Tiled-MapReduce: Efficient and Flexible MapReduce Processing on Multicore with Tiling

RONG CHEN and HAIBO CHEN, Shanghai Jiao Tong University

The prevalence of chip multiprocessors opens opportunities of running data-parallel applications originally in clusters on a single machine with many cores. MapReduce, a simple and elegant programming model to program large-scale clusters, has recently been shown a promising alternative to harness the multicore platform.

The differences such as memory hierarchy and communication patterns between clusters and multicore platforms raise new challenges to design and implement an efficient MapReduce system on multicore. This article argues that it is more efficient for MapReduce to iteratively process small chunks of data in turn than processing a large chunk of data at a time on shared memory multicore platforms. Based on the argument, we extend the general MapReduce programming model with a "tiling strategy", called *Tiled-MapReduce* (TMR). TMR partitions a large MapReduce job into a number of small subjobs and iteratively processes one subjob at a time with efficient use of resources; TMR finally merges the results of all subjobs for output. Based on Tiled-MapReduce, we design and implement several optimizing techniques targeting multicore, including the reuse of the input buffer among subjobs, a NUCA/NUMA-aware scheduler, and pipelining a subjob's reduce phase with the successive subjob's map phase, to optimize the memory, cache, and CPU resources accordingly. Further, we demonstrate that Tiled-MapReduce supports fine-grained fault tolerance and enables several usage scenarios such as online and incremental computing on multicore machines.

Performance evaluation with our prototype system called Ostrich on a 48-core machine shows that Ostrich saves up to 87.6% memory, causes less cache misses, and makes more efficient use of CPU cores, resulting in a speedup ranging from 1.86x to 3.07x over Phoenix. Ostrich also efficiently supports fine-grained fault tolerance, online, and incremental computing with small performance penalty.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Frameworks*

General Terms: Design, Performance

Additional Key Words and Phrases: MapReduce, Tiled-MapReduce, tiling, multicore

ACM Reference Format:
Chen, R. and Chen, H. 2013. Tiled-MapReduce: Efficient and flexible mapreduce processing on multicore with tiling. *ACM Trans. Archit. Code Optim.* 10, 1, Article 3 (April 2013), 30 pages.
DOI: http://dx.doi.org/10.1145/2445572.2445575

This article is an extension of an earlier version appearing in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques* [Chen et al. 2010].
This article is supported in part by China National Natural Science Foundation under grant no. 61003002, a grant from Shanghai Science and Technology Development Funds (no. 12QA1401700) and Fundamental Research Funds for the Central Universities of China.
Authors' address: R. Chen and H. Chen (corresponding author), Institute of Parallel and Distributed Systems, School of Software, Shanghai Jiao Tong University, Shanghai 200240, China; email: haibochen@sjtu.edu.cn.

DOI: http://dx.doi.org/10.1145/2445572.2445575

1. INTRODUCTION

Multicore or many-core processors, as another form embodying Moore's Law, are commercially prevalent recently. With the commercial availability of twelve cores on a chip, it is foreseeable that hundreds (or even thousands) of cores on a single chip will appear in the near future [Borkar 2007]. With the continuously increasing number of cores, it is vitally important to fully harness the abundant computing resources with still ease-to-use programming models.

MapReduce [Dean and Ghemawat 2008], designed to program large clusters with relatively simple functional primitives, has shown its power in solving nontrivial data-parallel problems such as document clustering, Web access statistics, inverted index, and statistical machine translation. In most cases, programmers only need to implement two interfaces: *Map*, which processes the input data and converts it into a number of key/value pairs; and *Reduce*, which aggregates values in the key/value pairs according to the key. Consequently, MapReduce frees programmers from handling tough tasks such as distributing data, specifying parallelism, and tolerating faults.

While MapReduce was initially designed for clusters, Ranger et al. have demonstrated the feasibility of running MapReduce applications on shared memory multicore machines with Phoenix [Ranger et al. 2007], which is heavily optimized by Yoo et al. [2009][1]. Phoenix uses the pthread library to assign tasks among CPU cores and relies on shared memory to handle inter-task communications. Compared to the cluster version, MapReduce on multicore is able to take advantage of fast inter-task communications with shared memory, thus avoids the expensive network communications among tasks.

Though Phoenix has demonstrated the applicability of running MapReduce on multicore, it is still limited in exploiting many features of commodity multicore systems due to its way of processing all (thus very large) input data at a time. As a result, the input and intermediate data will persist along the entire lifecycle of a processing phase (i.e., Map or Reduce). Hence, a relatively large data-parallel application can easily cause resource pressures on the runtime, operating systems, and the CPU caches, which could significantly degrade the performance. Based on the preceding observation, we argue that it is more efficient to iteratively process small chunks of data in turn than processing a large chunk of data at one time on shared memory multicore platforms, due to the potential of better cache/memory locality and less contentions.

To remedy these problems, this article proposes *Tiled-MapReduce*, which applies the *tiling strategy* [Coleman and McKinley 1995] in compiler optimization, to shorten the lifecycle and limit the footprint of the input and intermediate data, and to optimize the resource usages of the MapReduce runtime and mitigate contentions, thus increasing performance. The basic observation is that the reduce function of many data-parallel applications can be written as commutative and associative, including all 26 MapReduce applications in the test suite of Phoenix [Ranger et al. 2007] and Hadoop [Bialecki et al. 2005]. Based on this observation, Tiled-MapReduce further partitions a big MapReduce job into a number of small subjobs and processes each subjob in turn. The runtime system will finally merge the results of each subjob and generate the final results. In Tiled-MapReduce, the runtime only consumes the resources required for a subjob as well as the output for each subjob, which are usually much smaller than those of processing one big task at a time.

Tiled-MapReduce also enables three optimizations otherwise impossible for MapReduce. First, as each subjob is processed in turn, the data structures and memory

[1]Note that the version of Phoenix we use in this article is 2.0.0 released in May, 2009.

spaces for the input and intermediate data can be reused across the subjob boundaries. This avoids the costs of expensive memory allocation and deallocation, as well as the construction of data structures. Second, processing a small subjob provides the opportunity for fully exploiting the memory hierarchy of a multicore system, resulting in better memory and cache locality. Finally, according to our measurements, the Reduce phase on a multicore machine is usually not balanced, even with dynamic scheduling. This gives us the opportunity to overlap the execution of a subjob's Reduce phase with its successor's Map phase, which can fully harness the CPU cores. Besides, Tiled-MapReduce supports a fine-grained fault-tolerance scheme for multicore that saves the result at the granularity of subjobs, to save the cost of rerunning an entire MapReduce job after the victim machine gets recovered from a whole machine crash.

In addition, the incremental computing nature of Tiled-MapReduce also enables the online MapReduce model [Condie et al. 2010], which supports online aggregation and allows users to see the early results of an online job. It also enables incremental computing on MapReduce that operates on the newly appended data and combines new results with previous results [Popa et al. 2009]. To support these two computing models, Tiled-MapReduce is built with the support to periodically display the intermediate results after a subjob is done, as well as the support for continuous computation that saves the partial results of subjobs for further computation reuse.

It should be noted that Tiled-MapReduce does not require a deep understanding of the underlying MapReduce implementation, thus is orthogonal to a specific MapReduce implementation (e.g., the algorithm and data structures). Tiled-MapReduce also mostly retains the existing programming interfaces of MapReduce, with only two optional interfaces for the purpose of input reuse, which actually have the counterparts in other MapReduce implementations such as Google's MapReduce [Dean and Ghemawat 2008] and Hadoop [Bialecki et al. 2005].

We have implemented a prototype of Tiled-MapReduce based on Phoenix. The system, called Ostrich, outperforms Phoenix due to the mentioned optimizations. Experiments on a 48-core AMD machine using four different types of data-parallel applications (Word Count, Distributed Sort, Log Statistics, and Inverted Index) show that Ostrich can save up to 87.6% memory, causes less cache misses, and makes more efficient use of CPU cores, resulting in a speedup from 1.86x to 3.07x. We also show that the costs of supporting fine-grained fault tolerance, online, and incremental computing are very small.

In summary, this article makes the following contributions.

— An analysis is given wherein iteratively processing small chunks of data is more efficient than processing a large chunk of data for MapReduce on multicore platforms.
— The Tiled-MapReduce programming model extension is given that allows exploiting multicore environments for data-parallel applications.
— Three optimizations are provided that optimize the memory, cache, and CPU usage of the Tiled-MapReduce runtime.
— The enabled fine-grained fault tolerance is given, as well as online and incremental MapReduce computing based on Tiled-MapReduce.

The rest of the article is organized as follows. Section 2 presents the background of MapReduce, the Phoenix implementation, and the "tiling strategy" in parallel computing. Section 3 discusses the possible performance issues with Phoenix and illustrates the design spaces and optimization opportunities of MapReduce on multicore. Section 4 describes the extension from Tiled-MapReduce and its overall execution flow. Section 5 describes the optimization for the resource usage in Tiled-MapReduce. Section 6 describes two computing models based on Tiled-MapReduce, online and

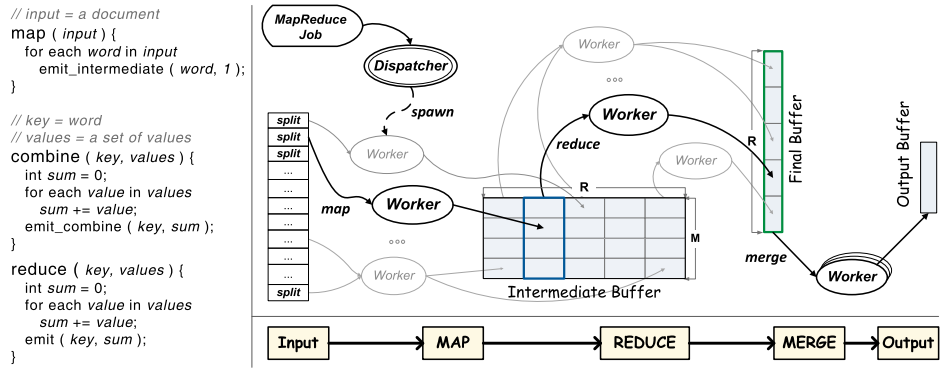

Fig. 1. Pseudocode of Word Count application on MapReduce and the execution flow of the Phoenix Library: the Map workers generate output to the rows of the Intermediate Buffer and the Reduce workers aggregate the columns of the Buffer to generate output to the Final Buffer.

incremental computing. Section 8 relates our work to previous work. Finally, we conclude the article with a brief discussion on future work in Section 9.

2. BACKGROUND

This section presents a short review on the MapReduce programming model, uses Phoenix as an example to illustrate MapReduce for multicore platforms, and briefly describes the tiling strategy in compiler optimization.

2.1. MapReduce Programming Model

The MapReduce [Dean and Ghemawat 2008] programming model mostly only requires programmers to describe the computation using two primitives inspired by functional programming languages, Map and Reduce. The `map` function usually independently processes a portion of the input data and emits multiple intermediate key/value pairs, while the `reduce` function groups all key/value pairs with the same key to a single key/value pair. Additionally, users can provide an optional `combine` function that locally aggregates the intermediate key/value pairs to save networking bandwidth and reduce memory consumption.

The pseudocode in Figure 1 shows the Word Count application counting the number of occurrences of each word in a document. The `map` function emits a $\langle word, 1\rangle$ pair for each *word* in document, and the `reduce` function counts all occurrences of a *word* as the output. The `combine` function is similar to the `reduce` function, but only processes a partial set of key/value pairs.

2.2. The Phoenix Implementation for Multicore

Phoenix [Ranger et al. 2007; Yoo et al. 2009] is an implementation of MapReduce on shared memory multiprocessor systems using the pthread library. It showed that applications written with the MapReduce programming model have competitive scalability and performance with those directly written with the pthread library on a multicore platform.

The lower part of Figure 1 uses a flowchart to illustrate the outline of Phoenix from input to output, which goes through three main phases, including Map, Reduce, and Merge. The right part illustrates the overall execution flow of processing a MapReduce job on the Phoenix runtime. The key data structure of Phoenix runtime is the Intermediate Buffer, which is formed as a matrix of buckets and stores the intermediate data produced in the Map phase and consumed by the Reduce phase. Each row of the buffer

is exclusively used by a worker in the Map phase while each column of the buffer is exclusively used by a worker in the Reduce phase. A MapReduce application starts a job by invoking the dispatcher, which spawns multiple workers and binds them to different CPU cores. In the Map phase, each worker repeatedly splits a piece of input data and processes them using the programmer-supplied `map` function. The `map` function parses the input data and emits multiple intermediate key/value pairs to the corresponding row of Intermediate Buffer. The runtime also invokes the `combine` function (if provided by users) for each worker to perform local reduction at the end of the Map phase. In the Reduce phase, each worker repeatedly selects a reduce task, which sends the intermediate data from the corresponding column of Intermediate Buffer to the programmer-supplied `reduce` function. It processes all values belonging to the same key and generates the final results for a key. In the Merge phase, all results generated by different reduce workers are merged into a single output sorted by key.

2.3. Tiling Strategy in Compiler Optimization

The tiling strategy, also known as *blocking*, is a common technique to efficiently exploit the memory hierarchy. It partitions data to be processed into a number of blocks, computes the partial results of each block, and merges the final results. The tiling strategy is also commonly used in the compiler community to reduce the latency of memory accesses [Coleman and McKinley 1995] and increase the data locality. For example, loop tiling (also known as loop blocking) is usually used to increase the data locality, by partitioning a large loop into smaller ones. Several variations of tiling are also used to optimize many central algorithms in matrix computations [Golub and Van Loan 1996], including the fixed-size tiling, the recursive tiling, and a combination of them.

3. CHALLENGES AND OPPORTUNITIES OF MAPREDUCE ON MULTICORE

This section compares the differences between MapReduce on clusters and that on multicore and discusses possible optimization and enhancing opportunities for MapReduce on multicore.

3.1. Performance and Scalability

Though Phoenix has successfully demonstrated the feasibility of running MapReduce applications on multicore, it also comes with some deficiencies when processing jobs with a relatively large amount of data, which would be common for machines with abundant memory and CPU cores.

This problem is not due to the implementation techniques of Phoenix. In fact, Phoenix has been heavily optimized from three layers: algorithm, implementation, and OS interaction [Yoo et al. 2009], which results in a significant improvement over its initial version. We attribute the performance issues to mainly the programming model of MapReduce on multicore, which process all input data at one time.

First, in cluster environments, the map tasks and reduce tasks are usually executed in different machines and data exchange among tasks is done through networking, compared to shared memory in multicore environments. Hence, the contentions on cache and shared data structures, instead of networking communications, are the major performance bottlenecks for processing large MapReduce jobs on multicore.

Second, there is a strict barrier between the Map and the Reduce phase, which requires the MapReduce runtime to keep all the input and intermediate data through the Map phase. This requires a large amount of resource allocations and comes with a large memory footprint. For relatively large input data, this creates pressure on the memory systems and taxes the operating systems (e.g., memory management), which have imperfect performance scalability on large-scale multicore systems [Song et al. 2011]. Further, it also limits the effects of some optimizations (such as the combiner) due to

restricted cache and memory locality. For example, the combiner interface, which is the key to reduce networking traffic of MapReduce applications in cluster environments, was shown to make little performance improvement along with other optimizations (e.g., prefetching) [Yoo et al. 2009].

Finally, cache and memory accesses in current multicore systems tend to be nonuniform, which makes exploiting the memory hierarchy even more important. Unlike those in cluster environments, MapReduce workers in a multicore platform can be easily controlled in a centralized way, making it possible to control memory and cache accesses in a fine-grained way.

3.2. Fault Tolerance

The failure model supposed by the MapReduce programming model targets at a large cluster of commodity machines, which has a large number of independent machines running MapReduce tasks. Hence, it simply takes a per-task-based fault-tolerance model that restarts a task in a failed machine to a new machine. However, such a failure model cannot be simply applied to MapReduce on multicore platforms.

First, many resources of a commodity multicore platform are universally shared and accessible to MapReduce workers, which makes global failures (e.g., failures resulting in a whole machine crash) more pervasive. Actually, there are currently no separate failure domains on current multicore platforms and a single failure on a CPU core, memory cell, and interconnect may easily crash the entire machine.

Second, compared to cluster environments, many resources of a commodity multicore platform are still relatively restricted, such as CPU and main memory. Therefore, it would be very expensive for multicore platforms to apply the fault-tolerance model based on redundant resources in the cluster platform such as backup tasks for stragglers and active failure detection.

Finally, all intermediate data generated in the Map phase and partial results generated in the Reduce phase are only stored in main memory until the Merge phase produces the final results and stores to persistent storage. Hence, the entire job needs to be reexecuted for memory failures, which is very costly.

3.3. Going Beyond Batch Processing

The MapReduce programming model has become a state-of-the-art for processing large-scale data. However, it is usually restricted to batched tasks. Only supporting batched processing is not appealing for many applications requiring online and incremental processing. Supporting online computation is very important as many usage scenarios require getting the results along with the computation, not waiting for a very long-time until the entire task has been completed. For example, users prefer continuous queries and a rough result for a long-time query to determine the next step, thus the runtime should support returning online reports according to the current progress.

Similarly, as many computing tasks are redundant or partially redundant, it may be very efficient to reuse previous computation results if only a small portion of input has been changed. For example, the real-time fraud detection tool prefers just processing incremental records of the system log, which requires the runtime to support reusing the results of prior computation.

4. TILED-MAPREDUCE

In this section, we describe the Tiled-MapReduce programming model that aims at addressing the challenges with MapReduce on multicore. We first illustrate the extension to the MapReduce programming model and show the execution flow of Tiled-MapReduce, as well as the major changes to runtime to support Tiled-MapReduce.

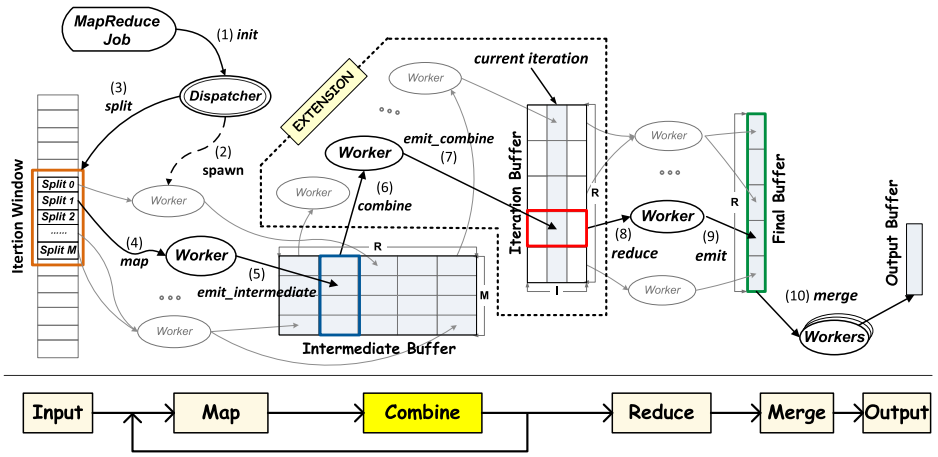

Fig. 2. Execution flow of Tiled-MapReduce.

Finally, we introduce the fine-grained fault-tolerance mechanism adopted in Tiled-MapReduce on multicore platforms.

4.1. Extension to MapReduce Programming Model

Being aware of the challenges and opportunities with MapReduce on multicore, we extend the general MapReduce programming model with Tiled-MapReduce. It uses the "tiling strategy" to decompose a large MapReduce job into a number of independent and small subjobs and iteratively processes one subjob at a time with efficient use of resources, given that the reduce interface for many data-parallel applications can be implemented as commutative and associative.

To support iterative processing of a number of subjobs derived from a large MapReduce job, Tiled-MapReduce replaces the general Map phase with a loop of Map and Reduce phases. In each iteration, Tiled-MapReduce processes a subjob and generates a partial result that can be either saved for computation reuse [Popa et al. 2009] or provided for users to know the status of the computation [Condie et al. 2010]. Tiled-MapReduce also extends the general Reduce phase to process the partial results of all iterations, rather than the intermediate data. The output generated by the Reduce phase is compatible with the output of the general Reduce phase. To differentiate with that in the final Reduce phase, we rename the Reduce phase within one subjob as the Combine phase. The lower part of Figure 2 illustrates the processing phases in Tiled-MapReduce.

Note that it was claimed that the combiner interface made little performance improvements on multicore in the context of the general MapReduce programming model with prefetching of data in the Reduce phase [Yoo et al. 2009]. Tiled-MapReduce mainly uses such an interface for the purpose of shortening the lifecycle of input and intermediate data, which may improve the cache and memory locality and enable further optimizations. This is detailed in Section 5.

4.2. Execution Flow

The top part of Figure 2 illustrates the overall execution flow of a Tiled-MapReduce job and the implementation of Tiled-MapReduce runtime.

A Tiled-MapReduce job is initiated by invoking the `mr_dispatcher` function, which initializes and configures the Dispatcher according to the arguments and runtime environments (e.g., available resources). Then, the Dispatcher spawns N Workers and

binds them to CPU cores. The Dispatcher also iteratively splits a chunk from input data in the Iteration Window, whose size is dynamically adjusted according to the runtime configuration. The chunk of data will be further split into M pieces, which forms M map tasks.

In the Map phase, a Worker selects a map task from the Iteration Window whenever it is idle, and invokes the programmer-provided `map` function, which parses the input data and generates intermediate key/value pairs. The `emit_intermediate` function provided by the runtime will be invoked to insert a key/value pair to Intermediate Buffer, which is organized as an M by R matrix of buckets, where R is the number of reduce tasks.

In the Combine phase, the Workers select the reduce tasks in turn and invoke the programmer-provided `combine` function to process a column in the Intermediate Buffer. The structure of the Iteration Buffer is an I by R matrix of buckets, where I is the total number of iterations.

When all subjobs have finished, the Workers invoke the programmer-provided `reduce` function to do the final reduce operation on the data in each column of the Iteration Buffer. The `reduce` function inserts the final results for a key to the Final Buffer by invoking the `emit` function. Finally, the results from all reduce tasks are merged and orted into a single Output Buffer.

4.3. Runtime Support of Tiled-MapReduce

The runtime of Tiled-MapReduce, called Ostrich, is based on Phoenix, which extends several parts to efficiently support the tiling strategy.

4.3.1. Iteration Window. The memory access pattern of MapReduce inherently has poor temporal and spatial locality. With regards to temporal locality, a MapReduce application usually sequentially touches the whole input data only once in the Map phase to generate the intermediate data. It also randomly touches discrete parts of intermediate data multiple times in the Map and Reduce phases to group key/value pairs and generates the final results. For spatial locality, though the input data is sequentially accessed in the Map phase, a large number of *key-compare* operations results in poor spatial locality. Even worse, each reduce task accesses the key/value pairs generated by different map workers in different time, causing poor spatial locality in the Reduce phase and even severe thrashing when the physical memory has been exhausted.

Tiled-MapReduce provides opportunities to improve data locality for data-parallel applications. The runtime uses the Iteration Window to dynamically adjust the input size for each subjob. As each time only a small subjob is handled, the working set of the subjob can be relatively small. A small working set is beneficial to exploit the cache hierarchy in a multicore platform. Specifically, Ostrich estimates the working set of a subjob by first running a sample subjob to collect its memory requirements. Based on the collected data, Ostrich automatically estimates the size of each subjob according to the cache hierarchy parameters. Currently, Ostrich tries to make the working set of each subjob fit into the last-level cache.

4.3.2. Intermediate Buffer. The use of Tiled-MapReduce provides opportunities to reuse the Intermediate Buffer among subjobs. This could save expensive operations such as concurrent memory allocation and deallocation, as well as building data structures. Instead of constructing an Intermediate Buffer for each subjob, Ostrich uses a global buffer to hold the intermediate data generated in the Map phase. Ostrich will indicate that the buffer is empty at the end of a subjob, but will not free the memory until all subjobs have finished.

Ostrich uses an additional Iteration Buffer to store partial results from all iterations, which is possibly too large for the final Reduce phase. Hence, Ostrich adds a

```
// kv_arr: an array of <key, value> pairs
// pairs: key=word, value=occurrences
serialize ( kv_arr, *stream, *slen ) {
    char *buf = alloc_buffer();
    long buf_len = 0;
    for each <key, value> in kv_arr {
        copy(buf, key);
        copy(buf, value);
        buf_len += key_len + value_len;
    }
    *stream = buf;
    *slen = buf_len;
}
```

```
// stream: serialized <key, value> pairs
// pairs: key=word, value=occurrences
deserialize ( stream, slen, kv_arr ) {
    keyval_t *kvs = alloc_kvs();
    int kv_num = 0;
    for each str in stream {
        kvs[kv_num].key = get_key(str);
        kvs[kv_num].value = get_value(str);
        kv_num ++;
    }
    kv_arr->kvs = kvs;
    kv_arr->num = kv_num;
}
```

```
// input = a document
// offset:len = input for sub-job
acquire ( input, offset, len ) {
    input->flen = len;
    input->fdata = mmap(
                0, len, PROT,
                MAP_PRIVATE,
                input->fd, offset);
}
release ( input ) {
    munmap(input->fdata,
                input->flen);
}
```

Fig. 3. Pseudocode of Word Count application to support fault tolerance (i.e., `serialize` and `deserialize`) and input buffer reuse (i.e., `acquire` and `release`).

new internal phase, namely Compress, to further combine the partial results among subjobs. When the size of partial results exceeds a threshold (e.g., 32), the Compress phase will invoke the `combine` function to combine the partial results among subjobs. The results generated in the Compress phase are also stored in the Iteration Buffer. The previous partial results will be simply discarded and the memory space is reused for the subsequent subjobs. It should be noted that the Compress phase is an internal phase of Ostrich and thus is transparent to programmers.

4.3.3. Thread Pool. The partition of a large job into a number of subjobs increases the number of thread creations and destructions. To avoid such overhead, Ostrich uses a worker thread pool to serve tasks in all phases within a subjob. The worker thread in the pool is also reused among subjob boundaries. The pool is initialized at the beginning of a MapReduce job and serves all tasks in all phases, including the final Reduce and Merge phase.

4.4. Fault Tolerance

The difference between cluster and multicore platform demands new features to the fault-tolerance model. For example, the MapReduce job should survive in global failures (e.g., whole machine crash) and avoid redundant computation during recovery. The fault-tolerance operations should exploit the parallelism of a multicore platform and consume only a few resources. Finally, the low-level details of model should be mostly hidden from user programmers by using general interfaces.

To achieve the these goals, Ostrich extends the fault-tolerance model with the support of backing up the partial results of individual subjobs to save the associated computation during a global failure. Generally, each subjob in Ostrich can back up its partial results to persistent storage (e.g., disk or HDFS [Borthakur 2013]) independently and in parallel. Ostrich uses a self-certified pathname [Mazières et al. 1999] to bookkeep subjobs that are backed up. During recovering from a global failure, Tiled-MapReduce can load the saved results in parallel and only compute those that are not bookkept, which saves a lot of time associated with reexecution. For applications consisting of multiple MapReduce jobs, Ostrich can replace all partial results with the final results to save storage when the current job has been done.

The detailed implementation of fault tolerance and computation reuse in Ostrich is hidden from user programmers, such as I/O parallelism and naming. They are only required to implement two interfaces that are pervasive in MapReduce for clusters (such as Google's MapReduce [Dean and Ghemawat 2008] and Hadoop [Bialecki et al. 2005]), `serialize` and `deserialize`, which convert key/value pairs to and from a character stream. Figure 3 shows the pseudocode of the `serialize` and `deserialize` functions for the Word Count application. In most cases, user programmers can simply use the default implementation provided by the Tiled-MapReduce framework.

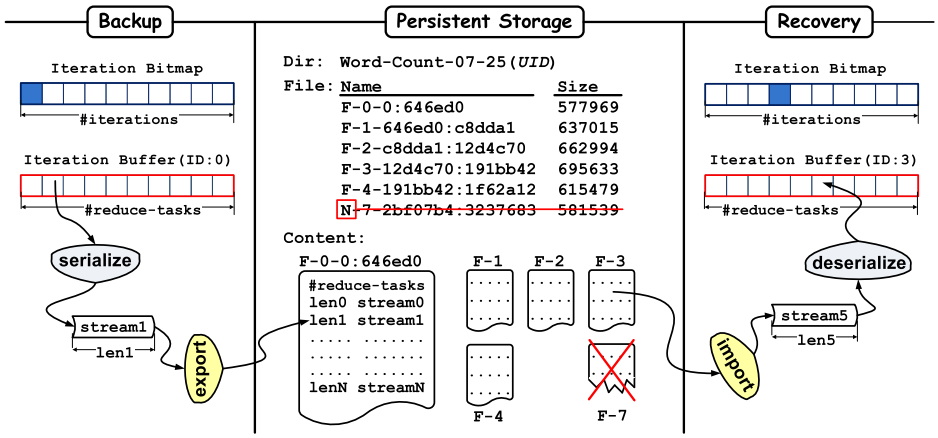

Fig. 4. The key structure in the main memory and persistent storage to support fault tolerance, and an example of the backup and recovery operations.

Figure 4 illustrates an example of the backup and recovery operations in Ostrich. At the beginning, Ostrich uses the Unique Identifier provided by user programmers (e.g., *Word-Count-07-25*) to make a directory on persistent storage, which stores all snapshot files of subjobs. Ostrich uses an Iteration Bitmap to record whether the partial result of the corresponding iteration has been saved to the snapshot file. When an iteration is finished, the combine worker backs up the partial results to a snapshot file on persistent storage. First, the worker uses the file state (N means in progress and F means finished), the iteration number, and the range of input data to create a temporary snapshot file. Then, the worker uses the `serialize` function provided by user programmers to transfer the partial results in the Iteration Buffer to multiple streams, and exports them to a temporary file. Finally, the worker atomically renames the temporary snapshot file to the final snapshot file. When a failure happens, users can recover the job with the same Unique Identifier. Ostrich imports partial results from the final snapshot files in a corresponding directory to the Iteration Buffer through the `deserialize` function provided by user programmers, and sets the Iteration Bitmap to skip these iterations.

5. OPTIMIZING RESOURCE USAGES

The support of iterative processing of small subjobs allows fine-grained management of resources among subjobs. This opens opportunities for new optimizations on resource usage in Tiled-MapReduce. In this section, we describe three optimizations that aim at improving memory efficiency, data locality, and task parallelism.

5.1. Memory Reuse: Input Buffer Reuse

In the general MapReduce programming model, the input data is kept in memory in a MapReduce job's whole lifecycle. The intermediate data is also kept in memory in a whole MapReduce phase. These create significant pressure to the memory systems. Tiled-MapReduce mitigates the memory pressure due to intermediate data by limiting the required memory for the input and intermediate data to subjob level, and reusing the Intermediate Buffer among subjobs. After reusing the Intermediate Buffer, the Input Buffer then consumes the major portions of memory.

The input data in MapReduce is usually required to be kept in memory along with the whole MapReduce process. For example, in the Word Count application, the Intermediate Buffer will keep a pointer to a string in the input data as a key, instead of

copying the string to another buffer. For a relatively large input, the input data will be one of the major consumers of memory and there is poor cache locality in accessing them.

Ostrich is designed to balance the benefit of a large memory footprint and the cost of additional memory copies. For applications that likely have abundant duplicated keys (or values), it would be more worthwhile to copy the keys (or values) instead of keeping a pointer to the input data. Copying the keys (or values) together can also improve the data locality, as the copied keys (or values) can be put together, instead of being randomly scattered across the input data. Hence, Ostrich allows copying keys (or values) at the `combine` phase.

By breaking the dependency between intermediate data and input data, Ostrich supports the reuse of a fix-sized memory buffer to hold the input data for each subjob. The memory buffer is remapped to the corresponding portion of input data when a new subjob starts.

Optional interfaces. `acquire` and `release`. To support dynamically acquiring and releasing input data, Ostrich provides two optional interfaces, namely `acquire` and `release`, which will be invoked at the beginning and end of each subjob. Figure 3 shows the pseudocode of the `acquire` and the `release` functions for the Word Count application. In the example, the `acquire` and `release` functions simply map and unmap a portion of the input file into/from memory.

These two interfaces are not provided from scratch. In fact, there are counterparts of the `acquire` and the `release` interfaces in Google's own implementation and Hadoop. For example, Hadoop requires programmers to implement the `RecordReader` interface to process a piece of input split generated from the `InputSplit` interface. The `acquire` interface resembles the `constructor` function of `RecordReader` interface, which is used to get data from a piece of input. The `release` interface also resembles the `close` function of the `RecordReader` interface, which closes a piece of input. Nevertheless, the two interfaces are optional, as Ostrich has provided multiple default implementations for input with general data structures like the Google's MapReduce and Hadoop, in order to reduce effort programmer.

5.2. Exploiting Locality: NUCA/NUMA-Aware Scheduler

Currently commodity multicore hardware usually organizes caches in a NonUniform Cache Access (NUCA) way. The memory also tends to be organized in a nonuniform way (i.e., NUMA) as the number of cores increases. Thus, the latency of accessing caches or memory on remote chips is much slower than that on local cache [Boyd-Wickizer et al. 2008]. Further, the cost of synchronization among threads on different cores also increases with a growing amount of cores. In a MapReduce runtime, even if the working set of each subjob fits into the local cache, the cross-chip cache accesses still cannot be avoided in the Combine phase. Thus, current MapReduce schedulers (e.g., Phoenix) have problems in scaling well on multicore systems, mainly due to the fact that they use all cores belonging to different chips to serve a single MapReduce job.

According to our measurement on Phoenix, MapReduce scales better with the increase of the input size than on the number of cores. For example, the total execution time (66s) of the Word Count application using four cores to process 1GByte input four times and merging the final results is notably less than the time (80s) of using 16 cores to process 4GByte input once. Hence, we propose a NUCA/NUMA-aware scheduler that runs each iteration on a single chip and allocates memory from local memory pool, which could significantly reduce the remote cache and memory accesses in the Combine phase. The NUCA/NUMA-aware scheduler also increases the utilization

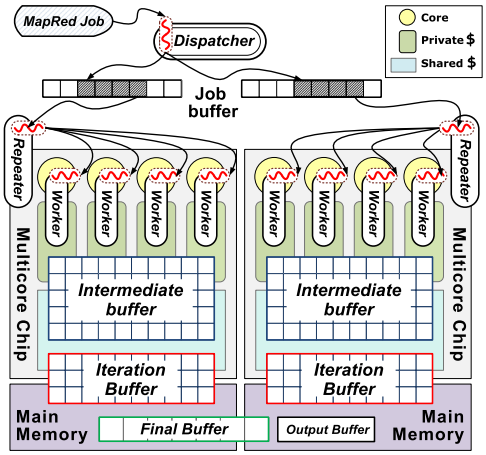

Fig. 5. The logical view of the multicore-oriented scheduling model used by Tiled-MapReduce.

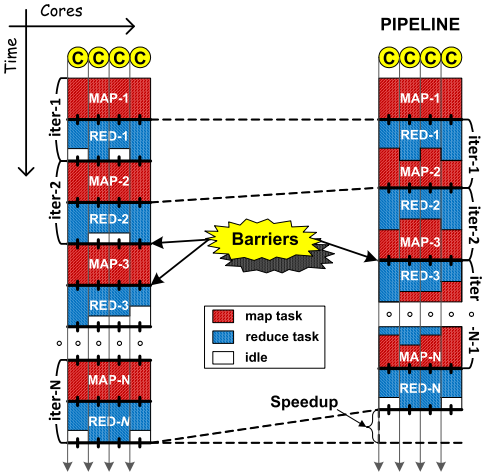

Fig. 6. The task pipeline optimization of Tiled-MapReduce: the left figure shows the original execution flow, while the right figure depicts the execution flow after the task pipeline optimization.

of caches by eliminating the duplication of input data and intermediate data among caches in multiple cores.

Figure 5 provides an overview of the NUCA/NUMA-aware scheduler. The main thread serving as the Dispatcher spawns a Repeater thread for each chip (a scheduling group), which further spawns Worker threads on each core within the chip. Each Repeater has a private Job Buffer to receive subjobs from Dispatcher, a private Intermediate Buffer used by workers to store the output of the Map phase and a private Iteration Buffer to store the partial results generated by workers in the Combine phase.

Using a NUCA/NUMA-aware scheduler might cause some imbalance among Repeaters in different scheduling groups, due to unbalanced workloads assignment to these groups. To balance workload among them, Ostrich implements the work-stealing mechanism proposed by Cilk [Frigo et al. 1998], to allow a Repeater to actively steal jobs from other scheduling groups when it has done all the local jobs.

5.3. Task Parallelism: Software Pipeline

In a general MapReduce programming model, there is a strict barrier between the Map and Reduce phases: the workers in one phase can only be started until all workers in the previous phase have finished. Hence, the execution time of a job is determined by the slowest worker in each phase. The imbalance of tasks can be solved by dynamic scheduling in the Map phase. However, in the Reduce phase, as all values for the same key must be in one reduce task, so it is not always feasible to generate a large number reduce tasks for dynamic scheduling. For example, applications with only a small number of keys can only generate a small number of reduce tasks. Moreover, the workload of each key could be imbalanced as the number of key/value pairs with the same key can vary significantly.

Tiled-MapReduce uses a software pipeline to create parallelism among adjacent subjobs, since there is no data dependency between one subjob's Reduce phase and its successor's Map phase. Figure 6 illustrates the pipeline optimization in Tiled-MapReduce. The y-axis is the time and the x-axis is a list of cores. The left of the figure is the execution flow of the normal Tiled-MapReduce runtime, which has strict

barriers between each phase and can have many idle states due to the imbalance of tasks. The right of figure is the execution flow with pipeline optimization, which overlaps the Reduce phase of the current subjob and the Map phase of its successor.

As the reuse of the Intermediate Buffer among subjobs also creates resource dependency between adjacent subjobs, Ostrich uses a dual buffer to eliminate the dependency. This will increase the memory consumption, but can shorten the total execution time. Users can avoid the use of a dual buffer by disabling the software pipeline optimization.

6. NEW COMPUTING MODELS BASED ON TILED-MAPREDUCE

Compared to the general MapReduce programming model, Tiled-MapReduce not only improves the performance and robustness on multicore platforms, but also enables new computing models. This section describes how Ostrich is adjusted to support the online computing and incremental computing models.

6.1. OOPS: Ostrich Online Prototype System

MapReduce has been used as the processing engine of high-level query languages, such as Sawzall [Pike et al. 2005], Pig [Olston et al. 2008], and Hive [Thusoo et al. 2009]. In many cases, users may prefer to observe the progress of their aggregation queries and control the query execution on-the-fly, which is known as *online aggregation* [Hellerstein et al. 1997] in the database community. However, the query submitted to the MapReduce processing engine will not return any results until the final accurate results have been found, since the general MapReduce programming model is naturally designed to be bulk-processing. To demonstrate the flexibility of Tiled-MapReduce, we build a prototype based on Ostrich to support online aggregation, called OOPS (Ostrich Online Prototype System).

To support online aggregation, OOPS generates the online output according to partial results of completed subjobs, which is quite close to the current progress and is naturally supported in Tiled-MapReduce. By reusing partial results, OOPS minimizes the performance overhead through avoiding repeated aggregations to the same intermediate data. The implementation of OOPS is also quite simple, since we only need to append a Show phase to the end of each subjob. In the Show phase, the Worker thread checks whether to return the online output according to user's configuration. If needed, the Worker thread invokes the `show` function with the online results produced by the `reduce` function from current partial results. Figure 7 illustrates the pseudocode of the `show` function for the Word Count application. The implementation of the `show` function is similar to the execution on the final results, which sorts the results by occurrence and dumps the top 10 words to screen. Hence, there is almost no additional burden to programmers.

OOPS also supports selective online aggregation according to the requirements of performance and latency. Programmers can use the *progress* information to decide whether to execute and output the online results. To harness the multicore resources, programmers may write a parallelized version of the `show` function, or even run a new MapReduce job to generate online results. Therefore, it is possible to concurrently run the `show` function and other subjobs on the same core due to the NUCA/NUMA-aware scheduler. To avoid potential contention on resources, the runtime can allocate the set of cores used by the current subjob to the `show` function.

Online aggregation could also be used to show the trend of results, such as the monthly change in an annual report. In these cases, the runtime should ensure the consistency between the ordering of input data and online output. OOPS provides the support for out-of-order execution of subjobs and invoking the show functions in order, which resembles current out-of-execution/in-order commit of instructions in modern

```
// progress: the number of iterations has done
// cset: the set of cores used by current sub-job
// cur_data: current online results
show ( progress, cset, cur_data ) {
    int num = cur_data->num;
    keyval_t data = cur_data->data;

    if (is_show(progress))
        return;

    qsort(data, num, sizeof(keyval_t),
            val_sort_cmp);

    dump_top(data, 10);
}
```

Fig. 7. Pseudocode of Word Count application to support online aggregation (i.e., Show).

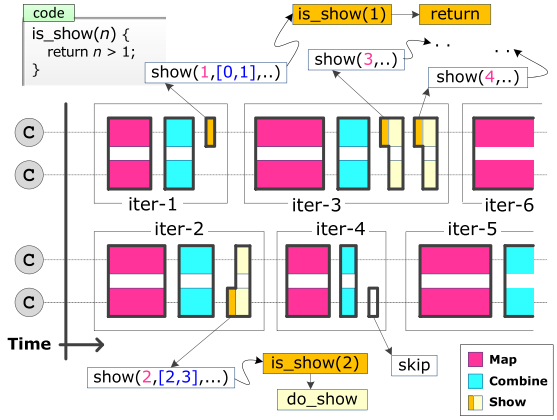

Fig. 8. An example of execution flow on OOPS with in-order online results.

processors. The iteration number assigned by the partition function for each subjob is used to describe the order of subjobs. The runtime uses an Online Bitmap to record the finished jobs, and invokes the `show` function when all previous subjobs have done.

Figure 8 illustrates an example of the execution flow on OOPS with 4 cores. OOPS executes the `show` function at the end of each iteration, with the progress and cores information. The `show` function first uses the `is_show` subroutine to selectively output the online results from the second iteration, and then executes the `do_show` subroutine on the cores used by the current iteration. To ensure the sequential orders of the online output, the runtime delays the Show phase of the fourth iteration until the third iteration has been completed.

Multitier online aggregation is naturally supported by OOPS, and users only need to provide different `show` functions for different MapReduce jobs. Figure 9 illustrates the execution flow of two-tier online aggregation. OOPS invokes the `show` function `S1` and `S2` for each iteration in the first and second tier of the MapReduce job, respectively. The implementation of the `S1` function is similar to that of the second-tier MapReduce job.

OOPS trivially reserves the fault-tolerance model of Tiled-MapReduce. The snapshot files of partial results could be used to generate the online results asynchronously, even on a remote machine.

The OOPS implementation has three minor changes, including aggregation computing, scheduling, and configuration, which are about 55, 90, and 10 lines of C code, respectively.

6.2. OstInc: Ostrich Incremental Computing System

The workload processed by MapReduce applications is often mutable. For example, new records are periodically appended to the system logs, and the Web pages crawled from the network are continuously changing. This incremental nature of workloads suggests that the programming model should support performing large-scale computation incrementally. The general MapReduce applications have to be executed repeatedly with small changes in their input. To demonstrate the flexibility of Tiled-MapReduce, we build a prototype based on Ostrich to support incremental computing, called OstInc (OSTrich INCremental computing system).

OstInc reuses the backup and the recovery functions that enable fine-grained fault tolerance to support two types of incremental computing, namely append-only and

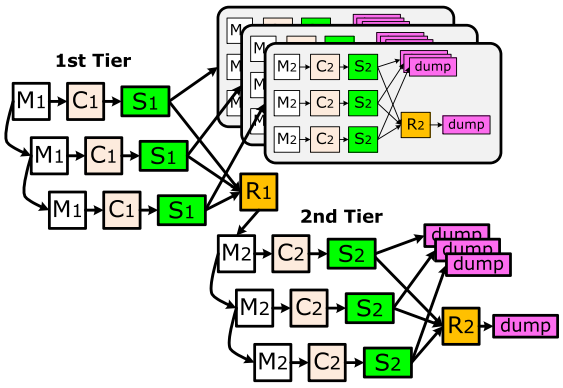

Fig. 9. The execution flow of two-tier online aggregation on OOPS.

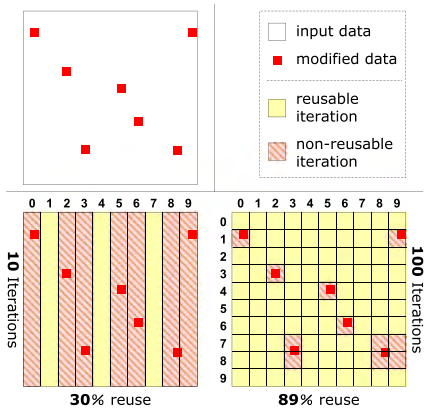

Fig. 10. The impact of iteration size on the proportion of reuse in incremental computing.

partial. For append-only incremental computing tasks such as processing incremental log files and reporting periodical statistical results, the new input data is just appended to the end of previous input data and previous saved results can be fully reused. For partial incremental computing such as modified Web pages and rendered pictures, only partial previous input data is modified and previous results should be reused as much as possible.

Currently, OstInc needs the Unique Identifier of the previous job and a compatible partition. It also assumes the behavior of two executions is compatible, meaning that same input would result in same output.

Figure 11 illustrates examples of how OstInc supports append-only and partial incremental computing. For append-only incremental computing, OstInc reuses the backup function to store the input length and final results to persistent storage, and reloads them via the Unique Identifier. OstInc directly reuses the previous final results as the partial results of the first iteration of the current job, and skips the reexecution of the subjob to process such input.

For partial incremental computing, OstInc reuses previous results at iteration level. Hence, programmers need to use a compatible partition function for the current job. OstInc hashes the input of each iteration using the MD5 hash as the Iteration Descriptor, and backs it up with the partial results to persistent storage. OstInc then loads all Iteration Descriptors to main memory at the beginning of the current job, according to the Unique Identifier. For each iteration, OstInc matches the MD5 hash of input with them to decide whether the iteration can be skipped. The results of unmatched iteration will be backed up for future reuse. The proportion of input reuse depends on the granularity of iterations. In Figure 10, for the same input with seven random changes, the job with 10 iterations has only 3 iterations (30%) to be reused. while the job with 100 iterations has 89 reused iterations (89%).

Due to the flexibility of Tiled-MapReduce, Ostrich can trivially support incremental computing. The OstInc implementation has three minor changes, including scheduling, task backup, and matching, with about 35, 40, and 25 lines of C code, respectively.

7. EVALUATION

This section presents the experimental results for Ostrich and its applications by using an AMD multicore machine. Since Ostrich follows the programming interfaces, basic algorithms, and data structure from the initial version of Phoenix [Ranger et al.

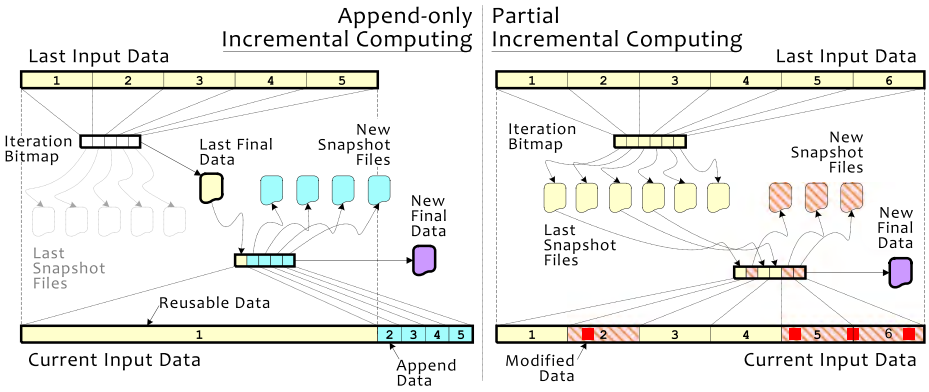

Fig. 11. Examples of OstInc to support append-only and partial incremental computing.

2007], we mainly compare the performance of Ostrich with Phoenix [Yoo et al. 2009], to demonstrate the performance benefit of Ostrich due to the tiling strategy and the associated optimizations. We also compare Ostrich with Phoenix++ [Talbot et al. 2011], a recent refinement of Phoenix with several sophisticated optimizations on data structures and the combiner abstraction after the our initial implementation of Ostrich [Chen et al. 2010]. Finally, we evaluate the performance of OOPS and OstInc described in Section 6.

All experiments were conducted on a 48-core machine with eight 2.4 GHz 6-core AMD Opteron chips. Each core has its own private 64KByte instruction and data caches, and a 512KByte L2 cache. The cores on each chip share a 5MByte L3 cache. The size of the physical memory is 128GByte. We use Debian Linux with kernel version 3.2.10, which is installed on a 40GByte SCSI hard disk with an ext3 file system. All input data are stored in a separated partition in a 116GByte SCSI disk. The memory allocator for both Ostrich and Phoenix is the jemalloc-3.0.0 [Evans 2013], which was shown to have good scalability on multicore platforms. We run the experiment five times and report the average result.

7.1. Tested Applications

As the performance benefit of Tiled-MapReduce is sensitive to characteristics of key/value pairs of MapReduce applications, we chose four different MapReduce applications representing four major types of MapReduce applications regarding the attributes of key/value pairs. Word Count has a large number of keys and each key has many duplicated key/value pairs (likeness: Term-vector, Sequence-Count [Ahmad et al. 2011a]). Distributed Sort also has a large number of keys, but there is no duplicate among key/value pairs (likeness: Self-Join, Tera-Sort [Ahmad et al. 2011a]). Log Statistics has a large number of key/value pairs, but with only a few numbers of keys (likeness: Histogram-Ratings/Movies, Classification [Ahmad et al. 2011a]). Inverted Index has only a few key/value pairs, and key/value pairs cannot be aggregated (likeness: Grep, Ranked-Inverted-Index [Ahmad et al. 2011a]). In a relatively small number of cores (e.g., 12 cores), Tiled-MapReduce is expected to have significant performance speedup on Word Count, and still show notable performance improvement on Distributed Sort. It's mainly due to the improvement of cache locality through tiling workloads and using a NUCA/NUMA-aware scheduler. For Log Statistics and Inverted Index, Tiled-MapReduce is expected to have relatively less performance improvement because there are a very few numbers of keys or key/value pairs. It still benefits from the elimination of CPU idle time via the software pipeline. In a relatively large number

Interfaces	*WC*	*DS*	*LS*	*II*
acquire	11	Default	Default	11
release	3	Default	Default	3
serialize	Default	Default	Default	Default
deserialize	Default	Default	Default	Default

Fig. 12. The required code modification for Tiled-MapReduce, in addition to the code in Phoenix. "Default" indicates that programmers simply use the default implementation provided by Ostrich.

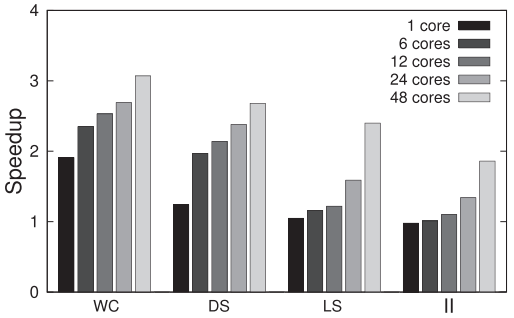

Fig. 13. Overall speed over Phoenix on four benchmarks for 1 to 48 cores with 1GByte input.

of cores (e.g., 48 cores), Tiled-MapReduce is expected to have significant performance speedup on four benchmarks because of much better scalability than Phoenix.

The following are brief descriptions of four benchmarks. If not mentioned, the `combine` function is the same as the `reduce` function.

Word Count (WC). It counts the number of occurrences of each word in a document. The `key` is the word and the `value` is the number of occurrences. The `map` function emits a $\langle word, 1 \rangle$ pair for each word in the document and the `reduce` function sums the values for each word and emits a $\langle word, total\ count \rangle$ pair.

Distributed Sort (DS). It models the TeraSort benchmark [Gray 2013] and uses the internal sorting logic of a MapReduce runtime to sort a set of records. The `key` is the key of the record and the `value` is the record itself. The `map` function emits a $\langle key, record \rangle$ pair for each record and the `reduce` function leaves all pairs unchanged. The sorting of records happens in both the `reduce` and the `merge` functions, which perform local and global sorting according to keys, respectively.

Log Statistics (LS). It calculates the cumulative online time for each user from a log file. The `key` is the user ID and the `value` is the time of login or logout. The `map` function parses the log file and emits $\langle user\ ID, \pm time \rangle$ pairs. The `reduce` function adds up the time for each user and emits a $\langle user\ ID, total\ time \rangle$ pair.

Inverted Index (II). It generates a position list for the word a user specified in a document. The `key` is the word and the `value` is its position. The `map` function scans the document and emits $\langle word, position \rangle$ pairs. The `combine` function is an identity function that emits all pairs unchanged. The `reduce` function sorts all positions of the word and emits a $\langle word, list(position) \rangle$ pair.

Figure 12 lists the code modifications to port a MapReduce application in Phoenix to Ostrich. To support input data reuse, programmers need to provide 11 and 3 lines of code for the `acquire` and `release` functions for WC, which maps/unmaps a portion of the input file into/from memory accordingly. Such code can be written by simply reusing the code that processes input data in Phoenix. For II, the `acquire` and `release` functions are identical to WC, as the input type is the same with WC. For all the four applications, programmers can simply use the default `serialize` and `deserialize` functions provided by the runtime to support fault tolerance.

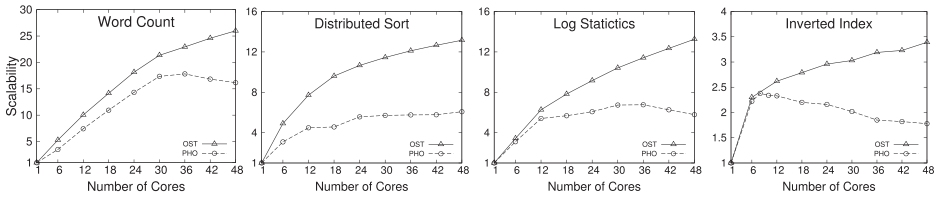

Fig. 14. A comparison of the scalability of Ostrich with that of Phoenix.

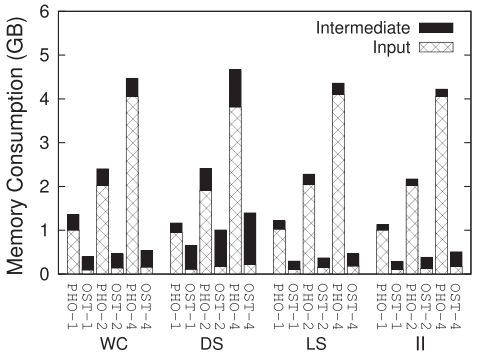

Fig. 15. A comparison of the peak memory consumption for Ostrich to Phoenix with 1-, 2-, and 4GByte input. OST-*N* and PHO-*N* represent the data for Ostrich and Phoenix with *N*GByte input.

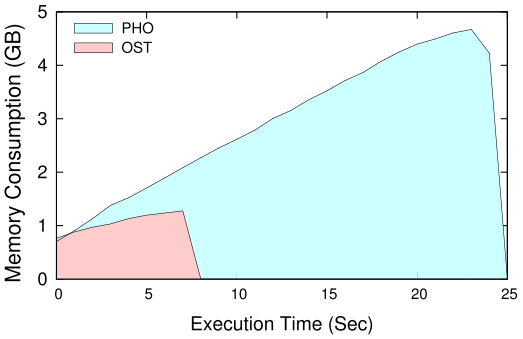

Fig. 16. A comparison of the memory footprint on Ostrich versus Phoenix for WC benchmark with 1GByte input.

7.2. Overall Performance

Figure 13 shows the speedup of Ostrich compared to Phoenix for 1 to 48 cores. The input size for each benchmark is 1GByte. As shown in the figure, the largest speedup comes from WC, in which Ostrich outperforms Phoenix by 1.91x, 2.53x, and 3.07x for 1, 12, and 48 cores accordingly, while for DS, the speedup is 1.24x, 2.14x, and 2.38x accordingly. The speedup is relatively small for LS (1.22x) and II (1.10x) with less than 12 cores. This is because they have relatively less numbers of intermediate data and keys, which limits the optimization space compared to the former two. But Ostrich still distinctly overcomes Phoenix by 2.40x and 1.86x on LS and II with 48 cores.

Figure 14 compares the scalability of Ostrich with that of Phoenix. As shown in the figure, Ostrich has much better scalability than Phoenix for the four tested applications. With an increasing number of cores, Ostrich enjoys much larger speedup compared to the performance on a single core. This confirms the effectiveness of Tiled-MapReduce and the associated optimizations. The scalability of Phoenix on all benchmarks stagnates or even drops when more cores are involved, which is mainly due to the poor scalability of the Reduce and Merge phases and the increased proportion of the time spent on these phases.

In the following sections, we will categorize the source of the speedups and improved scalability.

7.3. The Benefit of Memory Reuse

7.3.1. Peak Memory Consumption. We first investigate the benefit of the memory reuse optimization. In Figure 15, we compare the peak memory consumption of the four benchmarks on Ostrich versus Phoenix with the input size of 1-, 2-, and 4GByte. The main memory consumption in runtime is from input data and intermediate data. Due to the input buffer reuse optimization, Ostrich significantly reduces the memory

consumption for the input data. For the intermediate data, as Phoenix has already applied the combine optimization to reduce the memory space for intermediate data, Ostrich only has a small improvement over intermediate data for WC. For DS and II, as the key/value pairs cannot be locally combined, Ostrich shows no saving in intermediate data but additionally adds a small space overhead due to the copied keys and the spaces for the Iteration Buffer. For LS, as the intermediate data is relatively small, Ostrich shows no space saving for intermediate data. Anyway, copying keys and grouping data into a small centralized Intermediate Buffer does increase the cache locality.

7.3.2. Memory Footprint. Ostrich also reduces the memory footprint of MapReduce applications in their whole lifecycle, through tiling workloads and reusing buffers. Figure 16 shows both the size and the time of memory consumption for WC on Ostrich is significantly better than that on Phoenix. The increment of memory consumption on Ostrich is less and more steady, since the Input Buffer and Intermediate Buffer are allocated in the first iteration and reused among the rest of the iterations. On the contrary, the memory consumption on Phoenix increases with the processing of input data, and the stale data occupies the memory and is not released until the entire job is finished.

7.4. The Benefits of Exploiting Cache Locality

7.4.1. Improvement from NUCA/NUMA-Aware Scheduler. We compared the performance of Ostrich in two configurations. The configuration without the NUCA/NUMA-aware scheduler partitions all threads into a single group and makes them sequentially dispose of subjobs together, which is also used by Phoenix. By contrast, the NUCA/NUMA-aware scheduler partitions cores on the same chip into the same group and dispatches subjobs in parallel.

Figure 17 shows the effect of the NUCA/NUMA-aware scheduler on four benchmarks with 6, 12, 24, and 48 cores. The improvement of the NUCA/NUMA-aware scheduler is from the elimination of cross-chip cache accesses in the Combine phase. WC, DS, and LS benchmarks spend relatively more time in the Combine phase to reduce the intermediate data, resulting in more performance improvement. Hence, the first three applications (i.e., WC, DS, and LS) enjoy around 70% performance speedup (79%, 58%, and 69% accordingly) for 48 cores. Inverted Index has relatively smaller speedup (24%) because there are relatively few operations in the Combine phase, so the benefit from the scheduler is limited. As Ostrich treats all cores within a chip as a group, the behavior of the new scheduler is the same as the original scheduler when the number of cores is less than or equal to 6 (the number of cores in a chip). Hence, there is little performance improvement.

For all the four applications, performance improvement increases with number of cores. Thus, it is reasonable to foresee that Ostrich could have even more improvement for future machines with abundant cores, especially for the tiled architecture [Waingold et al. 1997].

7.4.2. Relevance of Iteration Size. Tiled-MapReduce provides good opportunities to exploit the memory hierarchy by limiting the footprint of a subjob within a certain range. Figure 18 shows the execution time of the Word Count application with different number of subjobs to process each 1GByte input data using 48 cores. The results show that the WC benchmark has the best performance when each iteration size is close to 6.3MByte (160 subjobs per GByte), for 1-, 2-, and 4GByte input data. This is because the size of the overall last-level cache (i.e., L3) in a chip is 6MByte. The remaining space of the cache is used to store other data such as the Intermediate Buffer. Thus, Tiled-MapReduce tends to enjoy the best performance when the footprint of a subjob

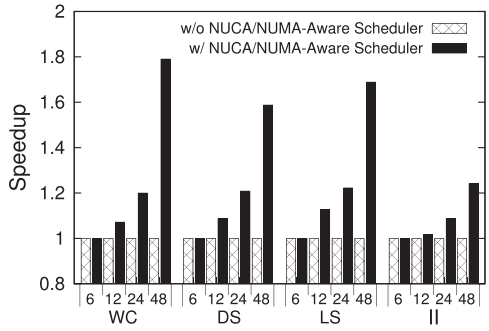

Fig. 17. The effect of NUCA/NUMA-aware scheduler on four benchmarks for 6 to 48 cores with 1GByte input.

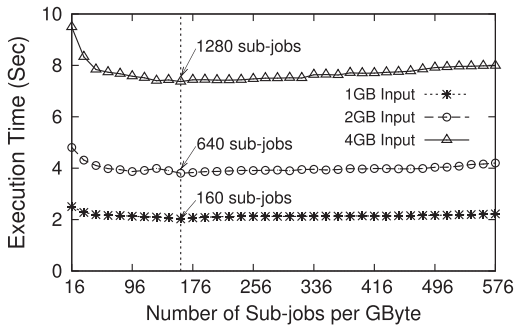

Fig. 18. The impact of iteration size on execution time using 48 cores with 1-, 2-, and 4GByte input.

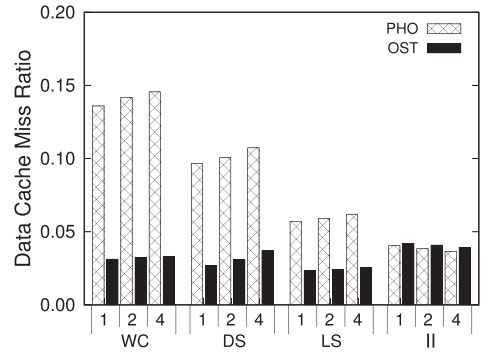

Fig. 19. A comparison of the data cache miss ratio on Ostrich to Phoenix using 48 cores for four benchmarks with 1-, 2-, and 4GByte input.

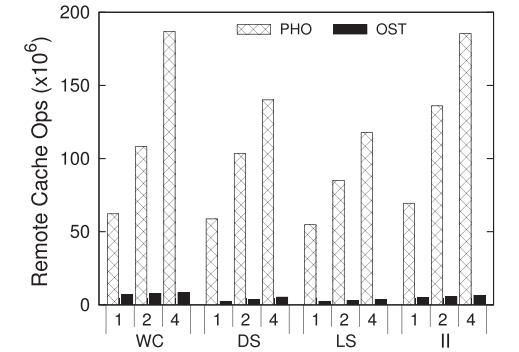

Fig. 20. A comparison of the remote cache access times on Ostrich to Phoenix using 48 cores for four benchmarks with 1-, 2-, and 4GByte input.

just fits in the last-level cache, as creating more subjobs would suffer more from the associated overhead caused by merging these subjobs. One can use a small sample input to estimate the size of each subjob to get an optimal job partition.

7.4.3. Improved Cache Locality. Figure 19 and Figure 20 present the data cache miss ratio and remote cache operations of four benchmarks on Ostrich versus Phoenix using 48 cores with 1-, 2-, and 4GByte input. All data is collected using OProfile [Levon 2004] with the patch from AMD CodeAnalyst [AMD 2013]. The data cache miss ratio indicates the total number of data cache misses divided by the total number of load operations. The miss ratio of WC, DS, and LS benchmarks running on Ostrich is from 2.4x to 4.4x fewer compared to that on Phoenix. As the working set of each subjob matches the total L2 and L3 cache size of a chip (i.e., 3- and 5MByte), thus it ensures that the data fetched in the Map phase may be reused in the Combine phase within the private cache of a chip and is unlikely be accessed after being flushed to memory. The cache miss ratio of the II benchmark for Ostrich is close to that of Phoenix, as there are few optimizing spaces in the II due to the fact that there are very few intermediate data and the Combine phase cannot reduce the memory consumption. The remote cache operations indicate the total number of data accesses from the remote cache. Compared to Phoenix, Ostrich reduces more than 90% remote cache accesses for all benchmarks due to the NUCA/NUMA-aware scheduler, which avoids accessing intermediate data from cache of remote chips.

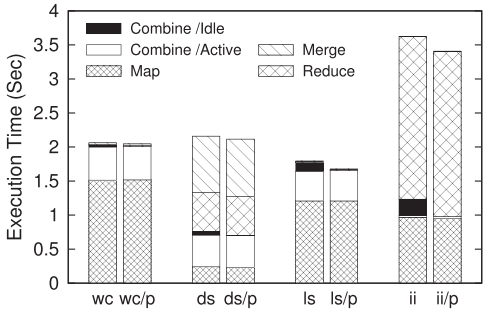

Fig. 21. The benefit of the pipeline with a breakdown of time in all phases using 48 cores with 1GByte input. The label "/p" means with pipeline.

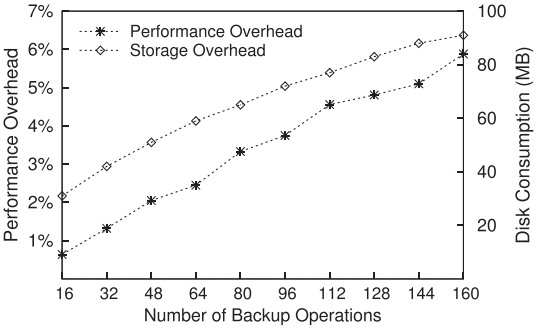

Fig. 22. The performance and storage overhead of backup operations for WC benchmark using 48 cores with 1GByte input, while performing backup operations after each iteration.

7.5. The Benefit of Pipeline

As the major benefit of pipeline optimization is from the elimination of wait time in the Combine phase, we further split the time spent on the Combine phase into active time and idle time. The active time is the average of total execution time of all workers during the Combine phase, and the idle time is the difference between the active time and the time spent on the Combine phase.

Figure 21 shows the time breakdown of benchmarks with and without pipeline optimization using 48 cores with 1GByte input. The idle time dominates the Combine phase of II (89%), because it only indexes one word. Thus only one worker is active in the Combine phase, which results in a notable imbalance. For LS, the number of login records of each User ID is remarkably different, so the workload of each worker in the Combine phase is unbalanced. Hence, the pipeline brings a relatively large improvement, reducing 22% of time spent on the Combine phase. For DS, the default partition function cannot thoroughly balance the workload, thus the pipeline still has some benefit (close to 11%). For WC, which has a large number of keys and duplicated pairs for dynamic load balancing, the improvement from the pipeline is limited (5%).

7.6. Fault Tolerance

7.6.1. Performance Overhead. Figure 22 shows the relationship between performance overhead and the number of backup operations, using the Word Count benchmark on 48 cores with 1GByte input. The performance overhead of the fault-tolerance mechanism depends on the frequency of backup operations. In the tested configuration using 1GByte input, there are 160 subjobs in the optimal configuration. Performing backup operations on every 10 subjobs incurs less than 1% performance overhead, while doing backup operations after each subjob incurs about 6% overhead. Thus, there is a trade-off between the cost of backup and the benefit from the efficiency of recovery from failure.

7.6.2. Storage Overhead. To support fault tolerance, Ostrich needs additional persistent storage to back up the partial results snapshot for each iteration. The storage overhead depends on the total size of snapshot, which is indirectly affected by the number of iterations. Figure 22 shows the relation between storage overhead and the number of iterations for WC using 48 cores with 1GByte input. The results show that the fault-tolerance mechanism requires 91MByte storage in the optimal configuration (i.e., 160 iterations). The storage overhead decreases to 31MByte while configured to 16 iterations.

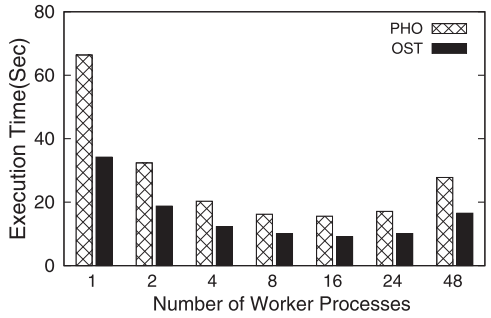

Fig. 23. A comparison on the throughput of MapReduce servers based on Ostrich and Phoenix with the number of worker processes from 1 to 48. The workload has 100 mixed MapReduce jobs, which are randomly generated from our four benchmarks with about 100MByte input.

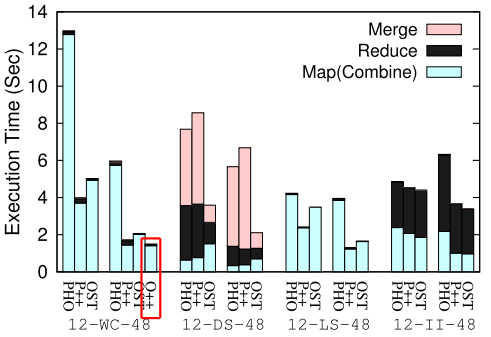

Fig. 24. A comparison among Phoenix, Phoenix++, and Ostrich with a breakdown of time in all phases using 12 and 48 cores with 1GByte input.

7.7. Ostrich as MapReduce Server

To demonstrate the effectiveness of using Ostrich when processing multiple MapReduce jobs in batch mode, we built a MapReduce server using both Ostrich and Phoenix. The server forks multiple worker processes to process MapReduce requests in parallel. Each worker process uses a partial number of cores to serve requests. Figure 23 shows the performance comparison of MapReduce servers based on Ostrich and Phoenix, with the number of worker processes ranging from 1 to 48. For the server with 48 worker processes, each process exclusively uses one core and continuously serves the MapReduce requests. As shown in the figure, the MapReduce server based on Ostrich outperforms the one based on Phoenix under each configuration from 1.60x to 1.94x, due to better cache locality and performance scalability in Ostrich. The results also show that running multiple MapReduce jobs in parallel is a much better way to maximize the overall throughput, since the scalability of MapReduce is not linear in many cases, especially when the number of cores exceeds 12 in our evaluation. However, contention on shared caches might result in performance degradation, which is shown by our results when the number of worker processes exceeds 16.

7.8. A Comparison with Phoenix++

Recently, the Phoenix runtime was completely revised by Talbot et al. in Phoenix++ [Talbot et al. 2011] after our initial implementation of Ostrich [Chen et al. 2010]. It provides a flexible intermediate key-value storage abstraction and a more effective combiner implementation to minimize the overhead and memory usage in the Map phase. This, however, may be at the cost of additional programmers' effort in providing nonstandard storage abstraction and containers. Figure 24 shows the time breakdown of benchmarks on Phoenix, Phoenix++, and Ostrich using 12 and 48 cores with 1GByte input. Compared to Phoenix, both Phoenix++ and Ostrich achieve significant performance improvement, but a similar effect is derived from different optimization techniques. For WC and LS, Phoenix++ outperforms Ostrich by 1.19x and 1.28x using 48 cores. This is because a large number of duplicated key-value pairs provide immense optimization space to the effective Intermediate Buffer and combiner implementation. For example, instead of the standard key/value pair abstraction, Phoenix++ uses a customized combiner to do in-place aggregation of key/value pairs, which thus eliminates the Reduce phase. For DS and II, the improvement of Phoenix++ is limited because there are no duplicate pairs for in-place aggregation. Ostrich also cannot improve the

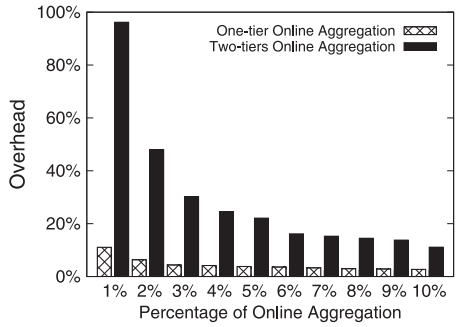

Fig. 25. The performance overhead of OOPS compared to Ostrich for WC benchmark using 48 cores with different frequency of online aggregation.

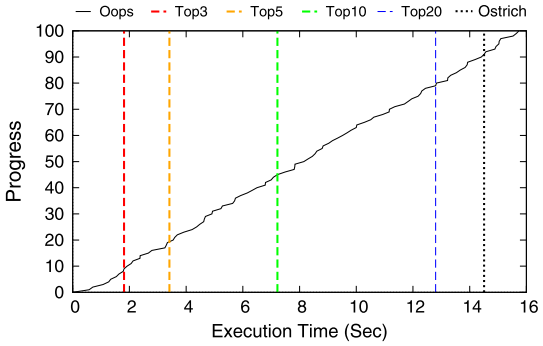

Fig. 26. Top query over 6.4GByte Wikipedia articles. The vertical lines note the times we observed the top query answers produced by online aggregation and the execution time on Ostrich.

performance of the Map phase, and even degrades due to additional memory copy of keys. However, the tiling strategy and NUCA/NUMA-aware scheduler significantly improve the locality of intermediate data. Hence, Ostrich outperforms Phoenix++ by 1.48x and 6.53x in Reduce and Merge phases for DS. For II, the improvement is not obvious because it only has one key.

Phoenix++ focuses on using more effective data structure and abstraction to optimize the aggregation operations, and outperforms Ostrich in WC-like applications. In contrast, Ostrich focuses on improving the programming model by the tiling strategy, as well as optimizing cache locality and CPU usages, and thus outperforms in DS-like applications. Actually, we believe the techniques adopted by Ostrich are orthogonal to Phoenix++, and these two can combined together for further improvement. To validate this, we incorporate the in-place *associative_combiner* abstraction in Phoenix++ for the WC application. As shown in Figure 24, Ostrich++ outperforms Ostrich and Phoenix++ by 1.37x and 1.15x for WC.

7.9. Costs of Supporting New Computing Models

7.9.1. OOPS. Figure 25 shows the performance overhead of one- and two-tier online aggregation on OOPS, using the Word Count benchmark on 48 cores with 1GByte input. The result shows that one-tier online aggregation incurs 11.1% and 2.7% overhead while calling the `show` function for each 1% input and 10% input. For two-tier online aggregation, the performance overhead increases to 96.2% and 11.0%, respectively. This is because the `show` function directly uses `qsort` to sort the result in one-tier online aggregation, and uses an additional MapReduce job to sort the result in two-tier. Thus, the performance overhead of OOPS mainly comes from the execution of the `show` function.

To show the time/accuracy trade-off in OOPS, we write a WC-like benchmark to query the top 100 frequent words in 6.4GByte Wikipedia articles [Wikimedia 2013]. The implementation of the `show` function is less than 5 lines of code using one-tier online aggregation. Figure 26 presents the progress and accuracy of top query on OOPS with 1% online aggregation. We note the time at which the top-K words in online output are the top-K words in the final results. Although the final results appearing in OOPS is 10.8% later than those in Ostrich, we did observe the top-3, -5, -10, and -20 values at the progress of 9%, 18%, 45%, and 79%, respectively.

7.9.2. OstInc. To evaluate the effectiveness of OstInc for append-only incremental computing, we run the LS benchmark on OstInc and Ostrich with an incremental log

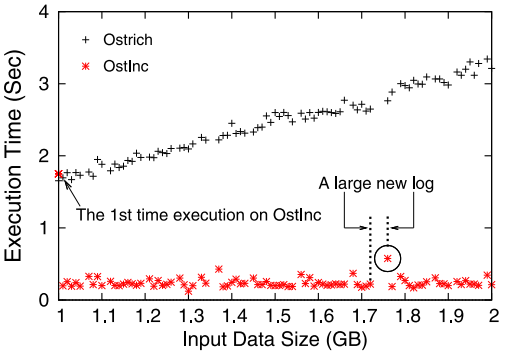

Fig. 27. A performance comparison between OstInc and Ostrich for LS benchmark with an incremental log file.

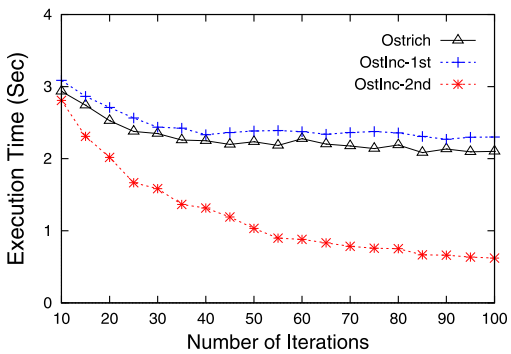

Fig. 28. A performance comparison between OstInc and Ostrich for WC benchmark with a random modified document.

file from 1- to 2GByte. The new log records are always appended to the end of the file. Figure 27 shows that OstInc significantly reduces the execution time except for the first time, because it can fully reuse the previous results for append-only incremental computing. Thus, the execution time of OstInc depends on the size of fresh data.

For partial incremental computing, the effectiveness of reuse depends on the number of iterations. We thus evaluate the WC benchmark with a 1GByte document, which is randomly modified in 20 places. As shown in Figure 28, the first time execution of the WC benchmark on OstInc incurs less than 11% performance overhead. But the WC benchmark for modified input on OstInc outperforms that on Ostrich through reusing the results from unmodified iterations. With an increasing number of iterations, OstInc enjoys more improvement compared to Ostrich. In the configuration with 100 iterations, OstInc reduces 70.4% execution time by skipping 82 iterations.

8. RELATED WORK

Our work is related to the research in programming models and runtime for data-parallel applications, MapReduce on multicore platforms, and nested data parallelism. We briefly discuss the most related work in turn.

8.1. Programming Model and Runtime Related to MapReduce

MapReduce [Dean and Ghemawat 2008] is a popular data-parallel programming model developed in Google. An open-source implementation, namely Hadoop [Bialecki et al. 2005], is provided by Apache, which is implemented in Java and uses HDFS as the underlying file system. The current implementation of Hadoop focuses on cluster environments rather than on a single machine. It does not exploit the data locality at the granularity of a single machine, neither does it provide a multicore-oriented scheduler.

Dryad [Isard et al. 2007] is a programming model for data-parallel applications from Microsoft. Dryad abstracts tasks as nodes in the resource graph and relies on the runtime to map nodes to graph. The Bulk Synchronous Parallel (BSP) model [Valiant 1990] also provides high-level abstractions to give programmers an option to avoid the burden of low-level memory management, communication, and synchronization. Compared to MapReduce, the applications on Dryad/BSP may be specified by an arbitrary DAG/superstep rather than a sequence of map and reduce operations. Hence, they sacrifice some simplicity of interfaces for possible efficiency and flexibility.

Piccolo [Power and Li 2010] is a programming model for data-centric applications that require accessing or changing shared state iteratively. Compared to data-flow

models (e.g., MapReduce and Dryad), Piccolo provides applications with an explicit communication mechanism via distributed key-value tables.

OpenCL [Khronos Group 2009] is a programming model for heterogeneous parallel computing systems, which provides a common programming layer to users. SnuCL [Kim et al. 2011] extends the OpenCL semantics to support heterogeneous CPU/GPU cluster environments. It allows the OpenCL applications to utilize the heterogeneous compute nodes as devices of the host node.

The popularity of MapReduce is also embodied in running MapReduce on other heterogeneous environments, such as on GPUs [He et al. 2008], Cell [de Kruijf and Sankaralingam 2007], and FPGA [Shan et al. 2010]. To ease the programming of MapReduce applications on heterogeneous platforms such as GPUs and CPUs, Hong et al. [2010] recently developed a system called MapCG, which provides sourcecode-level compatibility between these two platforms. Coupled with a lightweight memory allocator and hash table on CPUs, they showed that MapCG has considerable performance advantage over Phoenix and Mars. Tarazu [Ahmad et al. 2012] is a suite of optimizations to improve the performance of MapReduce applications on heterogeneous clusters, which provides three new scheduling mechanisms to reduce the network traffic cost and load imbalance.

8.2. Extension and Optimization to MapReduce

The MapReduce programming model has been applied to a wide range of domains, including data mining [Ekanayake et al. 2008], machine learning [Chu et al. 2006], visualization [Stuart et al. 2010], and bioinformatics [Matsunaga et al. 2008]. Several SQL-style declarative languages have appeared on top of MapReduce to hide the domain-specifics difference, such as Sawzall [Pike et al. 2005], Pig [Olston et al. 2008], Hive [Thusoo et al. 2009], and YSmart [Lee et al. 2011]. The counterpart on top of Dryad is DryadLINQ [Yu et al. 2008], which integrates Dryad with high-level language (i.e., .NET) and allows users to program using SQL-like programming language.

Since the MapReduce programming model is naturally designed only to bulk-processing tasks, there is some work aiming at extending the programming model for various purposes. For example, the database community extends the MapReduce programming model by adding an additional phase, namely Merge, to join two tables [Yang et al. 2007]. Online MapReduce [Condie et al. 2010] extends the MapReduce runtime using a pipelining scheme to support two features in the database domain, the online aggregation and continuous query processing.

To improve the performance of iterative MapReduce jobs, Twister [Ekanayake et al. 2010] uses distributed memory caches to avoid repeated input loading from disk. Spark [Zaharia et al. 2010] enhances the cross-job distributed cache with fault-tolerance support.

For incremental/continuous computing workloads, DryadInc [Popa et al. 2009] reuses computation results from previous jobs based on system job graphs of Dryad. Incoop [Bhatotia et al. 2011] supports executing MapReduce jobs in an incremental manner by reusing intermediate data from previous runs, and uses a memorization-aware scheduler to enhance the locality of memorization results. The in situ MapReduce [Logothetis et al. 2011] supports continuous log processing through sliding-window-based computation and provides two-dimensional incomplete results output.

Tiled-MapReduce naturally supports the preceding purposes by decomposing a large job into the subjob level, with a special focus on multicore platforms instead of clusters.

The MapReduce system, especially the open-source implementation (i.e., Hadoop) has been optimized from many aspects, including scheduling [Zaharia et al. 2008], data locality [Ananthanarayanan et al. 2011], auto-tuning [Babu 2010], and fault tolerance [Yang et al. 2010]. MaRCO [Ahmad et al. 2011b] also exploits the commutativity

and associativity of the reduce function to support overlapping the shuffle and reduce operations. However, none of these efforts focuses on the multicore platform.

Azwraith [Xiao et al. 2011] has tried to combine a previous version of Ostrich [Chen et al. 2010] to Hadoop using a hierarchical approach, by which a Map/Reduce task assigned to each machine is further decomposed to the Ostrich runtime to a series of Map/Reduce tasks in Ostrich. In this way, Azwraith gains 1.4x to 3.5x performance speedup over Hadoop due to the exploiting of parallelism and locality in Ostrich.

8.3. MapReduce on Multicore Platforms

Ranger et al. [2007] provide a MapReduce implementation on multicore platform. Their implementation, namely Phoenix, successfully demonstrates that applications written using MapReduce are comparable in performance to their pthread counterparts. Compared to MapReduce, Tiled-MapReduce partitions a big MapReduce job into a number of independent subjobs, which improves resource efficiency and data locality, thus significantly improves the performance. Yoo et al. [2009] heavily optimize Phoenix from three layers: algorithm, implementation, and OS interaction, and Talbot et al. [2011] further improve Phoenix runtime by flexible containers and more effective combiner implementation. Mao et al. [2010] also builds a MapReduce runtime for multicore that applies many algorithm- and data-structure-level optimizations over Phoenix. In contrast, Ostrich optimizes MapReduce mainly at the programming-model level by limiting the data to be processed in each MapReduce job, which significantly reduces the footprint and enables other locality-aware optimizations. Hence, Ostrich is orthogonal to these optimizations and can further improve the performance of these systems.

Merge [Linderman et al. 2008] is a programming model that targets heterogeneous multicore platforms. It uses library-based model that is similar to MapReduce to hide the underlying machine heterogeneity. It also allows automatic mapping of the computation tasks into available resources. Lee et al. [2010] implement an OpenCL framework which aims at heterogeneous multicore architectures with local memory. It uses software-managed caches and coherence protocols to overcome the limitation from internal local memory.

A multicore operating system, named Corey [Boyd-Wickizer et al. 2008], proposes three new abstractions (*address ranges*, *shares*, and *kernel cores*) to scale a MapReduce application (i.e., Word Revert Index) running on Corey. The work in Corey is orthogonal to Ostrich. The abstractions in Corey, if available in commodity OSes, could further improve the efficiency of Ostrich due to the reduced time spent in the OS kernel.

Thread clustering [Tam et al. 2007] schedules threads with similar cache affinity to close cores, thus improves the cache locality. Tiled-MapReduce also aims to improve cache locality, but tries to limit the working set and fits data in a subjob in cache.

8.4. Nested Data Parallelism

Many data-parallel languages and their implementations, for example, NESL [Blelloch 1996], are designed to support nested data parallelism, which is critical for the performance of nested parallel algorithms. Meanwhile, nested data processing has been explored by the database community for several decades, such as Volcano [Graefe 1994] data-flow query processing systems. In the compiler community, Packirisamy and Zhai [2009] design a new algorithm to exploit parallelism for programs with nested loops by statically mapping cores to different levels of loop-nests. We observed that the MapReduce programming model also could be paralleled in two dimensions and it could be

efficiently implemented on multicore architecture. Our work is on the programming-model level and introduces several additional optimizations to reduce the pressure on the system resources, including main memory, caches, and processors.

9. CONCLUSION

Multicore is prevalent and it is important to harness the power of the likely abundant CPU cores. MapReduce is a promising programming model for multicore platforms to fully utilize the power of such processing resources.

This article argued that the environmental differences between clusters and multicore open new design spaces and optimization opportunities to improve performance of MapReduce on multicore. Based on the observation, this article proposed Tiled-MapReduce, that uses the "tiling strategy" to partition a large MapReduce job into a number of small subjobs and handles the subjobs iteratively. This article also explored several optimizations otherwise impossible for MapReduce, to improve the memory, cache, and CPU efficiency. We also demonstrated that Tiled-MapReduce can support fine-grained fault tolerance and two new computing models. Experimental results showed that our implementation, namely Ostrich, outperforms Phoenix by up to 3.07x and saves up to 87.6% memory. Our profiling results confirmed that the improvement comes from the reduced memory footprint, better data locality, and task parallelism. The overhead of supporting new computing models is also small.

REFERENCES

Ahmad, F., Lee, S., Thottethodi, M., and Vijaykumar, T. N. 2011a. Mapreduce benchmarks. http://web.ics.purdue.edu/~fahmad/benchmarks.htm.

Ahmad, F., Lee, S., Thottethodi, M., and Vijaykumar, T. N. 2011b. MapReduce with communication overlap (MaRCO). Tech. rep. ECE-TR-413, Electrical and Computer Engineering, Purdue University.

Ahmad, F., Chakradhar, S. T., Raghunathan, A., and Vijaykumar, T. N. 2012. Tarazu: Optimizing mapreduce on heterogeneous clusters. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12)*. ACM Press, New York, 61–74.

AMD 2013. Codeanalyst performance analyzer. http://developer.amd.com/tools/codeanalyst/pages/default.aspx.

Ananthanarayanan, G., Agarwal, S., Kandula, S., Greenberg, A., Stoica, I., Harlan, D., and Harris, E. 2011. Scarlett: Coping with skewed content popularity in mapreduce clusters. In *Proceedings of the 6th Conference on Computer systems (EuroSys'11)*. ACM Press, New York, 287–300.

Babu, S. 2010. Towards automatic optimization of mapreduce programs. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC'10)*. ACM Press, New York, 137–142.

Bhatotia, P., Wieder, A., Rodrigues, R., Acar, U. A., and Pasquin, R. 2011. Incoop: Mapreduce for incremental computations. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC'11)*. ACM Press, New York, 7:1–7:14.

Bialecki, A., Cafarella, M., Cutting, D., and O'Malley, O. 2005. Hadoop: A framework for running applications on large clusters built of commodity hardware. http://lucene.apache.org/hadoop.

Blelloch, G. E. 1996. Programming parallel algorithms. *Comm. ACM 39*, 85–97.

Borkar, S. 2007. Thousand core chips: A technology perspective. In *Proceedings of the 44th Annual Design Automation Conference (DAC'07)*. ACM Press, New York, 746–749.

Borthakur, D. 2013. The hadoop distributed file system: Architecture and design. http://hadoop.apache.org/hdfs/docs/current/hdfs_design.html.

Boyd-Wickizer, S., Chen, H., Chen, R., Mao, Y., Kaashoek, F., Morris, R., Pesterev, A., Stein, L., Wu, M., Dai, Y., Zhang, Y., and Zhang, Z. 2008. Corey: An operating system for many cores. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, 43–57.

Chen, R., Chen, H., and Zang, B. 2010. Tiled-mapreduce: Optimizing resource usages of data-parallel applications on multicore with tiling. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT'10)*. ACM Press, New York, 523–534.

Chu, C. T., Kim, S. K., Lin, Y. A., Yu, Y., Bradski, G. R., Ng, A. Y., and Olukotun, K. 2006. Map-reduce for machine learning on multicore. In *Proceedings of Neural Information Processing Systems Conference (NIPS'06)*. MIT Press, 281–288.

Coleman, S. and McKinley, K. S. 1995. Tile size selection using cache organization and data layout. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'95)*. ACM Press, New York, 279–290.

Condie, T., Conway, N., Alvaro, P., Hellerstein, J. M., Elmeleegy, K., and Sears, R. 2010. Mapreduce online. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation (NSDI'10)*. USENIX Association, 21–21.

Dean, J. and Ghemawat, S. 2008. Mapreduce: Simplified data processing on large clusters. *Commun. ACM 51*, 107–113.

de Kruijf, M. and Sankaralingam, K. 2007. MapReduce for the cell be architecture. Tech. rep. CS-TR-2007, Computer Sciences, University of Wisconsin.

Ekanayake, J., Pallickara, S., and Fox, G. 2008. Mapreduce for data intensive scientific analyses. In *Proceedings of the 4th IEEE International Conference on eScience*. IEEE Computer Society, 277–284.

Ekanayake, J., Li, H., Zhang, B., Gunarathne, T., Bae, S.-H., Qiu, J., and Fox, G. 2010. Twister: A runtime for iterative mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC'10)*. ACM Press, New York, 810–818.

Evans, J. 2013. jemalloc. http://www.canonware.com/jemalloc/.

Frigo, M., Leiserson, C. E., and Randall, K. H. 1998. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'98)*. ACM Press, New York, 212–223.

Golub, G. and Van Loan, C. 1996. *Matrix Computations*. Johns Hopkins University Press.

Graefe, G. 1994. Volcano, an extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Engin. 6,* 1, 120–135.

Gray, J. 2013. http://www.hpl.hp.com/hosted/sortbenchmark.

He, B., Fang, W., Luo, Q., Govindaraju, N. K., and Wang, T. 2008. Mars: A mapreduce framework on graphics processors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT'08)*. ACM Press, New York, 260–269.

Hellerstein, J. M., Haas, P. J., and Wang, H. J. 1997. Online aggregation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'97)*. ACM Press, New York, 171–182.

Hong, C., Chen, D., Chen, W., Zheng, W., and Lin, H. 2010. Mapcg: Writing parallel program portable between cpu and gpu. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT'10)*. ACM Press, New York, 217–226.

Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D. 2007. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007 (EuroSys'07)*. ACM Press, New York, 59–72.

Khronos Group. 2009. Opencl overview. http://www.khronos.org/opencl/.

Kim, J., Seo, S., Lee, J., Nah, J., Jo, G., and Lee, J. 2011. Opencl as a programming model for gpu clusters. In *Proceedings of the 24th International Workshop on Languages and Compilers for Parallel Computing (LCPC'11)*.

Lee, J., Kim, J., Seo, S., Kim, S., Park, J., Kim, H., Dao, T. T., Cho, Y., Seo, S. J., Lee, S. H., Cho, S. M., Song, H. J., Suh, S.-B., and Choi, J.-D. 2010. An opencl framework for heterogeneous multicores with local memory. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT'10)*. ACM Press, New York, 193–204.

Lee, R., Luo, T., Huai, Y., Wang, F., He, Y., and Zhang, X. 2011. Ysmart: Yet another sql-to-mapreduce translator. In *Proceedings of the 31st International Conference on Distributed Computing Systems (ICDCS'11)*. IEEE Computer Society, 25–36.

Levon, J. 2004. *OProfile Manual*. Victoria University of Manchester. http://oprofile.sourceforge.net/doc/.

Linderman, M. D., Collins, J. D., Wang, H., and Meng, T. H. 2008. Merge: A programming model for heterogeneous multi-core systems. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'08)*. ACM Press, New York, 287–296.

Logothetis, D., Trezzo, C., Webb, K. C., and Yocum, K. 2011. In-situ mapreduce for log processing. In *Proceedings of the USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'11)*. USENIX Association.

Mao, Y., Morris, R., and Kaashoek, F. 2010. Optimizing MapReduce for Multicore Architectures. Tech. rep. MIT-CSAIL-TR-2010-020, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology.

Matsunaga, A., Tsugawa, M., and Fortes, J. 2008. Cloudblast: Combining mapreduce and virtualization on distributed resources for bioinformatics applications. In *Proceedings of the 4th IEEE International Conference on eScience*. IEEE Computer Society, 222–229.

Mazières, D., Kaminsky, M., Kaashoek, M. F., and Witchel, E. 1999. Separating key management from file system security. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP'99)*. ACM Press, New York, 124–139.

Olston, C., Reed, B., Srivastava, U., Kumar, R., and Tomkins, A. 2008. Pig latin: A not-so-foreign language for data processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'08)*. ACM Press, New York, 1099–1110.

Packirisamy, V. and Zhai, A. 2009. Exploiting tls parallelism at multiple loop-nest levels. In *Proceedings of the 15th International Conference on Parallel and Distributed Systems (ICPADS'09)*. IEEE Computer Society, 205–212.

Pike, R., Dorward, S., Griesemer, R., and Quinlan, S. 2005. Interpreting the data: Parallel analysis with sawzall. *Sci. Program. 13*, 277–298.

Popa, L., Budiu, M., Yu, Y., and Isard, M. 2009. Dryadinc: Reusing work in large-scale computations. In *Proceedings of the Conference on Hot Topics in Cloud Computing (HotCloud'09)*. USENIX Association, 21–21.

Power, R. and Li, J. 2010. Piccolo: Building fast, distributed programs with partitioned tables. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association, 1–14.

Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G., and Kozyrakis, C. 2007. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proceedings of the IEEE 13th International Symposium on High Performance Computer Architecture (HPCA'07)*. IEEE Computer Society, 13–24.

Shan, Y., Wang, B., Yan, J., Wang, Y., Xu, N., and Yang, H. 2010. Fpmr: Mapreduce framework on fpga. In *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'10)*. ACM Press, New York, 93–102.

Song, X., Chen, H., Chen, R., Wang, Y., and Zang, B. 2011. A case for scaling applications to many-core with os clustering. In *Proceedings of the 6nd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys'11)*. ACM Press, New York.

Stuart, J. A., Chen, C.-K., Ma, K.-L., and Owens, J. D. 2010. Multi-gpu volume rendering using mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC'10)*. ACM Press, New York, 841–848.

Talbot, J., Yoo, R. M., and Kozyrakis, C. 2011. Phoenix++: Modular mapreduce for shared-memory systems. In *Proceedings of the 2nd International Workshop on MapReduce and its Applications (MapReduce'11)*. ACM Press, New York, 9–16.

Tam, D., Azimi, R., and Stumm, M. 2007. Thread clustering: Sharing-aware scheduling on smp-cmp-smt multiprocessors. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007 (EuroSys'07)*. ACM Press, New York, 47–58.

Thusoo, A., Sarma, J. S., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P., and Murthy, R. 2009. Hive: A warehousing solution over a map-reduce framework. *Proc. VLDB Endow. 2*, 1626–1629.

Valiant, L. G. 1990. A bridging model for parallel computation. *Comm. ACM 33*, 103–111.

Waingold, E., Taylor, M., Sarkar, V., Lee, W., Lee, V., Kim, J., Frank, M., Finch, P., Devabhaktuni, S., Barua, R., et al. 1997. Baring it all to software: The raw machine. *IEEE Comput. 30,* 9, 86–93.

Wikimedia. 2013. Downloads. http://dumps.wikimedia.org/.

Xiao, Z., Chen, H., and Zang, B. 2011. A hierarchical approach to maximizing mapreduce efficiency. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT'11)*. 167–168.

Yang, C., Yen, C., Tan, C., and Madden, S. R. 2010. Osprey: Implementing mapreduce-style fault tolerance in a shared-nothing distributed database. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE'10)*. IEEE Computer Society, 657–668.

Yang, H.-C., Dasdan, A., Hsiao, R.-L., and Parker, D. S. 2007. Map-reduce-merge: Simplified relational data processing on large clusters. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'07)*. ACM Press, New York, 1029–1040.

Yoo, R. M., Romano, A., and Kozyrakis, C. 2009. Phoenix rebirth: Scalable mapreduce on a large-scale shared-memory system. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC'09)*. IEEE Computer Society, 198–207.

Yu, Y., Isard, M., Fetterly, D., Budiu, M., Erlingsson, U., Gunda, P. K., and Currey, J. 2008. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In

Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08). USENIX Association, 1–14.

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. 2010. Spark: Cluster computing with working sets. In *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud'10)*. USENIX Association, 10–10.

Zaharia, M., Konwinski, A., Joseph, A. D., Katz, R., and Stoica, I. 2008. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08)*. USENIX Association, 29–42.

Received January 2012; revised August 2012; accepted October 2012