Themis: An I/O-Efficient MapReduce

Alexander Rasmussen
arasmuss@cs.ucsd.edu
UC San Diego

Michael Conley
mconley@cs.ucsd.edu
UC San Diego

Rishi Kapoor
rkapoor@cs.ucsd.edu
UC San Diego

Vinh The Lam
vtlam@cs.ucsd.edu
UC San Diego

George Porter
gmporter@cs.ucsd.edu
UC San Diego

Amin Vahdat
vahdat@cs.ucsd.edu
UC San Diego & Google, Inc.

ABSTRACT

"Big Data" computing increasingly utilizes the MapReduce programming model for scalable processing of large data collections. Many MapReduce jobs are I/O-bound, and so minimizing the number of I/O operations is critical to improving their performance. In this work, we present Themis, a MapReduce implementation that reads and writes data records to disk exactly twice, which is the minimum amount possible for data sets that cannot fit in memory.

In order to minimize I/O, Themis makes fundamentally different design decisions from previous MapReduce implementations. Themis performs a wide variety of MapReduce jobs – including click log analysis, DNA read sequence alignment, and PageRank – at nearly the speed of TritonSort's record-setting sort performance [29].

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*

General Terms

Design, Performance, Experimentation

1. INTRODUCTION

Building efficient and scalable data processing systems is a challenging – and increasingly important – problem. *Scale-out* software systems implementing the MapReduce programming model, such as Google's MapReduce [8] and Apache Hadoop [39], have made great strides in providing efficient system architectures for these workloads [40]. However, a significant gap remains between the delivered performance of these systems and the potential performance available from the underlying hardware [29, 2].

Our recent experience building TritonSort [29], a large-scale sorting system, shows that an appropriately balanced implementation can realize orders of magnitude improvement in throughput and efficiency. Translating these types of gains to more general-purpose data processing systems will help close this efficiency gap, allowing more work to be performed with the same hardware, or the same amount of work to be performed with less hardware. This improved efficiency will result in substantially lowered system cost, energy usage, and management complexity.

Given that many MapReduce jobs are I/O-bound, an efficient MapReduce system must aim to minimize the number of I/O operations it performs. Fundamentally, every MapReduce system must perform at least two I/O operations per record when the amount of data exceeds the amount of memory in the cluster [1]. We refer to a system that meets this lower-bound as having the "2-IO" property. Any data processing system that does not have this property is doing more I/O than it needs to. Existing MapReduce systems incur additional I/O operations in exchange for simpler and more fine-grained fault tolerance.

In this paper, we present Themis[1], an implementation of MapReduce designed to have the 2-IO property. Themis accommodates the flexibility of the MapReduce programming model while simultaneously delivering high efficiency. It does this by considering fundamentally different points in the design space than existing MapReduce implementations:

1. Eliminating task-level fault tolerance: At the scale of tens of thousands of servers, failures are common, and so MapReduce was designed with a strong task-level fault tolerance model. However, more aggressive fault tolerance gains finer-grained restart at the expense of lower overall performance. Interestingly, many Hadoop users report cluster sizes of under 100 nodes [14], much smaller than those deployed by MapReduce's early adopters. In 2011, Cloudera's VP of Technology Solutions stated that the mean size of their clients' Hadoop clusters is 200 nodes, with the median size closer to 30 [23]. At this moderate scale, failures are much less common, and aggressive fault tolerance is wasteful in the common case. Foregoing task-level fault tolerance permits a design that achieves the 2-IO property. When a job experiences a failure, Themis simply re-executes it. This optimistic approach to fault tolerance enables Themis to aggressively pipeline record processing without unnecessarily materializing intermediate results to disk. As we will show, for moderate cluster sizes this approach has the counter-intuitive effect of improving performance despite the occasional job re-execution.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SOCC'12, October 14-17, 2012, San Jose, CA USA
Copyright 2012 ACM 978-1-4503-1761-0/12/10 ...$15.00.

[1] Themis is a Titan in Greek mythology who is tasked with creating balance, order and harmony.

2. Dynamic, adaptive memory allocation: To maintain the 2-IO property, Themis must process a record completely once it is read from disk. This prevents Themis from putting records back on disk in response to memory pressure through swapping or writing spill files. Themis implements a policy-based, application-level memory manager that provides fine-grained sharing of memory between operators processing semi-structured, variably-sized records. This allows it to support datasets with as much as a factor of 10^7 skew between record sizes while maintaining the 2-IO property.

3. Central management of shuffle and disk I/O: Themis uses a centralized, per-node disk scheduler that ensures that records from multiple sources are written to disk in large batches to reduce disk seeks. Themis delivers nearly sequential disk I/O across a variety of MapReduce jobs, even for workloads that far exceed the size of main memory.

To validate our design, we have written a number of MapReduce programs on Themis, including a web user session tracking application, PageRank, n-gram counting, and a DNA read sequence alignment application. We found that Themis processes these jobs at nearly the per-node performance of TritonSort's record-setting sort run and nearly the maximum sequential speed of the underlying disks.

2. MOTIVATION

Themis achieves the 2-IO property by making different design decisions than those made in Google's and Hadoop's MapReduce implementations. In this section, we discuss our motivations for making these decisions.

2.1 Target Deployment Environments

A large number of "Big Data" clusters do not approach the size of warehouse-scale data centers like those at Google and Microsoft because moderately-sized clusters (10s of racks or fewer) are increasingly able to support important real-world problem sizes. The storage capacity and number of CPU cores in commodity servers are both increasing rapidly. In Cloudera's reference system design [7], in which each node has 16 or more disks, a petabyte worth of 1TB drives fits into just over three racks, or about 60 nodes. Coupled with the emergence of affordable 10 Gbps Ethernet at the end host and increasing bus speeds, data can be packed more densely than ever before while keeping disk I/O as the bottleneck resource. This implies that fewer servers are required for processing large amounts of data for I/O-bound workloads. We now consider the implications of this increased density on fault tolerance.

2.2 Fault Tolerance for "Dense" Clusters

A key principle of Themis's design is that it performs job-level, rather than task-level, fault tolerance. Job-level fault tolerance allows for much more aggressive operator pipelining than task-level fault tolerance can achieve while still maintaining the 2-IO property. However, it is not self-evident that the overhead of re-executing failed jobs does not cancel any performance gained by this aggressive pipelining. In this section, we show not only that job-level fault tolerance is a feasible approach for moderately-sized clusters, but also that there are significant potential performance benefits for using job-level fault tolerance in these environments.

Component	Failure rates
Node	4.3 months
Disk	2-4% annualized
Rack	10.2 years

Table 1: Component-level failure rates observed in a Google data center as reported in [12].

Understanding the expected impact of failures is critical to choosing the appropriate fault tolerance model. MapReduce was designed for clusters of many thousands of machines running inexpensive, failure-prone commodity hardware [8]. For example, Table 1 shows component-level mean-time to failure (MTTF) statistics in one of Google's data centers [12]. Google's failure statistics are corroborated by similar studies of hard drive [27, 33] and node [25, 32] failure rates. At massive scale, there is a high probability that some portion of the cluster will fail during the course of a job. To understand this probability, we employ a simple model [3], shown in Equation 1, to compute the likelihood that a node in a cluster of a particular size will experience a failure during a job:

$$P(N, t, MTTF) = 1 - e^{-N \cdot t/MTTF} \tag{1}$$

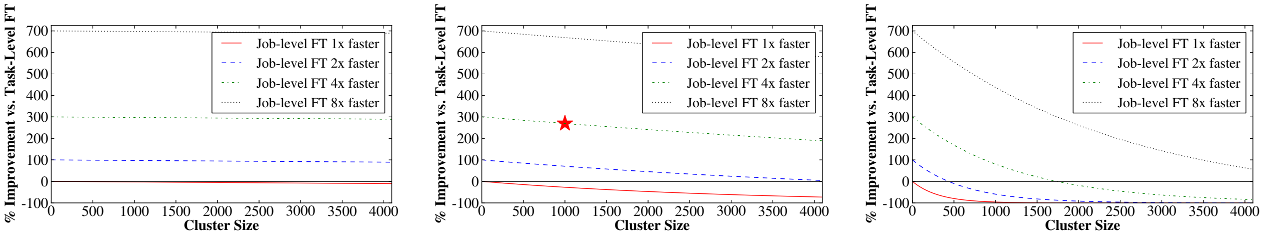
The probability of a failure occurring in the next t seconds is a function of (1) the number of nodes in the cluster, N, (2) t, and (3) the mean time to failure of each node, $MTTF$, taken from the node-level failure rates in Table 1. This model assumes that nodes fail with exponential (Pareto) probability, and we simplify our analysis by considering node failures only. We do this because disk failures can be made rare by using node-level mechanisms (i.e., RAID), and correlated rack failures are likely to cripple the performance of a cluster with only a few racks. Based on the above model, in a 100,000 node data center, there is a 93% chance that a node will fail during any five-minute period. On the other hand, in a moderately-sized cluster (e.g., 200 nodes, the average Hadoop cluster size as reported by Cloudera), there is only a 0.53% chance of encountering a node failure during a five-minute window, assuming the MTTF rates in Table 1.

This leads to the question of whether smaller deployments benefit from job-level fault tolerance, where if any node running a job fails the entire job restarts. Intuitively, this scheme will be more efficient overall when failures are rare and/or jobs are short. In fact, we can model the expected completion time of a job $S(p, T)$ as:

$$S(p, T) = T\left(\frac{p}{1-p} + 1\right) \tag{2}$$

where p is the probability of a node in the cluster failing, and T is the runtime of the job (a derivation of this result is given in Appendices A and B). This estimate is pessimistic, in that it assumes that jobs fail just before the end of their execution. By combining equations 1 and 2, we can compute the expected benefit–or penalty–that we get by moving to job-level fault tolerance. Modeling the expected runtime of a job with task-level fault tolerance is non-trivial, so we instead compare to an error-free baseline in which the system's performance is not affected by node failure. This comparison underestimates the benefit of job-level fault tolerance.

Figure 1 shows the expected performance benefits of job-level fault tolerance compared to the error-free baseline. More formally, we measure performance benefit as $S(P(\cdot), T_{job})/T_{task}$, where T_{job} is the time a job on an error-



(a) 5-minute job

(b) 1-hour job (see text below for explanation of marked point)

(c) 10-hour job

Figure 1: A lower-bound of the expected benefit of job-level fault tolerance for varying job durations and cluster sizes. Given that an error-free execution of a job with task-level fault tolerance takes five minutes (a), an hour (b), or ten hours (c) to complete, we explore the expected performance improvement gained from job-level fault tolerance if an error-free run of the job executes 1, 2, 4, and 8x faster with job-level fault tolerance than it does with task-level fault tolerance.

free cluster takes to execute with job-level fault tolerance and T_{task} is the time the same job takes to execute with task-level fault tolerance.

The benefits of job-level fault tolerance increase as the error-free performance improvement made possible by moving to job-level fault tolerance (i.e. T_{task}/T_{job}) increases. For example, if T_{task}/T_{job} is 4, T_{task} is one hour and we run on a cluster of 1,000 nodes, we can expect Themis to complete the job 240% faster than the task-level fault tolerant alternative on average; this scenario is marked with a star in Figure 1(b). There are also situations in which job-level fault tolerance will significantly under-perform task-level fault tolerance. For example, if T_{task}/T_{job} is 2, Themis will under-perform a system with task-level fault tolerance for clusters bigger than 500 nodes. From this, we make two key observations: for job-level fault tolerance to be advantageous, the cluster has to be moderately-sized, and Themis must significantly outperform the task-level alternative.

In the next section, we describe key challenges in designing a system that meets these high performance requirements while maintaining the 2-IO property.

3. THE CHALLENGE OF SKEW

One of MapReduce's attractive properties is its ability to handle semi-structured and variably-sized data. This variability makes maintaining the 2-IO property a challenge. In this section, we describe two sources of variability and the resulting requirements they place on our design.

An input dataset can exhibit several different kinds of *skew*, which simply refers to variability in its structure and content. These include:

Record Size Skew: In systems with semi-structured or unstructured data, some records may be much larger than others. This is called *record size skew*. In extreme cases, a single record may be gigabytes in size.

Partitioning Skew: Data that is not uniformly distributed across its keyspace exhibits *partitioning skew*. This can cause some nodes to process much more data than others if the data is naïvely partitioned across nodes, creating stragglers [9]. Handling skew in MapReduce is complicated by the fact that the distribution of keys in the data produced by a `map` function is often not known in advance. Existing MapReduce implementations handle partitioning skew by spilling records to disk that cannot fit into memory.

Computational Skew: In a dataset exhibiting *computational skew*, some records take much longer than average to process. Much of the work on mitigating computational skew in MapReduce involves exploiting the nature of the particular problem and relying on a degree of user guidance [16] or proactively re-partitioning the input data for a task [17]. As the focus of our work is I/O-bound jobs, we do not consider computational skew in this work.

Performance Heterogeneity: The performance of a population of identical machines can vary significantly; the reasons for this heterogeneity are explored in [31]. In addition, clusters are rarely made up of a homogeneous collection of machines, due both to machine failures and planned incremental upgrades. While we believe that the techniques presented in this work can be applied to heterogeneous clusters, we have not evaluated Themis in such a setting.

To handle record skew, Themis dynamically controls its memory usage, which we describe in Section 5. Themis adopts a sampling-based skew mitigation technique to minimize the effects of partitioning skew. We discuss this mitigation technique in Section 6.

4. SYSTEM ARCHITECTURE

In this section, we describe the design of Themis.

4.1 Core Architecture

Themis reuses several core runtime components that were used to build the TritonSort [29] sorting system. Like TritonSort, Themis is written as a sequence of *phases*, each of which consists of a directed dataflow graph of *stages* connected by FIFO queues. Each stage consists of a number of *workers*, each running as a separate thread.

4.2 MapReduce Overview

Unlike existing MapReduce systems, which executes `map` and `reduce` tasks concurrently in waves, Themis implements the MapReduce programming model in three phases of operation, summarized in Table 2. Phase zero, described in Section 6, is responsible for sampling input data to determine the distribution of record sizes as well as the distribution of keys. These distributions are used by subsequent phases to

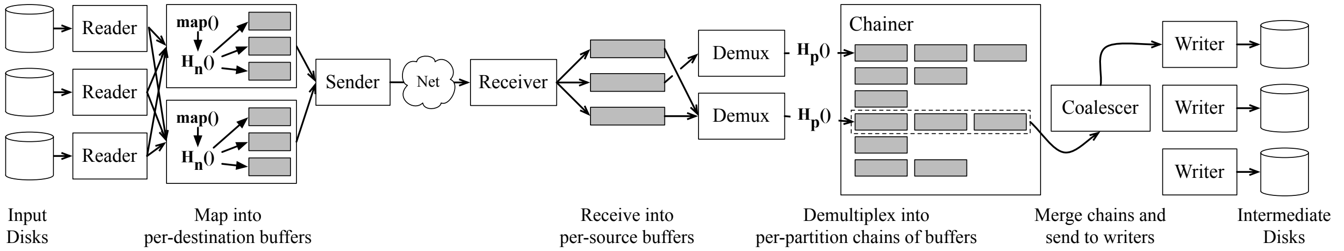

Reader
Reader
Reader
map()
Hn()
map()
Hn()
Sender
Net
Receiver
Demux
Demux
Hp()
Hp()
Chainer
Coalescer
Writer
Writer
Writer
Input Disks
Map into per-destination buffers
Receive into per-source buffers
Demultiplex into per-partition chains of buffers
Merge chains and send to writers
Intermediate Disks

Figure 2: Stages of Phase One (Map/Shuffle) in Themis

Phase	Description	Required?
0	Skew Mitigation	Optional
1	`map()` and shuffle	Required
2	sort and `reduce()`	Required

Table 2: Themis's three stage architecture

minimize partitioning skew. Phase one, described in Section 4.3, is responsible for applying the `map` function to each input record, and routing its output to an appropriate partition in the cluster. This is the equivalent of existing systems' map and shuffle phases. Phase two, described in Section 4.4, is responsible for sorting and applying the `reduce` function to each of the intermediate partitions produced in phase one. At the end of phase two, the MapReduce job is complete.

Phase one reads each input record and writes each intermediate record exactly once. Phase two reads each intermediate partition and writes its corresponding output partition exactly once. Thus, Themis has the 2-IO property.

4.3 Phase One: Map and Shuffle

Phase one is responsible for implementing both the `map` operation as well as shuffling records to their appropriate intermediate partition. Each node in parallel implements the stage graph pipeline shown in Figure 2.

The **Reader** stage reads records from an input disk and sends them to the **Mapper** stage, which applies the `map` function to each record. As the `map` function produces intermediate records, each record's key is hashed to determine the node to which it should be sent and placed in a per-destination buffer that is given to the sender when it is full. The **Sender** sends data to remote nodes using a round-robin loop of short, non-blocking `send()` calls. We call the Reader to Sender part of the pipeline the "producer-side" pipeline.

The **Receiver** stage receives records from remote nodes over TCP using a round-robin loop of short, non-blocking `recv()` calls. We implemented a version of this stage that uses `select()` to avoid unnecessary polling, but found that its performance was too unpredictable to reliably receive all-to-all traffic at 10Gbps. The receiver places incoming records into a set of small per-source buffers, and sends those buffers to the Demux stage when they become full.

The **Demux** stage is responsible for assigning records to partitions. It hashes each record's key to determine the partition to which it should be written, and appends the record to a small per-partition buffer. When that buffer becomes full, it is emitted to the **Chainer** stage, which links buffers for each partition into separate *chains*. When chains exceed a pre-configured length, which we set to 4.5 MB to avoid doing small writes, it emits them to the **Coalescer** stage. The **Coalescer** stage merges chains together into a single large buffer that it sends to the **Writer** stage, which appends buffers to the appropriate partition file. The combination of the Chainer and Coalescer stages allows buffer memory in front of the Writer stage to be allocated to partitions in a highly dynamic and fine-grained way. We call the Receiver to Writer part of the pipeline the "consumer-side" pipeline.

A key requirement of the consumer-side pipeline is to perform large, contiguous writes to disk to minimize seeks and provide high disk bandwidth. We now describe a node-wide, application-driven disk scheduler that Themis uses to ensure that writes are large.

Each writer induces back-pressure on chainers, which causes the per-partition chains to get longer. In this way, data gets buffered within the chainer. This buffering can grow very large–to over 10GB on a machine with 24GB of memory. The longer a chain becomes, the larger the corresponding write will be. We limit the size of a chain to 14MB, to prevent very large writes from restricting pipelining. The large writes afforded by this scheduler allow Themis to write at nearly the sequential speed of the disk. [29] provides a detailed evaluation of the relationship between write sizes and system throughput.

Signaling back-pressure between the chainer and the writer stage is done by means of *write tokens*. The presence of a write token for a writer indicates that it can accept additional buffers. When the writer receives work, it removes its token, and when it finishes, it returns the token. Tokens are also used to prevent the queues between the chainer and writer stages from growing without bound.

4.4 Phase Two: Sort and Reduce

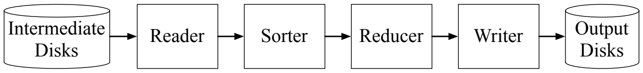


Figure 3: Stages of Phase Two (Sort/Reduce) in Themis

By the end of phase one, the map function has been applied to each input record, and the records have been grouped into partitions and stored on the appropriate node so that all records with the same key are stored in a single partition. In phase two, each partition must be sorted by key, and the `reduce` function must be applied to groups of records with the same key. The stages that implement phase two are shown in Figure 3.

There is no network communication in phase two, so nodes process their partitions independently. Entire partitions are read into memory at once by the **Reader** stage. A **Sorter**

	TritonSort	Themis	Used in Phase			Subject to deadlock?
			0	1	2	
Pool	✓	✓	✓	✓		
Quota		✓	✓	✓		
Constraint		✓			✓	✓

Table 3: A comparison of Themis's memory allocator implementations.

stage sorts these partitions by key, keeping the result in memory. The **Reducer** stage applies the `reduce` function to all records sharing a key. Reduced records are sent to the **Writer**, which writes them to disk.

All records with a single key must be stored in the same partition for the `reduce` function to produce correct output. As a result, partitioning skew can cause some partitions to be significantly larger than others. Themis's memory management system allows phase two to process partitions that approach the size of main memory, and its optional skew mitigation phase can reduce partitioning skew without user intervention. We describe these systems in Sections 5 and 6, respectively.

A key feature of Themis's sorter stage is that it can select which sort algorithm is used to sort a buffer on a buffer-by-buffer basis. There is a pluggable *sort strategy* interface that lets developers use different sorting algorithms; currently quicksort and radix sort are implemented. Each sort strategy calculates the amount of scratch space it needs to sort the given buffer, depending on the buffer's contents and the sort algorithm's space complexity. For both quicksort and radix sort, this computation is deterministic. In Themis, radix sort is chosen if the keys are all the same size and the required scratch space is under a configurable threshold; otherwise, quicksort is used.

5. MEMORY MANAGEMENT AND FLOW CONTROL

Themis relies on a dynamic and flexible memory management system to partition memory between operators. Themis's memory manager actually serves two distinct purposes: (1) it enables resource sharing between operators, and (2) it supports enforcing back-pressure and flow control. In the first case, Themis requires flexible use of memory given our desire to support large amounts of record size skew while maintaining the 2-IO property. In the second case, individual stages in the Themis pipeline naturally run at different speeds (e.g., the network is 10 Gbps, whereas the disk subsystem only supports writing at approximately 5.5 Gbps), and so back-pressure and flow control are needed to prevent faster stages from starving slower stages for resources.

Themis supports a single memory allocation interface with pluggable memory policies. We first describe the memory allocator's interface, and then describe the three policies that we've implemented.

5.1 Memory allocation interface

Worker stages in Themis allocate space for buffers and other necessary scratch space using a unified and simple memory allocator interface, shown in Table 4.

Memory allocators can be assigned on a stage-by-stage basis, but in the current implementation we assume that memory regions are allocated and deallocated by the same allocator. The `allocate` call blocks until the underlying memory allocation policy satisfies the allocation, which can be an unbounded amount of time. As we will see, this simple mechanism, paired with one of three memory policies, provides for both resource sharing as well as flow control. We now examine each of these polices.

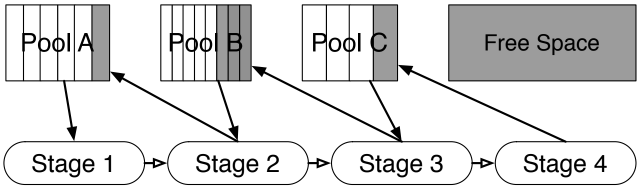


Figure 4: A diagrammatic overview of pool-based memory management. Note that memory in each pool is divided into fixed-size regions, and that any memory not allocated to pools cannot be utilized by stages.

5.2 Policy 1: Pool-Based Management

The first policy we consider is a "pool" memory policy, which is inherited from TritonSort [29]. A memory pool is a set of pre-allocated buffers that is filled during startup. Buffers can be checked out from a pool, and returned when they are finished being used as illustrated in Figure 4. When a worker tries to check out a buffer from an empty pool, it blocks until another worker returns a buffer to that pool. The pool memory policy has the advantage that all memory allocation is done at startup, avoiding allocation during runtime. Through efficient implementation, the overhead of checking out buffers can be very small. This is especially useful for stages that require obtaining buffers with very low latency, such as the Receiver stage, which obtains buffers to use in receiving data from the network. The receiver receives uninterpreted bytes from network sockets into small, fixed-size byte buffers. These buffers are passed to a subsequent stage, which converts them into buffers containing complete records. For this reason, the receiver can use pool-based management while still supporting record-size skew.

Pools can be used to provide resource sharing between workers by giving each of those workers a reference to a single pool. The producer-consumer relationship between pairs of stages also provides a form of flow control; the upstream stage checking out buffers can only produce work at the rate at which the downstream stage can return them to the pool. However, pools have several disadvantages. First, the buffers in a pool are all fixed-size, and so the pool memory policy supports very limited amounts of data skew. By carving memory up into fixed-size pools, the maximum record size supported by this policy is limited to the size of the smallest pool. Additionally, buffer pools reserve a fixed amount of memory for a particular pair of stages. One consequence of this is a loss of flexibility; if one stage temporarily needs more memory than usual (e.g., if it is handling a large record), it cannot "borrow" that memory from another stage due to the static partitioning of memory across pools.

5.3 Policy 2: Quota-Based Management

While the pool memory policy is simple, it is quite inflexible, and does not handle skewed record sizes very well.

Function	Description
`CallerID registerCaller(Worker worker)`	Register `worker` with the allocator
`void* allocate(CallerID caller, uint64_t size)`	allocate a memory region of `size` bytes for `caller`
`void deallocate(void* memory)`	deallocate `memory` that was allocated by this allocator

Table 4: A summary of the Themis memory allocator API

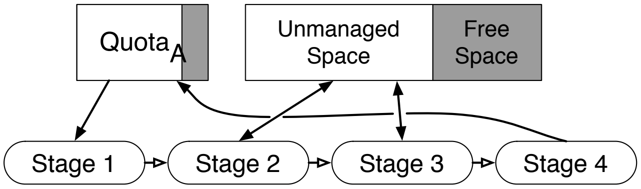


Figure 5: A diagrammatic overview of quota-based memory management. In this figure, $Quota_A$ provides a memory quota between Stage 1 and Stage 4. Stages 2 and 3 use unmanaged memory created with standard `malloc` and `free` syscalls.

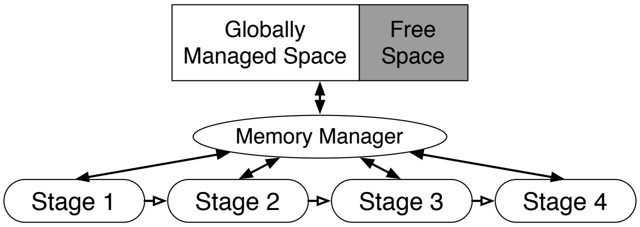


Figure 6: A diagrammatic overview of constraint-based memory management. All stages' memory requests are satisfied by a central memory manager that schedules these requests according to the stage graph's structure.

The quota-based memory policy is designed to support more flexible memory allocation while still providing flow control. At a high level, the quota policy ensures that stages producing records do not overwhelm stages that eventually consume them. For example, most of our evaluation is writer limited, and so we want to ensure that the receiver stage, which produces records received from the network, does not overwhelm the writer stage, which is the bottleneck.

Themis has three such producer-consumer pairs: between the reader and the mapper (with the mapper acting as the consumer), between the mapper and the sender (with the mapper acting as the producer), and between the receiver and the writer. The mapper acts as both a consumer and a producer, since it is the only stage in the phase one pipeline that creates records as directed by the `map` function that were not read by the reader.

Quotas are enforced by the queues between stages. A quota can be viewed as the amount of memory that the pipeline between a producer and a consumer can use. When a producer stage pushes a buffer into the pipeline, the size of that buffer is debited from the quota. When a consumer stage consumes that buffer, the buffer's size is added back to the quota. If a producer is about to exceed the quota, then it blocks until the consumer has consumed sufficient memory. Quota-based allocation is illustrated in Figure 5.

Quota-based memory management dramatically reduces the number of variables that need to be tuned relative to the pool-based memory policy. One need only adjust the quota allocations present between pairs of stages, rather than the capacity of a much larger number of buffer pools. Additionally, stages that are not producers and consumers do not need to do any form of coordination, which makes their memory allocations very fast.

Quota-based management assumes that any scratch space or additional memory needed by stages between the producer and consumer is accounted for in the quota. This is to prevent intermediate stages from exceeding the total amount of memory, since their memory accesses are not tracked. It also tacitly assumes that the size of a buffer being produced cannot exceed the size of the quota. This is much less restrictive than a pool-based approach, as quotas are typically several gigabytes.

5.4 Policy 3: Constraint-Based Management

In situations where the amount of memory used by stages to process records cannot be determined in advance, quota-based systems are not ideal for providing flow control. In these situations, Themis uses a more heavyweight, constraint-based memory management policy.

In the constraint-based memory policy, the total amount of memory in use by workers is tracked centrally in the memory allocator. If a worker requests memory, and enough memory is available, that request is granted immediately. Otherwise, the worker's request is added to a per-worker queue of outstanding requests and the worker sleeps on a condition variable until the request can be satisfied. Constraint-based allocation is illustrated in Figure 6.

When multiple workers have outstanding unsatisfied allocation requests, the memory allocator prioritizes worker requests based on a worker's distance in the stage graph to a stage that consumes records. The producer-side pipeline measures distance to the sender stage, and the consumer-side pipeline measures distance to the writer stage. The rationale behind this decision is that records that are being processed should be completely processed before more work is admitted. This decision is inspired by work on live-lock prevention in routers [22]. In this way, the constraint-based allocator implements flow control based on the structure of the dataflow graph.

While this system places precise upper bounds on the amount of memory present in the system, it requires a great deal of coordination between workers, which requires significant lock contention in our implementation. In effect, the reliance on keeping the amount of available memory consistent requires that all allocation and deallocation requests are processed serially. Hence, constraint-based memory allocation is useful for situations where the number of allocation requests being made is relatively small, but the probability of exceeding available memory in common-case operation is high. Phase two in Themis uses constraint-based memory management for precisely these reasons.

In the constraint-based policy, it is possible that certain allocation interleavings can trigger deadlock. Predicting whether a general dataflow system will deadlock is undecidable [24], and deadlock prevention requires knowledge of data dependencies between stages that we deemed too heavyweight. To addressed the problem of deadlocks, Themis provides a deadlock detector. The deadlock detector periodically probes workers to see if they are waiting for a memory allocation request to complete. If multiple probe cycles pass in which all workers are waiting for an allocation or are idle, the deadlock detector informs the memory allocator that a deadlock has occurred. We have not experienced deadlock using the policy choices described in Table 3 in any of the MapReduce jobs we have evaluated. Efficient ways of handling deadlock is the subject of ongoing work.

In summary, Themis provides a pluggable, policy-driven memory allocation subsystem that provides for flexible resource sharing between stages and workers to handle record size skew while also enabling flow control.

6. SKEW MITIGATION

To satisfy the 2-IO property, Themis must ensure that every partition can be sorted in memory, since an out-of-core sort would induce additional I/Os. In addition, to support parallelism, partitions must be small enough that several partitions can be processed in parallel. Phase zero is responsible for choosing the number of partitions, and selecting a partitioning function to keep each partition roughly the same size. This task is complicated by the fact that the data to be partitioned is generated by the `map` function. Thus, even if the distribution of input data is known, the distribution of intermediate data may not be known. This phase is optional: if the user has knowledge of the intermediate data's distribution, they can specify a custom partitioning function, similar to techniques used in Hadoop.

Phase zero approximates the distribution of intermediate data by applying the `map` function to a subset of the input. If the data is homoscedastic, then a small prefix of the input is sufficient to approximate the intermediate distribution. Otherwise, more input data will need to be sampled, or phase two's performance will decrease. DeWitt et al. [10] formalize the number of samples needed to achieve a given skew with high probability; typically we sample 1 GB per node of input data for nodes supporting 8 TB of input. The correctness of phase two only depends on partitions being smaller than main memory. Since our target partition size is less than 5% of main memory, this means that a substantial sampling error would have to occur to cause job failure. So although sampling does impose additional I/O over the 2-IO limit, we note that it is a small and constant overhead.

Once each node is done sampling, it transmits its sample information to a central coordinator. The coordinator uses these samples to generate a partition function, which is then re-distributed back to each node.

6.1 Mechanism

On each node, Themis applies the `map` operation to a prefix of the records in each input file stored on that node. As the `map` function produces records, the node records information about the intermediate data, such as how much larger or smaller it is than the input and the number of records generated. It also stores information about each intermediate key and the associated record's size. This information varies based on the sampling policy. Once the node is done sampling, it sends that metadata to the coordinator.

The coordinator merges the metadata from each of the nodes to estimate the intermediate data size. It then uses this size, and the desired partition size, to compute the number of partitions. Then, it performs a streaming merge-sort on the samples from each node. Once all the sampled data is sorted, partition boundaries are calculated based on the desired partition sizes. The result is a list of "boundary keys" that define the edges of each partition. This list is broadcast back to each node, and forms the basis of the partitioning function used in phase one.

The choice of sampling policy depends on requirements from the user, and we now describe each policy.

6.2 Sampling Policies

Themis supports the following sampling policies:

(1) Range partitioning: For MapReduce jobs in which the ultimate output of all the reducers must be totally ordered (e.g., sort), Themis employs a range partitioning sampling policy. In this policy, the entire key for each sampled record is sent to the coordinator. A downside of this policy is that very large keys can limit the amount of data that can be sampled because there is only a limited amount of space to buffer sampled records.

(2) Hash partitioning: For situations in which total ordering of `reduce` function output is not required, Themis employs hash partitioning. In this scheme, a hash of the key is sampled, instead of the keys themselves. This has the advantage of supporting very large keys, and allowing Themis to use reservoir sampling [37], which samples data in constant space in one pass over its input. This enables more data to be sampled with a fixed amount of buffer. This approach also works well for input data that is already partially or completely sorted because adjacent keys are likely to be placed in different partitions, which spreads the data across the cluster.

7. EVALUATION

We evaluate Themis through benchmarks of several different MapReduce jobs on both synthetic and real-world data sets. A summary of our results are as follows:

- Themis is highly performant on a wide variety of MapReduce jobs, and outperforms Hadoop by 3x - 16x on a variety of common jobs.
- Themis can achieve nearly the sequential speed of the disks for I/O-bound jobs, which is approximately the same rate as TritonSort's record-setting performance.
- Themis's memory subsystem is flexible, and is able to handle large amounts of data skew while ensuring efficient operation.

7.1 Workloads and evaluation overview

We evaluate Themis on a cluster of HP DL380G6 servers, each with two Intel E5520 CPUs (2.27 GHz), 24 GB of memory, and 16 500GB 7200 RPM 2.5" SATA drives. Each hard drive is configured with a single XFS partition. Each

		Data Size		
Job Name	**Description**	Input	Intermediate	Output
Sort-100G	Uniformly-random sort, 100GB per node	2.16TB	2.16TB	2.16TB
Sort-500G	Uniformly-random sort, 500GB per node	10.8TB	10.8TB	10.8TB
Sort-1T	Uniformly-random sort, 1TB per node	21.6TB	21.6TB	21.6TB
Sort-1.75T	Uniformly-random sort, 1.75TB per node	37.8TB	37.8TB	37.8TB
Pareto-1M	Sort with Pareto-distributed key/value sizes, $\alpha = 1.5$, $x_0 = 100$ (1MB max key/value size)	10TB	10TB	10TB
Pareto-100M	Sort with Pareto-distributed key/value sizes, $\alpha = 1.5$, $x_0 = 100$ (100MB max key/value size)	10TB	10TB	10TB
Pareto-500M	Sort with Pareto-distributed key/value sizes, $\alpha = 1.5$, $x_0 = 100$ (500MB max key/value size)	10TB	10TB	10TB
CloudBurst	CloudBurst (two nodes, aligning `lakewash_combined_v2.genes.nucleotide`)	971.3MB	68.98GB	517.9MB
PageRank-U	PageRank (synthetic uniform graph, 25M vertices, 50K random edges per vertex)	1TB	4TB	1TB
PageRank-PL	PageRank (synthetic graph with power-law vertex in-degree, 250M vertices)	934.7GB	3.715TB	934.7GB
PageRank-WEX	PageRank on WEX page graph	1.585GB	5.824GB	2.349GB
WordCount	Count words in text of WEX	8.22GB	27.74GB	812MB
n-Gram	Count 5-grams in text of WEX	8.22GB	68.63GB	49.72GB
Click-Sessions	Session extraction from 2TB of synthetic click logs	2TB	2TB	8.948GB

Table 5: A description and table of abbreviations for the MapReduce jobs evaluated in this section. Data sizes take into account 8 bytes of metadata per record for key and value sizes

XFS partition is configured with a single allocation group to prevent file fragmentation across allocation groups, and is mounted with the `noatime` flag set. Each server has two HP P410 drive controllers with 512MB on-board cache, as well as a Myricom 10 Gbps network interface. All nodes are connected to a single Cisco Nexus 5596 datacenter switch. All servers run Linux 2.6.32. Our implementation of Themis is written in C++ and is compiled with g++ 4.6.2.

To evaluate Themis at scale, we often have to rely on large synthetically-generated data sets, due to the logistics of obtaining and storing freely-available, large data sets. All synthetic data sets are evaluated on 20 cluster nodes. Non-synthetic data sets are small enough to be evaluated on a single node.

All input and output data is stored on local disks without using any distributed filesystem and without replication. We are actively exploring integrating Themis with storage systems like HDFS, but an evaluation of such integration is the subject of future work.

We evaluate Themis's performance on several different MapReduce jobs. A summary of these jobs is given in Table 5, and each job is described in more detail below.

Sort: Large-scale sorting is a useful measurement of the performance of MapReduce and of data processing systems in general. During a sort job, all cluster nodes are reading from disks, writing to disks, and doing an all-to-all network transfer simultaneously. Sorting also measures the performance of MapReduce independent of the computational complexity of the `map` and `reduce` functions themselves, since both `map` and `reduce` functions are effectively no-ops. We study the effects of both increased data density and skew on the system using sort due to the convenience with which input data that meets desired specifications can be generated. We generate skewed data with a Pareto distribution. The record size in generated datasets is limited by a fixed maximum, which is a parameter given to the job.

WordCount: Word count is a canonical MapReduce job. Given a collection of words, word count's `map` function emits `<word, 1>` records for each word. Word count's `reduce` function sums the occurrences of each word and emits a single `<word, N>` record, where N is the number of times the word occurred in the original collection.

We evaluate WordCount on the 2012-05-05 version of the Freebase Wikipedia Extraction (WEX) [38], a processed dump of the English version of Wikipedia. The complete WEX dump is approximately 62GB uncompressed, and contains both XML and text versions of each page. We run word count on the text portion of the WEX data set, which is approximately 8.2GB uncompressed.

n-Gram Count: An extension of word count, n-gram count counts the number of times each group of n words appears in a text corpus. For example, given "The quick brown fox jumped over the lazy dog", 3-gram count would count the number of occurrences of "The quick brown", "quick brown fox", "brown fox jumped", etc. We also evaluate n-gram count on the text portion of the WEX data set.

PageRank: PageRank is a graph algorithm that is widely used by search engines to rank web pages. Each node in the graph is given an initial rank. Rank propagates through the graph by each vertex contributing a fraction of its rank evenly to each of its neighbors.

PageRank's `map` function is given a `<vertex ID, adjacency list of vertex IDs|initial rank>` pair for each vertex in the graph. It emits `<adjacent vertex ID, rank contribution>` pairs for each adjacent vertex ID, and also re-emits the adjacency list so that the graph can be reconstructed. PageRank's `reduce` function adds the rank contributions for each vertex to compute that vertex's rank, and emits the vertex's existing adjacency list and new rank.

We evaluate PageRank with three different kinds of graphs. The first (PageRank-U) is a 25M vertex synthetically-generated graph where each vertex has an edge to every other vertex with a small, constant probability. Each vertex has an expected degree of 5,000. The second (PageRank-PL) is a 250M vertex synthetically-generated graph where vertex in-degree follows a power law distribution with values between 100 and 10,000. This simulates a more realistic page graph where a relatively small number of pages are linked to frequently. The third (PageRank-WEX) is a graph derived from page links in the XML portion of the WEX data set; it is approximately 1.5GB uncompressed and has 5.3M vertices.

CloudBurst: CloudBurst [21] is a MapReduce implementation of the RMAP [35] algorithm for short-read gene alignment, which aligns a large collection of small "query" DNA sequences called *reads* with a known "reference" genome. CloudBurst performs this alignment using a standard technique called *seed-and-extend.* Both query and reference sequences are passed to the `map` function and emitted as a series of fixed-size *seeds.* The `map` function emits seeds as sequence of `<seed, seed metadata>` pairs, where the seed metadata contains information such as the seed's location in its parent sequence, whether that parent sequence was a query or a reference, and the characters in the sequence immediately before and after the seed.

CloudBurst's `reduce` function examines pairs of query and reference strings with the same seed. For each pair, it computes a similarity score of the DNA characters on either side of the seed using the Landau-Vishkin algorithm for approximate string matching. The `reduce` function emits all query/reference pairs with a similarity score above a configured threshold.

We evaluate CloudBurst on the lakewash_combined_v2 data set from University of Washington [15], which we preprocess using a slightly modified version of the CloudBurst input loader used in Hadoop.

Click Log Analysis: Another popular MapReduce job is analysis of click logs. Abstractly, click logs can be viewed as a collection of `<user ID, timestamp|URL>` pairs indicating which page a user loaded at which time. We chose to evaluate one particular type of log analysis task, *session tracking.* In this task, we seek to identify disjoint ranges of timestamps at least some number of seconds apart. For each such range of timestamps, we output `<user ID, start timestamp|end timestamp|start URL|end URL>` pairs.

The `map` function is a pass-through; it simply groups records by user ID. The `reduce` function does a linear scan through records for a given user ID and reconstructs sessions. For efficiency, it assumes that these records are sorted in ascending order by timestamp. We describe the implications of this assumption in the next section.

7.2 Job Implementation Details

In this section, we briefly describe some of the implementation details necessary for running our collection of example jobs at maximum efficiency.

Combiners: A common technique for improving the performance of MapReduce jobs is employing a *combiner.* For example, word count can emit a single `<word, `k`>` pair instead of k `<word, 1>` pairs. Themis supports the use of combiner functions. We opted to implement combiners within the mapper stage on a job-by-job basis rather than adding an additional stage. Despite what conventional wisdom would suggest, we found that combiners actually decreased our performance in many cases because the computational overhead of manipulating large data structures was enough to make the mapper compute-bound. The large size of these data structures is partially due to our decision to run the combiner over an entire job's intermediate data rather than a small portion thereof to maximize its effectiveness.

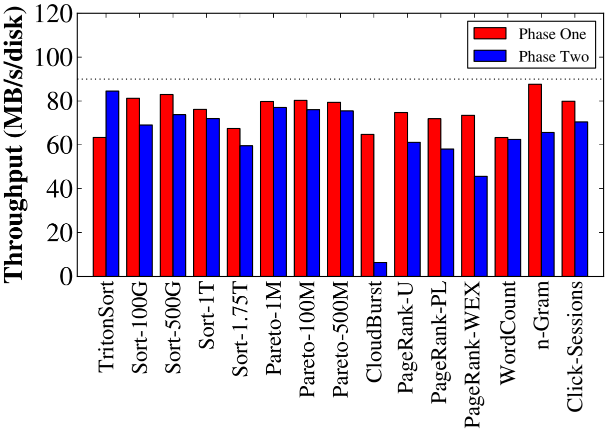


Figure 7: Performance of evaluated MapReduce jobs. Maximum sequential disk throughput of approximately 90 MB/s is shown as a dotted line. Our TritonSort record from 2011 is shown on the left for comparison.

In some cases, however, a small data structure that takes advantage of the semantics of the data provides a significant performance increase. For example, our word count MapReduce job uses a combiner that maintains a counter for the top 25 words in the English language. The combiner updates the appropriate counter whenever it encounters one of these words rather than creating an intermediate record for it. At the end of phase one, intermediate records are created for each of these popular words based on the counter values.

Improving Performance for Small Records: The `map` functions in our first implementations of word count and n-gram count emitted `<word/n-gram, 1>` pairs. Our implementations of these `map` functions emit `<hash(word), 1|word>` pairs instead because the resulting intermediate partitions are easier to sort quickly because the keys are all small and the same size.

Secondary Keys: A naïve implementation of the session extraction job sorts records for a given user ID by timestamp in the `reduce` function. We avoid performing two sorts by allowing the Sorter stage to use the first few bytes of the value, called a *secondary key*, to break ties when sorting. For example, in the session extraction job the secondary key is the record's timestamp.

7.3 Performance

We evaluate the performance of Themis in two ways. First, we compare performance of the benchmark applications to the cluster's hardware limits. Second, we compare the performance of Themis to that of Hadoop on two benchmark applications.

7.3.1 Performance Relative to Disk Speeds

The performance of Themis on the benchmark MapReduce jobs is shown in Figure 7. Performance is measured in

Application	Running Time		Improvement
	Hadoop	Themis	
Sort-500G	28881s	1789s	16.14x
CloudBurst	2878s	944s	3.05x

Table 6: Performance comparison of Hadoop and Themis.

terms of *MB/s/disk* in order to provide a relative comparison to the hardware limitations of the cluster. The 7200 RPM drives in the cluster are capable of approximately 90 MB/s/disk of sequential write bandwidth, which is shown as a dotted line in the figure. A job running at 90 MB/s/disk is processing data as fast as it can be written to the disks.

Most of the benchmark applications run at near maximum speed in both phases. CloudBurst's poor performance in phase two is due to the computationally intensive nature of its `reduce` function, which is unable to process records fast enough to saturate the disks. More CPU cores are needed to drive computationally intensive applications such as CloudBurst at maximum speed in both phases. Notice however that CloudBurst is still able to take advantage of our architecture in phase one.

We have included TritonSort's performance on the Indy 100TB sort benchmark for reference. TritonSort's 2011 Indy variant runs a much simpler code base than Themis. We highlight the fact that Themis's additional complexity and flexibility does not impact its ability to perform well on a variety of workloads. Our improved performance in phase one relative to TritonSort at scale is due to a variety of internal improvements and optimizations made to the codebase in the intervening period, as well as the improved memory utilization provided by moving from buffer pools to dynamic memory management. Performance degradation in phase two relative to TritonSort is mainly due to additional CPU and memory pressure introduced by the Reducer stage.

7.3.2 Comparison with Hadoop

We evaluate Hadoop version 1.0.3 on the Sort-500G and CloudBurst applications. We started with a configuration based on the configuration used by Yahoo! for their 2009 Hadoop sort record [36]. We optimized Hadoop as best we could, but found it difficult to get it to run many large parallel transfers without having our nodes blacklisted for running out of memory.

The total running times for both Hadoop and Themis are given in Table 6. I/O-bound jobs such as sort are able to take full advantage of our architecture, which explains why Themis is more than a factor of 16 faster. As explained above, CloudBurst is fundamentally compute-bound, but the performance benefits of the 2-IO property allow the Themis implementation of CloudBurst to outperform the Hadoop implementation by a factor of 3.

7.4 Memory Management

In this section, we evaluate the performance of our different memory allocation policies. We also show that our allocation system is robust in the face of transient changes in individual stage throughputs.

7.4.1 Memory Allocator Performance

We examine both the individual allocation times of our different memory allocation policies and their end-to-end performance. We evaluate the performance on phase one of a 200GB, 1-node sort job. Table 7 shows that phase one's throughput is essentially unaffected by the choice of allocator policy in this particular instance. These performance numbers can be explained by looking at the mean allocation time for each worker in the system. Figure 8 shows that while the constraint-based allocator is more than twice as slow as the quota-based allocator, the absolute allocation times are both measured in tens of microseconds, which is negligible compared to time taken to actually do useful work.

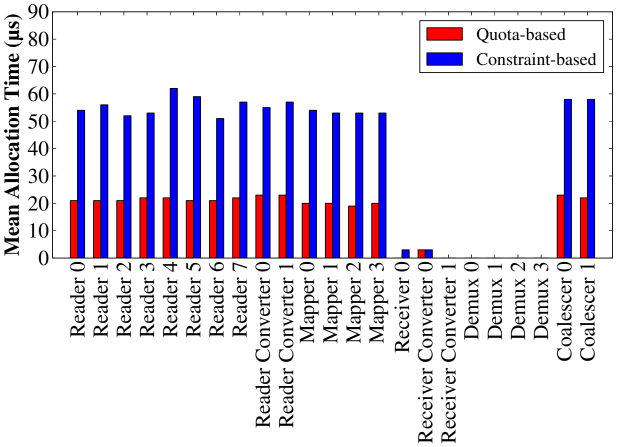


Figure 8: Effects of allocation policy on mean allocation times across workers

Allocation Policy	Phase One Throughput
Constraint-Based	84.90 MBps/disk
Quota-Based	83.11 MBps/disk

Table 7: Performance of allocation policies

However, the results above only hold in the case where the constraint-based allocator does not deadlock. While we never experienced deadlock in phase two, we found it was quite easy to construct situations in which phase one deadlocked. For example, the exact same experiment conducted on a slightly larger data set causes deadlock in phase one with the constraint-based allocator.

The performance results in Figure 7 demonstrate the constraint-based allocation policy performs well in phase two. Because phase two handles entire intermediate partitions in memory, its allocations are orders of magnitude larger than those in phase one. This dramatically increases the likelihood that a single memory request is larger than one of the phase's quotas.

7.4.2 Robustness of the Quota-Based Memory Allocation Policy

We evaluate the robustness of the quota-based memory allocator by artificially slowing down the network for a period of time. We observe the effect on the total quota usage of a stage in the pipeline. Figure 9 shows that the Reader Converter's quota usage spikes up to its limit of 2GB in response to a slow network and then returns back to a steady state of near 0. A slow network means that stages upstream of the network are producing data faster than the network can transmit data. This imbalance leads to data backing up in

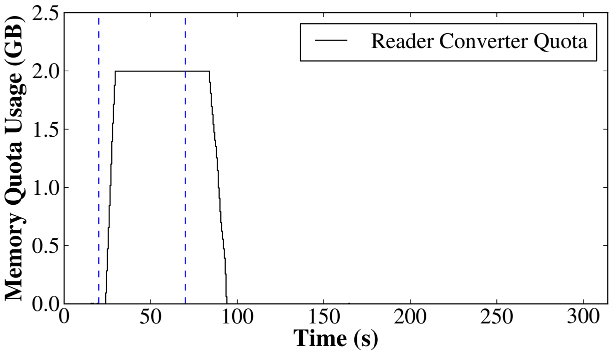


Figure 9: Memory quota usage of the Reader Converter stage. The network was made artificially slow in the time period designated by the dashed lines.

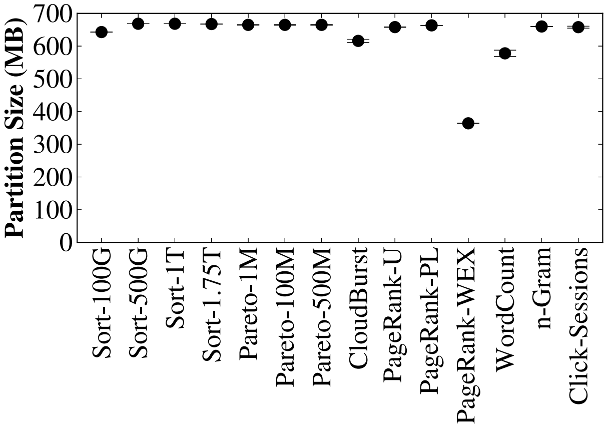


Figure 10: Partition sizes for various Themis jobs. Error bars denoting the 95% confidence intervals are hard to see due to even partitioning.

front of the network. In the absence of the quota allocation policy, this data backlog grows unbounded.

7.5 Skew Mitigation

Next, we evaluate Themis's ability to handle skew by observing the sizes of the intermediate data partitions created in phase one. Figure 10 shows the partition sizes produced by Themis on the evaluated applications. The error bars denoting the 95% confidence intervals are small, indicating that all partitions are nearly equal in size. This is unsurprising for applications with uniform data, such as sort. However, Themis also achieves even partitioning on very skewed data sets, such as Pareto-distributed sort, PageRank, and WordCount. PageRank-WEX has fairly small partitions relative to the other jobs because its intermediate data size is not large enough for phase zero to create an integer number of partitions with the desired size.

7.6 Write Sizes

One of primary goals of phase one is to do large writes to each partition to avoid unnecessary disk seeks. Figure 11 shows the median write sizes of the various jobs we evaluated. For jobs like Sort and n-Gram where the `map` function is extremely simple and mappers can map data as fast as readers can read it, data buffers up in the Chainer stage and all writes are large. As the amount of intermediate data per node grows, the size of a chain that can be buffered for a given partition decreases, which fundamentally limits the size of a write. For example, Sort-1.75T writes data to 2832 partitions, which means that its average chain length is not expected to be longer than about 5 MB given a receiver memory quota of 14GB; note, however, that the mean write size is above this minimum value, indicating that the writer is able to take advantage of temporary burstiness in activity for certain partitions. If the stages before the Writer stage cannot quite saturate it (such as in WordCount, CloudBurst and PageRank), chains remain fairly small. Here the minimum chain size of 4.5 MB ensures that writes are still reasonably large. In the case of PageRank-WEX, the data size is too small to cause the chains to ever become very large.

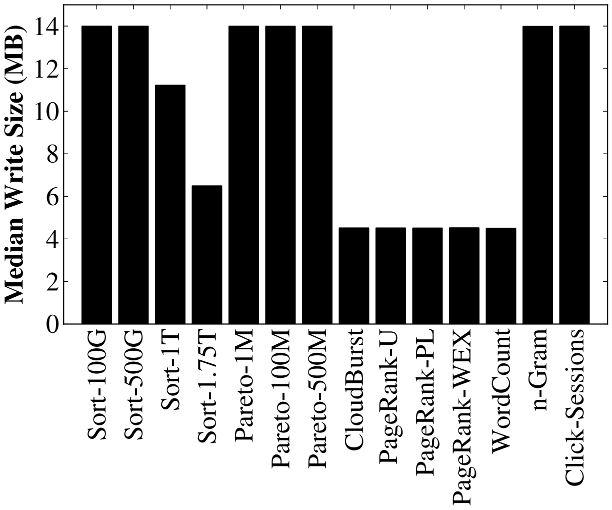


Figure 11: Median write sizes for various Themis jobs

8. RELATED WORK

There is a large continuum of fault tolerance options between task-level restart and job-level restart, including distributed transactions [26], checkpointing and rollback [11], lineage-based recovery [41] and process-pairs replication [34]. Each fault tolerance approach introduces its own overheads and has its own complexities and limitations. With Themis, we choose to focus our efforts on creating a MapReduce system model that is able to handle large real-world data sets while utilizing the resources of an existing cluster as much as possible.

Recovery-Oriented Computing (ROC) [30, 5] is a research vision that focuses on efficient recovery from failure, rather than focusing exclusively on failure avoidance. This is helpful in environments where failure is inevitable, such as data centers. The design of task-level fault tolerance in existing MapReduce implementations shares similar goals with the ROC project.

Sailfish [28] aims to mitigate partitioning skew in MapReduce by choosing the number of reduce tasks and intermediate data partitioning dynamically at runtime. It chooses these values using an index constructed on intermediate data. Sailfish and Themis represent two design points in a space with the similar goal of improving MapReduce's performance through more efficient disk I/O.

Several efforts aim to improve MapReduce's efficiency and performance. Some focus on runtime changes to better handle common patterns like job iteration [4], while others have extended the programming model to handle incremental updates [18, 26]. Work on new MapReduce scheduling disciplines [42] has improved cluster utilization at a map- or reduce-task granularity by minimizing the time that a node waits for work. Tenzing [6], a SQL implementation built atop the MapReduce framework at Google, relaxes or removes the restriction that intermediate data be sorted by key in certain situations to improve performance.

Massively parallel processing (MPP) databases often perform aggregation in memory to eliminate unnecessary I/O if the output of that aggregation does not need to be sorted. Themis could skip an entire read and write pass by pipelining intermediate data through the `reduce` function directly if the `reduce` function was known to be commutative and associative. We chose not to do so to keep Themis's operational model equivalent to the model presented in the original MapReduce paper.

Characterizing input data in both centralized and distributed contexts has been studied extensively in the database systems community [19, 20, 13], but many of the algorithms studied in this context assume that records have a fixed size and are hence hard to adapt to variably-sized, skewed records. Themis's skew mitigation techniques bear strong resemblance to techniques used in MPP shared-nothing database systems [9].

The original MapReduce paper [8] acknowledges the role that imbalance can play on overall performance, which can be affected by data skew. SkewReduce [16] alleviates the computational skew problem by allowing users to specify a customized cost function on input records. Partitioning across nodes relies on this cost function to optimize the distribution of data to tasks. SkewTune [17] proposes a more general framework to handle skew transparently, without requiring hints from users. SkewTune is activated when a slot becomes idle in the cluster, and the task with the greatest estimated remaining time is repartitioned to take advantage of that slot. This reallocates the unprocessed input data through range-partitioning, similar to Themis's phase zero.

9. LIMITATIONS AND FUTURE WORK

Themis's high level of performance is predicated on its ability to tightly control access to its host machine's I/O and memory. As a consequence, it is unclear how Themis would perform when sharing a cluster of machines with other applications. It is possible that some of Themis's features (such as its unified control over disk I/O) might be incorporated into a lower-level service that all processes could share, but we have not explored this approach.

At present, phase one of Themis's execution is limited by the speed of the slowest node, and is thus negatively affected by stragglers. Since Themis does not split its jobs into tasks, it is harder for it to support traditional methods of straggler mitigation such as speculative execution. Investigating alternate means of straggler mitigation is the subject of ongoing work.

Our current implementation of Themis assumes that jobs will execute serially. We realize the limitations of this restriction in real-world MapReduce deployments, and are actively developing a version of Themis that allows multiple MapReduce jobs to execute concurrently while being managed by a single instance of Themis's memory and I/O management subsystems, hence maintaining both high performance and the 2-IO property.

10. CONCLUSIONS

Many MapReduce jobs are I/O-bound, and so minimizing the number of I/O operations is critical to improving their performance. In this work, we present Themis, a MapReduce implementation that meets the 2-IO property, meaning that it issues the minimum number of I/O operations for jobs large enough to exceed memory. To avoid materializing intermediate results, Themis foregoes task-level fault tolerance, relying instead on job-level fault tolerance. Since the 2-IO property prohibits it from spilling records to disk, Themis must manage memory dynamically and adaptively. To ensure that writes to disk are large, Themis adopts a centralized, per-node disk scheduler that batches records produced by different mappers.

There exist a large and growing number of clusters that can process petabyte-scale jobs, yet are small enough to experience a qualitatively lower failure rate than warehouse-scale clusters. We argue that these deployments are ideal candidates to adopt more efficient implementations of MapReduce, which result in higher overall performance than more pessimistic implementations. Themis has been able to implement a wide variety of MapReduce jobs at nearly the sequential speed of the underlying storage layer, and is on par with TritonSort's record sorting performance.

11. ACKNOWLEDGMENTS

The authors wish to thank Kenneth Yocum for his valuable input, as well as Mehul Shah and Chris Nyberg for their input on Themis's approach to sampling. This work was sponsored in part by NSF Grants CSR-1116079 and MRI CNS-0923523, as well as through donations by Cisco Systems and a NetApp Faculty Fellowship.

12. REFERENCES

[1] A. Aggarwal and J. Vitter. The Input/Output Complexity of Sorting and Related Problems. *CACM*, 31(9), Sept. 1988.

[2] E. Anderson and J. Tucek. Efficiency Matters! In *HotStorage*, 2009.

[3] E. Bauer, X. Zhang, and D. Kimber. *Practical System Reliability (pg. 226)*. Wiley-IEEE Press, 2009.

[4] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: Efficient Iterative Data Processing on Large Clusters. In *VLDB*, 2010.

[5] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot – A Technique for Cheap Recovery. In *OSDI*, 2004.

[6] B. Chattopadhyay, L. Lin, W. Liu, S. Mittal, P. Aragonda, V. Lychagina, Y. Kwon, and M. Wong. Tenzing: A SQL Implementation On The MapReduce Framework. In *Proc. VLDB Endowment*, 2011.

[7] Dell and Cloudera Hadoop Platform. `http://www.cloudera.com/company/press-center/`

releases/dell-and-cloudera-collaborate-to-enable-large-scale-data-analysis-and-modeling-through-open-source/.

[8] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, 2004.

[9] D. DeWitt and J. Gray. Parallel Database Systems: The Future of High Performance Database Systems. *CACM*, 35(6), June 1992.

[10] D. DeWitt, J. Naughton, and D. Schneider. Parallel Sorting on a Shared-Nothing Architecture Using Probabilistic Splitting. In *PDIS*, 1991.

[11] E. N. M. Elnozahy, L. Alvisi, Y. Wang, and D. B. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *ACM CSUR*, 34(3), Sept. 2002.

[12] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in Globally Distributed Storage Systems. In *OSDI*, 2010.

[13] M. Hadjieleftheriou, J. Byers, and G. Kollios. Robust Sketching and Aggregation of Distributed Data Streams. Technical Report 2005-011, Boston University, 2005.

[14] Hadoop PoweredBy Index. http://wiki.apache.org/hadoop/PoweredBy.

[15] B. Howe. lakewash_combined_v2.genes.nucleotide. https://dada.cs.washington.edu/research/projects/db-data-L1_bu/escience_datasets/seq_alignment/.

[16] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skew-Resistant Parallel Processing of Feature-Extracting Scientific User-Defined Functions. In *SoCC*, 2010.

[17] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. SkewTune: Mitigating Skew in MapReduce Applications. In *SIGMOD*, 2012.

[18] D. Logothetis, C. Olston, B. Reed, K. C. Webb, and K. Yocum. Stateful Bulk Processing for Incremental Analytics. In *SoCC*, 2010.

[19] G. S. Manku, S. Rajagopalan, and B. G. Lindsay. Random Sampling Techniques for Space Efficient Online Computation of Order Statistics of Large Datasets. In *SIGMOD*, 1999.

[20] J. P. McDermott, G. J. Babu, J. C. Liechty, and D. K. Lin. Data Skeletons: Simultaneous Estimation of Multiple Quantiles for Massive Streaming Datasets with Applications to Density Estimation. *Statistics and Computing*, 17(4), Dec. 2007.

[21] Michael C Schatz. CloudBurst: Highly Sensitive Read Mapping with MapReduce. *Bioinformatics*, 25(11):1363–9, 2009.

[22] J. C. Mogul and K. K. Ramakrishnan. Eliminating Receive Livelock in an Interrupt-Driven Kernel. *ACM TOCS*, 15(3), Aug. 1997.

[23] C. Monash. Petabyte-Scale Hadoop Clusters (Dozens of Them). http://www.dbms2.com/2011/07/06/petabyte-hadoop-clusters/.

[24] W. A. Najjar, E. A. Lee, and G. R. Gao. Advances in the Dataflow Computational Model. *Parallel Computing*, 25(13):1907 – 1929, 1999.

[25] S. Nath, H. Yu, P. B. Gibbons, and S. Seshan. Subtleties in Tolerating Correlated Failures in Wide-Area Storage Systems. In *NSDI*, 2006.

[26] D. Peng and F. Dabek. Large-Scale Incremental Processing Using Distributed Transactions and Notifications. In *OSDI*, 2010.

[27] E. Pinheiro, W. Weber, and L. A. Barroso. Failure Trends in a Large Disk Drive Population. In *FAST*, 2007.

[28] S. Rao, R. Ramakrishnan, A. Silberstein, M. Ovsiannikov, and D. Reeves. Sailfish: A framework for large scale data processing. Technical Report YL-2012-002, Yahoo! Research, 2012.

[29] A. Rasmussen, G. Porter, M. Conley, H. V. Madhyastha, R. N. Mysore, A. Pucher, and A. Vahdat. TritonSort: A Balanced Large-Scale Sorting System. In *NSDI*, 2011.

[30] Recovery-Oriented Computing. http://roc.cs.berkeley.edu/.

[31] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. Fail-Stutter Fault Tolerance. In *HotOS*, 2001.

[32] B. Schroeder and G. Gibson. A Large-Scale Study of Failures in High-Performance Computing Systems. In *DSN*, 2006.

[33] B. Schroeder and G. A. Gibson. Understanding Disk Failure Rates: What Does an MTTF of 1,000,000 Hours Mean to You? *ACM TOS*, 3(3), Oct. 2007.

[34] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An Adaptive Partitioning Operator for Continuous Query Systems. In *ICDE*, 2003.

[35] A. D. Smith and W. Chung. The RMAP Software for Short-Read Mapping. http://rulai.cshl.edu/rmap/.

[36] Sort Benchmark. http://sortbenchmark.org/.

[37] J. S. Vitter. Random Sampling with a Reservoir. *ACM TOMS*, 11(1), Mar. 1985.

[38] Freebase Wikipedia Extraction (WEX). http://wiki.freebase.com/wiki/WEX.

[39] Apache Hadoop. http://hadoop.apache.org/.

[40] Scaling Hadoop to 4000 Nodes at Yahoo! http://developer.yahoo.net/blogs/hadoop/2008/09/scaling_hadoop_to_4000_nodes_a.html.

[41] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*, 2012.

[42] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *OSDI*, 2008.

APPENDIX

A. EXPECTED RUNNING TIME OF A FAILED JOB

Let T be the job's duration and $MTTF$ be the mean time to failure of the cluster. In our model, failure occurs as a Poisson process. We compute the expected running time of a failed job (denoted T_F) as follows:

$$\begin{aligned} T_F &= \int_0^T t \cdot \frac{1}{MTTF} e^{-\frac{t}{MTTF}} dt \\ &= \left[-te^{-\frac{t}{MTTF}} - MTTF \cdot e^{-\frac{t}{MTTF}} \right]_{t=0}^{t=T} \\ &= MTTF - (T + MTTF)e^{-\frac{T}{MTTF}} \qquad (3) \end{aligned}$$

Therefore, if the job duration T is much larger than the MTTF of the cluster ($T \gg MTTF$), Equation 3 implies that $T_F \approx MTTF$, and we expect the job to fail. On the other hand, if $T \ll MTTF$, Equation 3 implies that $T_F \approx T$, and we expect the job to succeed.

B. EXPECTED COMPLETION TIME OF A THEMIS JOB

Let p denote the probability of failure in a single Themis job. Let T denote the running time of the job when there are no failures.

Consider a situation in which the job fails during the first $(n-1)$ trials and completes in the n^{th} trial. The probability of this event is $p^{n-1}(1-p)$. Note that a successful trial takes time T and a failed trial takes time T_F as in Appendix A. To simplify our notation, let $\alpha = T_F/T$ be the fraction of its successful runtime the failed job spent running. Then the total running time in this case is

$$(n-1)\alpha T + T = ((n-1)\alpha + 1)T.$$

By considering such an event for all possible values of n, we get the expected running time to completion for the job:

$$\begin{aligned} S(p,T) &= \sum_{n=1}^{\infty} ((n-1)\alpha + 1)T \cdot p^{n-1}(1-p) \\ &= T(1-p) \sum_{n=1}^{\infty} \left(\alpha n p^{n-1} + (1-\alpha) p^{n-1} \right) \\ &= T(1-p) \left(\alpha \sum_{n=1}^{\infty} n p^{n-1} + (1-\alpha) \sum_{n=1}^{\infty} p^{n-1} \right) \\ &= T(1-p) \left(\alpha \frac{1}{(1-p)^2} + (1-\alpha) \frac{1}{1-p} \right) \\ &= T \left(\alpha \frac{p}{1-p} + 1 \right) \qquad (4) \end{aligned}$$