table IV
THE PERFORMANCE COMPARISON OF DIFFERENT CONFIGURATIONS FOR SHMSTREAMING WITH *word count* USING A FILE OF 4 GB SIZE.

Config.	Performance Improvement(%)
busy wait	5.96
$batch\_size = 1$	5.14
$batch\_size = 64$ (best)	22.5

E. Comparison with Busy Wait

This experiment compares the performance of ShmStreaming using busy wait and batching. Table IV illustrates the performance improvement over Hadoop Streaming. We can observe that the performance of busy wait is similar to the configuration where *batch_size* is one (note which is the situation of simple lock with batching), slightly more than 5% improvement, and both perform far from the optimal of 22.5%. For busy wait, the lower performance is because readers and writers are not synchronized. In addition, the busy waiting causes high CPU utilization, which is harmful for other jobs running on the same node. For small batch size, it is because of the JNI overhead. The default ShmStreaming chooses the right value for *batch_size*, batching read and write operations as well as reducing the number of system calls.

VI. RELATED WORK

The drawbacks of Hadoop Streaming have been addressed by Baidu Inc. with an implementation of HCE [17], which implements much of Hadoop's functionality in C++, such as reading input stream, mapping `<key, value>` pairs, shuffling, sorting, and reducing final `<key, value>` pairs. Performance results indicate that HCE achieves significant improvement comparing to Hadoop streaming. Our work is different from HCE in three aspects. First, HCE avoids data copies with C++ implementations of input stream readers and mappers (or reducers), while our approach uses shared memory for data exchange between Hadoop and external programs. Second, HCE makes a large number of changes to Hadoop, while our modifications are very limited. Finally, HCE only supports C++, while ShmStreaming can be used by programs written in other languages.

Historically, critical system calls like `read` and `write` have been considered as vital factors that would potentially make differences to the performance of overall systems. Zodok et al. [19] witness an overall performance lag of data copying to user-level processes by almost two orders of magnitude for data-intensive operations. The performance issues on system calls have been witnessed in many studies. Soares and Stumm [13] point out the application needs to wait for the completion of system calls due to a synchronous execution model enforced in most common operating systems, which results in performance inefficiencies. Zhao [20] anatomizes the call stack of `read` system call in the EXT2 file system and finds that the call goes through over seven logic levels in order to finally access the data. During the process, at least two context switches happen between user and kernel space. For Hadoop streaming, we find that system calls and context switches introduce a large overhead for data-intensive applications.

Researchers have studied methods of implementing concurrent data structures without the use of mutual exclusion [15], [1], [8]. Recently, lock-free or even wait-free techniques have been introduced into multicore or parallel environments [10], [7], [11], [14], [6]. Many of the lock-free algorithms are complicated in consideration of the asynchronized context and non-linear structures. Our approach uses a lock-free FIFO queue, with the difference of employing a batching mechanism for performance improvement.

VII. Conclusions

The pipe mechanism used by Hadoop Streaming incurs high performance overhead while dealing with data-intensive jobs. We discover that system calls, i.e., `read` and `write` calls, account for most of the overhead of Hadoop Streaming. To address this problem, we introduce ShmStreaming that employs shared memory for data exchange between Hadoop and external programs. With the minimum modifications to Hadoop and the streaming applications (when the source code is available), ShmStreaming achieves a $20-30\%$ performance improvement over the native Hadoop Streaming for benchmarks of word count, grep, and inverted index.

Our prototype implementation can be further improved. We notice that except system calls, the overhead of memory copy is also a major part during the interprocess communication. Thus, we may improve performance by reducing the number of memory copies. Finally, it would be interesting to study the relationship between $batch\_size$ and $record\_size$.

Acknowledgment

We thank anonymous referees for their helpful comments. This work was supported in part by National Natural Science Foundation of China (NSFC) (Grant No. 61003012, 61261160502) and 863 programs of China (Grant No. 2011AA01A202). This work was as well supported by Japan Society for the Promotion of Science (JSPS). This work was also supported by the NSFC with Grant Nos. 61272442 and 61073148, and Key Basic Research Project of the Science and Technology Commission of Shanghai Municipality with Grant No. 12JC1405400.

References

[1] G. Barnes. A method for implementing lock-free shared-data structures. In *Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures*, pages 261–270. ACM, 1993.

[2] G. Bell, J. Gray, and A. Szalay. Petascale computational systems. *Computer*, 39(1):110–112, 2006.

[3] D. Bovet and M. Cesati. *Understanding the Linux kernel*. O'Reilly Media, 2006.

[4] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI'04: Sixth Symposium on Operating System Design and Implementation*, San Francisco, CA, 2004.

[5] M. Ding, L. Zheng, Y. Lu, L. Li, S. Guo, and M. Guo. More convenient more overhead: the performance evaluation of hadoop streaming. In *Proceedings of the 2011 ACM Symposium on Research in Applied Computation*, RACS '11, pages 307–313, New York, NY, USA, 2011. ACM.

[6] J. Giacomoni, T. Moseley, and M. Vachharajani. Fastforward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 43–52. ACM, 2008.

[7] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 206–215. ACM, 2004.

[8] M. Herlihy and J. Moss. Transactional memory: Architectural support for lock-free data structures. *ACM SIGARCH computer architecture news*, 21(2):289–300, 1993.

[9] P. Lyman and H. R. Varia. How much information. http://www.sims.berkeley.edu/how-much-info-2003, 2003.

[10] M. Michael and M. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *Journal of Parallel and Distributed Computing*, 51(1):1–26, 1998.

[11] M. Moir, D. Nussbaum, O. Shalev, and N. Shavit. Using elimination to implement scalable and lock-free fifo queues. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 253–262. ACM, 2005.

[12] P. Semanchuk. System V IPC for Python - Semaphores, Shared Memory and Message Queues. http://semanchuk.com/philip/sysv_ipc/, Oct 2010.

[13] L. Soares and M. Stumm. Flexsc: Flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, pages 1–8. USENIX Association, 2010.

[14] H. Sundell and P. Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. *Journal of Parallel and Distributed Computing*, 65(5):609–627, 2005.

[15] J. Valois. Implementing lock-free queues. In *Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems*, pages 64–69, 1994.

[16] L. Wall, T. Christiansen, and J. Orwant. *Programming perl*. O'Reilly Media, 2000.

[17] S. Wang. Hadoop c++ extention. https://issues.apache.org/jira/browse/MAPREDUCE-1270, Dec 2009.

[18] WIKIPEDIA. Wiktionary:frequency lists. http://en.wiktionary.org/wiki/Wiktionary:Frequency_lists.

[19] E. Zadok, I. Badulescu, and A. Shender. Extending file systems using stackable templates. In *Proceedings of the Annual USENIX Technical Conference*, pages 57–70, 1999.

[20] J. Zhao. Analysis through read system call. http://www.ibm.com/developerworks/cn/linux/l-cn-read/, March 2008.