C-MR: A Continuous MapReduce Processing Model for Low-Latency Stream Proceesing on Multi-Core Architectures

Nathan Backman, Karthik Pattabiraman, Ugur Cetintemel

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-10-01
February 2010

C-MR: A Continuous MapReduce Processing Model for Low-Latency Stream Processing on Multi-Core Architectures

Nathan Backman
Brown University
backman@cs.brown.edu

Karthik Pattabiraman
Brown University
karthikp@cs.brown.edu

Uğur Çetintemel
Brown University
ugur@cs.brown.edu

ABSTRACT

MapReduce has seen a large degree of success due to its simple yet expressive model that exposes data parallelism opportunities. While it has been widely adopted, MapReduce is not suited for stream-oriented, continuous applications due to a reliance on intermediate disk buffers, the absence of pipelined parallelism, and the inability to focus on minimizing the end-to-end latency of complex workflows of MapReduce jobs.

We address this limitation with the design of Continuous-MapReduce (C-MR) which allows the use of the MapReduce programming model with windowing constructs to support continuous execution on data streams.

While it is possible to execute MapReduce jobs continuously within the standard stream processing model, this solution is inflexible and suffers from inefficient computing resource utilization. We therefore adopt and adapt the MapReduce processing model to support streams to retain its benefits.

C-MR allows us to merge the benefits of both worlds by uniting generic computing nodes and windowing constructs. This allows for pipelined data consumption, incremental processing, and reduction in redundant computations for shared input streams. At the same time, generic computing nodes allow us to perform fine-grained load distribution and improve the scheduling of time-sensitive data to reduce end-to-end latencies. C-MR was developed for use on a multi-core system where we demonstrate its effectiveness at supporting high-performance stream processing.

Keywords

stream processing, multi-core, MapReduce

1. INTRODUCTION

Streaming workloads are often dynamic in processing requirements due to varying data complexity, skew, volatile input rates, and bursty streams. The optimization goal of time-critical, streaming applications is to minimize end-to-end latency which requires efficient use of parallel computing resources to achieve a delicate mix of data and task parallelism.

MapReduce[6] is able to achieve a high degree of data parallelism because it breaks workloads down into tasks that can be processed independently of each other on any generic computing node. However, the MapReduce processing model is batch-oriented and is unable to process unbounded data streams. To accommodate continuous processing, it becomes necessary to subdivide the data stream into windows, support pipelined parallelism between operations, and create temporally aware scheduling policies which have a goal of minimizing end-to-end latency.

Only recently, with the Hadoop Online Prototype[5], has there been an attempt to augment an implementation of MapReduce to support pipelined parallelism. However, their reliance on maintaining compliance with Hadoop, its disk-based intermediate buffers, and task-oriented scheduling policies impedes their ability to more efficiently support streaming applications.

Existing stream processing engines facilitate data, task, and pipelined parallelism but in a very coarse-grained manner that results in inefficient computing node utilization and higher end-to-end latencies. Workflow operators are generally assigned to computing nodes using one of the following policies: 1) defining static operator-to-node assignments which are meant to be resilient to volatility[8, 13]; 2) migrating the entire workload of an operator from one computing node to another[3, 9, 14]; and 3) interposing load-balancing operators in the workflow to migrate portions of an operator's data parallel workload between the computing nodes assigned to that operation[4, 11].

These strategies constrain operators to run on a fixed number of computing nodes and those nodes are generally restricted from contributing to arbitrary operators in the workflow. This inflexibility results in workflow-wide load imbalances which ultimately harm end-to-end latency.

We developed C-MR to leverage the benefits of both processing paradigms, merging the flexibility of MapReduce's generic computing nodes with windowing constructs and related optimizations from stream processing research.

1.1 Contributions

With C-MR, we propose an alternative parallel processing model that offers fine-grained workload balancing across all computing nodes and end-to-end latency minimization for stream processing workloads. We achieve these goals in C-MR through the introduction of:

- **Windowing constructs into Continuous MapReduce** which subdivide the data stream into workloads that are suitable for consumption by map and reduce tasks.

Department of Computer Science, Brown University
Providence, Rhode Island 02912 USA

- **Generic computing nodes** which are capable of processing data anywhere in the workflow waiting to be consumed. This also supports fine-grained load balancing across the workflow, and improved scheduling of time-sensitive data.
- **Pipelined parallelism into Continuous MapReduce** to support continuous execution of map and reduce tasks over data streams.
- **Incremental processing** of workloads to increase computing node utilization by moving the combine phase to produce temporally adjacent sub-window aggregates.
- **Reduced redundant computation and storage** as an optimization for overlapping windows and shared operator input streams.

1.2 Outline

In this paper, we present the architecture and fundamental design decisions of Continuous-MapReduce (C-MR) and demonstrate the effectiveness of C-MR at supporting data streams.

The remainder of the paper is organized as follows. Section 2 presents the Continuous-MapReduce processing model. Section 3 introduces the prototype architecture. Section 4 describes implementation details concerning window evaluation, load balancing, and load scheduling. Section 5 details experimental results. And finally, section 6 contains our conclusion and future work.

2. PROCESSING MODEL

C-MR extends the traditional MapReduce processing model to support continuous execution over unbounded streams of data. To understand C-MR, it is important to first understand the traditional MapReduce processing model.

2.1 Traditional MapReduce Model

Google developed the MapReduce framework to leverage clusters of commodity computers for large-scale, data-parallel processing while abstracting away the difficulties of managing such resources from the application programmers. These application programmers are supplied with a notion of *map* and *reduce* tasks whose functionality they implement to manipulate their data.

The MapReduce processing model begins with a **Map Phase** in which data is read from a distributed file system, partitioned among a set of available computing nodes, and sent to those nodes as a set of `(key, value)` pairs. The map tasks, as implemented by the application programmer, will process each of its input records, independently of the other records, and for each record generate intermediate results of the form `list(key, value)`. This intermediate data is buffered to the local disk while the nodes finishing processing the data associated with their map tasks.

After the map tasks on *all* computing nodes have been completed, the **Reduce Phase** begins in which all intermediate data with the same key is collected at the same node to perform an aggregation specified by the application programmer. As computing nodes are allocated to perform reduce tasks, they will first contact a centralized oracle to be assigned a set of keys to aggregate over. They must then fetch the corresponding data from any nodes that produced those keys as output during the map phase. This data is retrieved and sorted locally after which the data is processed and results of the form `list(value)` are produced for each key.

Optionally, there is a *combine* step which can perform a reduce-like aggregation on intermediate data of the same key for the output of a mapper before it is transfered to a reduce node. This serves to conserve network bandwidth and further distribute the aggregation workload associated with a reduce task.

2.2 Continuous MapReduce Model

To support efficient continuous execution of complex MapReduce workflows over unbounded data streams, we made the following departures from the traditional model.

2.2.1 Elimination of Disk Buffers

Due to the low latency requirements of stream processing applications, it is generally considered taboo to buffer data to disk unless it is absolutely necessary. The latency introduced by reading from and writing to disk is unacceptable. For this reason we maintain intermediate data on shared-memory buffers accessible by all computing nodes. The copying of data is minimized by passing the control of references to data between computing nodes as the data migrates through the logical workflow.

This approach has been taken by the MapReduce implementations Phoenix[10] and Mars[7] for utilizing multi-core architectures and SIMD graphics processors, respectively. However, they can only use these buffers between the map and reduce tasks. They do not support workflows with multiple MapReduce operations and to do so would necessitate invoking multiple copies of these applications and require communication through disk buffers. C-MR allows workflows to be created from many pipelined instances of MapReduce tasks while leveraging shared-memory buffers for communication within the workflow at all times.

2.2.2 Introducing Windows

For C-MR, and any framework which supports aggregation on unbounded streams, it is necessary to introduce the concept of windowing. Windows are subdivisions of a stream which contain a set of temporally contiguous data items. Operations are evaluated over these windows and the results propagated along the workflow. Windows are defined by a *size*, denoting the boundaries of the temporally contiguous set included in the window, and a *slide* which defines the interval at which windows are defined and evaluated.

In C-MR, we support the creation of windows based on two differing notions of time. The first uses wall-clock time to assign timestamps to data as they enter the workflow, and the second applies timestamps derived from the actual application. Whether taken from the application or applied by checking the wall-clock time at entry to the workflow, the data is internally represented in the workflow as `(key, value, timestamp)` to aide in the identification of window boundaries, for periodic evaluation of the stream, and to maintain the sorted order of the stream.

2.2.3 Pipelined Processing

Traditional implementations of MapReduce such as Google's[6], Hadoop's[1], and even those previously mentioned, which are geared towards different architectures [10][7], use a strict

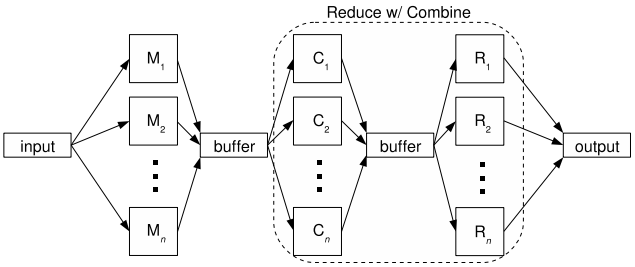

Figure 1: C-MR is able to insert Combine tasks into the workflow to process sub-windows of the windows destined for the reduce tasks. This encourages incremental processing of reduce windows, data parallelism within reduce windows, and mitigates redundant processing of shared sub-window regions across temporally overlapping reduce windows.

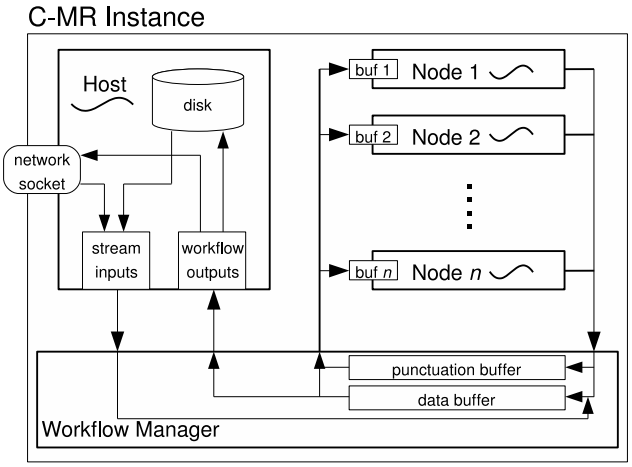

Figure 2: The host invokes a node thread for each local processor/core and then manages the workflow inputs and outputs while the nodes asynchronously execute the arbitrary tasks they are issued by a workload balancer.

two-phase method of performing map and reduce operations. In doing so, they first assign a subset of the computing resources to execute map tasks and, when those tasks have finished, they then assign a second, potentially overlapping, subset of computing nodes to execute the reduce tasks.

Both the batching of data between phases and requiring the strict assignments of computing nodes to tasks inhibit efficient node utilization. The result is coarse-grained, task-based workload allocations and the requirement to delay downstream processing until upstream stragglers have finished the assigned workloads. This is, in part, due to a fundamental focus on throughput instead of latency.

The introduction of windows in C-MR allows us to remove the impermeable barrier between the map and reduce phases. As soon as a window for any particular key is materialized, then it can be processed at *any* downstream computing node regardless of whether any other windows have been materialized. This also enables the concurrent processing of map and reduce tasks.

C-MR utilizes the concept of *generic computing nodes* which are capable of participating anywhere in the workflow at any time. They have the ability to asynchronously extract data from any intermediate buffer, execute the corresponding task, and place the result in a downstream buffer. With a collection of asynchronous map tasks, this is straightforward and trivial to implement. When we consider reduce tasks, however, we are required to fully materialize windows before allowing computing nodes to begin processing them. Thus, computing nodes never have to wait for stragglers to finish a corresponding set of tasks when there is data elsewhere waiting to be processed. They are free to process available data in any of the intermediate workflow buffers.

2.2.4 Incremental Processing

Latency suffers when an entire reduce window must be batched before it can be processed. If we incrementally process data as they arrive, however, we can decrease the time it takes between receiving the last item of a window and producing the results for that window.

With C-MR, we promote incremental processing of reduce windows in data streams by decomposing the windows into sub-windows which we can perform a *Combine* operation on (as seen in Figure 1). While this combine operation itself is identical to the one defined in the original MapReduce paper, it is used quite differently. Instead of applying the combine operation to the output of each individual map task, it is treated, functionally, as another reduce operation in the workflow (between the original map and reduce operations) where windows are buffered, materialized, and processed.

This offers many advantages: 1) the granularity of blocking is decreased as sub-windows will be populated faster than entire reduce windows; 2) multiple sub-windows can be processed concurrently at arbitrary computing nodes in order to further distribute and parallelize the reduce computation; and 3) we can eliminate redundant reduce computations that would occur across multiple temporally adjacent windows by performing a combine operation on shared sub-windows and then sharing the aggregated results to the temporally adjacent windows that would have otherwise repeated those aggregations.

3. ARCHITECTURE

C-MR allows for continuous execution of complex MapReduce workflows on multi-core and symmetric multiprocessor systems. We define each physical core on a computer to be a *computing node* capable of executing any defined map or reduce task.

Our computing nodes have a different task execution strategy than is seen in existing MapReduce implementations. Traditionally, a computing node is scheduled to execute a set of map or reduce tasks and once all nodes processing these like-tasks have finished, those nodes may be re-allocated to other tasks.

C-MR uses an asynchronous, push-based processing model allowing computing nodes to execute any map or reduce workloads as they are received. Computing nodes communicate with one another via shared memory to avoid buffering data to disk and sacrificing latency.

We have outlined the physical architecture of our system in Figure 2.

3.1 Host Process

A computer running an instance of C-MR will launch a `cmrHost` process. This process is responsible for determining the number of available processors/cores on the computer and launching a node thread for each of them. The host will then instantiate a variety of Map and Reduce operators, as specified by the application programmer, which are accessible to nodes to execute tasks with appropriate application code. With the workflow instantiated as a directed acyclic graph of map and reduce operations, the host can proceed to attach input streams to the workflow.

3.2 Workflow Buffer Management

Data destined for reduce operations are collected in an intermediate staging area to reconstruct its input stream into a sorted sequence of windows. Materializing these windows provides us with workloads that can be processed independently of each other, facilitating the use of generic computing nodes to concurrently process arbitrary workflow data.

To maintain the temporal order of intermediate streams in the workflow, it is necessary to sort the data prior to materializing windows. This is necessary because data-parallel execution of tasks on computing nodes can lead to results being received out of order at intermediate workflow buffers.

The Workflow Manager takes on the task of ensuring the inputs to reduce tasks are sorted to ensure that windowed aggregates maintain the correctness of the streaming application. This is handled by the insertion and evaluation of punctuations[12] in the data stream.

Punctuations are inserted into the workflow by the Workflow Manager upstream from each reduce operation. The punctuations are inserted at intervals corresponding to the window slide value and at a location where the stream is already sorted (e.g., at the input to the workflow or just prior to an upstream reduce operation whose input has just been sorted). Punctuations inserted into the stream are replicated to all computing nodes, then passed through the node processing queues, and delivered back to the Workflow Manager. The Workflow Manager will wait for punctuations, related to a given window, to be received from all nodes. The receipt of the last punctuation denotes that all data has arrived for the window that is to be reconstructed. The window defined by the punctuation is then materialized and propagated to computing nodes to be processed by downstream operations in the workflow.

3.3 Computing Nodes

The host launches a computing node thread for each core available on the computer. The thread is confined to run on only that core with the use of the Linux `sched_setaffinity` function to prevent the operating system from scheduling computing node threads on the same computing resource. Each node has its own input queue which is asynchronously populated as data moves through the workflow manager and are assigned to the available nodes. Nodes regularly extract data from their input queues, execute the corresponding operation on the data, and forward the result downstream.

3.3.1 Self-Describing Workloads

Computing nodes are able to extract arbitrary workloads from their input queues and process them by evaluating meta-data associated with each workload. C-MR allows these workloads to be self-describing in that the meta-data defines the type of operation to be applied as well as the downstream operator in the workflow the data is destined for. This enables a computing node to extract an arbitrary workload from its queue, process it with respect to the corresponding map or reduce operation written by the application programmer, and then deliver the resulting workload downstream for further processing.

4. IMPLEMENTATION DETAILS

4.1 Window/Punctuation Management

While windowed data streams have been studied by the stream processing community for years, the application of windows to data parallel stream processing has been less closely examined.

Windows can only be formed on sorted streams, therefore streams delimited by data-parallel operations pose an added complication – data parallel processing does not preserve order. It is therefore necessary to put in place provisions to re-sort the stream between the output of a data-parallel operation and the materialization of windows for operations that perform aggregations over the stream.

4.1.1 Window Formulation & Optimization

Windowing can easily be integrated into traditional batch-oriented implementations of MapReduce which follow a strict 2-phase map-then-reduce policy. Without making any changes to these implementations, and by treating them as black boxes, we can simply construct windows prior to the insertion of data into the systems. In other words, we buffer the sorted input data until a window can be materialized and then send the window to the MapReduce job and shift the input buffer by the slide-amount of the window. While perfectly sound in principle and correctness of results, such a naïve implementation results in large amounts of data replication, redundant processing at both map and reduce operations, and penalizes latency due to a lack of the ability to perform incremental processing on windows.

A more efficient approach, requiring only minor modifications to traditional MapReduce implementations, involves the construction of windows between the map and reduce phases where intermediate results are buffered. This allows all data to be processed by map tasks only once and redundant computations are only observed between shared sub-windows of temporally overlapping windows processed by reduce tasks.

When the degree of overlap between these windows increases in size, the amount of required redundant processing can quickly grow. Consider a continuous MapReduce job which evaluates the last hour of observed data every 5 minutes. In this scenario, a single 5-minute sub-window of data is to be included in 12 temporally adjacent, full-sized windows that will be processed by the reduce task.

In C-MR we leverage the additional Combine phase, when possible, to process these sub-windows only once and share the results of the computation with any full-sized windows that will later require the results of the computation. This aggregation of sub-windows relieves a large portion of the redundant computation mentioned in the previous example. The only redundant computation that still exists after such an optimization is the aggregation of the results of the sub-windows. As this redundant computation is only aggregating aggregates, the redundancy is not substantial.

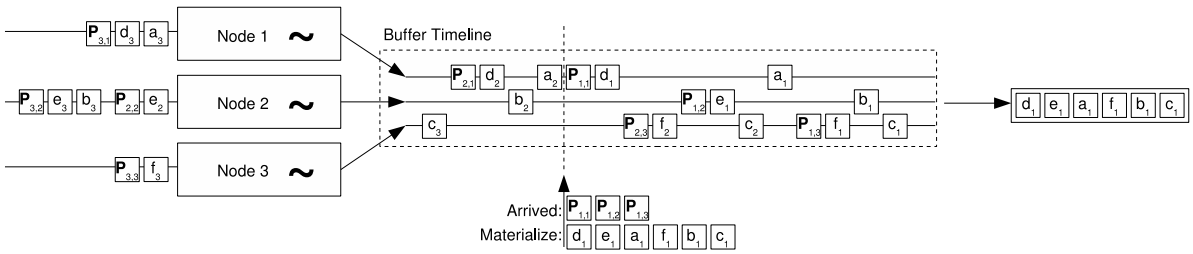

Figure 3: Here we present records a_i through f_i for tumbling windows of timestamps $i \in \{1, 2, 3\}$. Each node's input queue has been populated with 2 records for each window followed by a punctuation for the window. The buffer shows the order of arrival of both records and punctuations as they are produced by the nodes and shows how windows can be materialized once all punctuations for a specific window boundary have been received.

The integration of such a Combine phase in C-MR is represented as an extra reduce phase in the workflow, between map and reduce, with a window size that is a common factor of the full-window's size and slide values and a tumbling window slide value which is equal to the sub-window's size to ensure no overlap between sub-windows.

4.1.2 Window Materialization

As mentioned previously, the inclusion of data-parallel operations in a workflow violates the integrity of a stream's temporal ordering. We cannot be assured in which order parallel computing nodes will return results, thus, we must assume that the intermediate results produced by a data-parallel operation will arrive out-of-order.

To remedy the sort order of our streams, we use punctuations. Punctuations are inserted into sorted streams at some interval relative to an application or system attribute (i.e., application timestamp or system time) to denote the ends of subsets of data within the stream. They are, however, only useful when the stream is sorted. Otherwise, the punctuations may become invalidated if, for example, a punctuation arrives prior to the end of the subset it is meant to terminate. In the case of a data-parallel operation, the merged stream that the computing nodes produce may be unsorted, however, the parallel input/output streams of each of the individual computing nodes does maintain its own order. In C-MR, we exploit this feature and insert window boundary punctuations into each of those parallel streams (one for each active computing node). Punctuations are inserted into the node input streams as if they were any other data item meant to be processed. However, when a node extracts a punctuation from its input queue, it performs no operation but instead simply passes on the punctuation to the downstream buffer.

After collecting all n punctuations at the downstream buffer, where n corresponds to the number of computing nodes performing the parallel processing, we can be assured that no additional data will be received from any node that corresponds to the window boundary defined by the received punctuations. This triggers a materialization of the corresponding window as all its data is available at the buffer. The output of this buffer is a sorted sequence of temporally adjacent windows. The data within each window, however, is not necessarily sorted as this window is merely a relation of temporally adjacent records. An example of punctuation insertion, nodes passing punctuations, and punctuation evaluation for window materialization is presented in Figure 3.

Notice that punctuations are only necessary for aggregate operations that require windowing (reduce and combine), and that punctuations can only be inserted into streams that have already been sorted. To support the materialization of windows via punctuations at arbitrary points in the workflow, we require the input stream to be sorted by some timestamp, whether application timestamp or system timestamp. Doing so allows us to punctuate the sorted input stream for downstream windowing operations. Likewise as those punctuations are evaluated downstream to produce another stream of sorted data, we can punctuate this newly sorted stream for any additional downstream windowing operations.

4.1.3 Window Types

In our C-MR prototype, we currently support only windows whose size and slide are defined by time. Time-based windows enable data associated with all keys to be evaluated with the same periodicity and therefore may make use of the same punctuations. We allow time to be defined in one of two ways. Either records are issued a system timestamp upon entry to the workflow, in which the input is considered as sorted by arrival time, or records may use a timestamp relative to the stream processing application such that the application programmer provides records of the type (key, value, timestamp). Our only constraint on this method is that the input data must be sorted upon its arrival, otherwise it is not possible to know when all data of a particular timestamp has been received without waiting and batching indefinitely.

It is also possible to support count-based windows, whose size and slide are defined by the number of observed records, through punctuations. To do so would require a punctuation per key which involves additional overhead for punctuation management that increases with the range of keys present in the MapReduce job.

4.2 Workload Balancing

In traditional implementations of MapReduce, data processing is batch-oriented and read from disks. They tightly couple workload allocations to the distributed file system. Data are partitioned well before the MapReduce job is executed and the intermediate results are re-partitioned after the map phase and prior to the reduce phase. With Google's

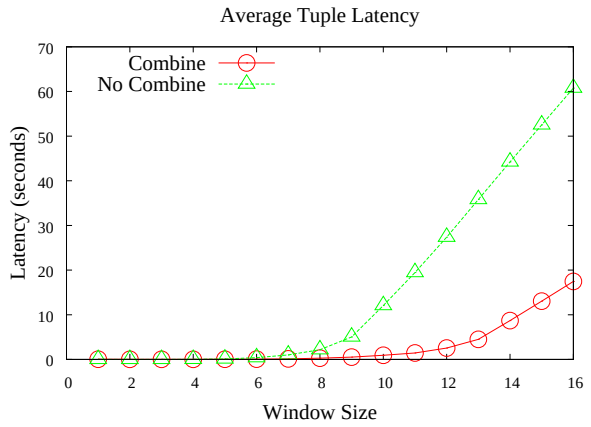

Figure 4: As window overlap increases, No-Combine is forced to do additional processing which increases end-to-end latency whereas Combine is able to eliminate much of the redundant computations.

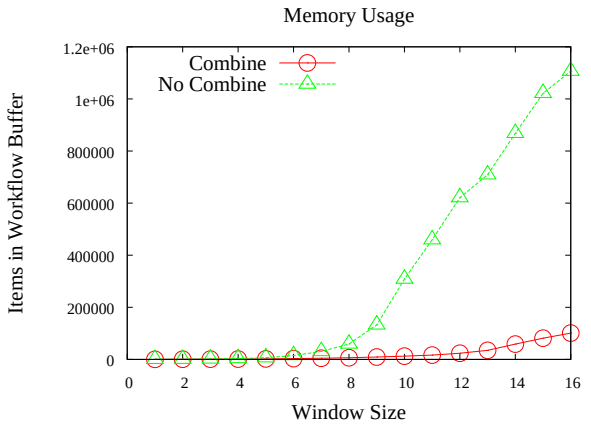

Figure 5: Increased window overlap requires a significantly larger memory footprint for No-Combine while Combine is able to aggregate regions of overlap to conserve memory and processing time.

MapReduce implementation, it is common for MapReduce jobs to process data spread over thousands of disk where the data has already been partitioned by the distributed file system and is ready for consumption by the computing nodes.

In a streaming paradigm it is common for the data stream to originate from a single source such as a TCP socket. This mandates the need for real-time workload assignments for records that are asynchronously received from the input stream. Likewise, between map and reduces phases, or even between MapReduce jobs in a pipeline, it is also necessary to do on-line workload assignments so that intermediate data can have the option of being processed immediately without being batched.

In C-MR we have implemented and compared four different workload assignment strategies.

1. **Randomized assignment**, in which workloads are assigned to random computing nodes.

2. **Round-Robin assignment**, in which workloads are assigned to alternating computing nodes.

3. **Queue-size aware assignment**, in which workloads are assigned to the computing node that currently has the smallest input-queue size.

4. **Weighted Queue-size aware assignment**, in which workloads are assigned to the computing node that has the lowest score computed from the product of input-queue size and external core utilization.

Queue-size assignments allow nodes to accrue data proportionally to the rate at which they consume data. Weighted queue-size uses this same method while also accounting for external contention for computing time (i.e., contention from the `cmrHost` as it manages input queues). Experimental results and analysis for these strategies can be found in section 5.2.

4.3 Workload Scheduling

After workloads have been assigned to computing nodes, they are placed in the appropriate computing node's input queue. The computing node must then make the decision regarding which workload to process first from the queue.

The naïve option is to treat the queue as a first-in-first-out (FIFO) queue and to first process the data that has been in the queue the longest ensures that the average time data waits to be processed at a node is minimized.

Another option is to treat each node's input queue as a priority queue giving precedence to items that have the oldest timestamps. Such a prioritization would encourage computing nodes to process the data that has been in the workflow the longest. We approximate this strategy by giving each computing node multiple FIFO input queues – one for each operator in the workflow. Computing nodes then prioritize their selections by first picking from the queues whose corresponding operator is closest to the output of the workflow.

Experimental results and analysis for these workload scheduling strategies can be found in section 5.3.

5. EXPERIMENTAL RESULTS

We analyzed C-MR on a computer with an Intel®Core^{TM}2 Quad Processor Q6600 (8M Cache, 2.40 GHz, 1066 MHz FSB) with 3GB of RAM. We used the Linux 2.6.26 kernel with Pthreads using the GNU C Library v2.7. The following tests, unless otherwise specified, use a simple moving average stock price application with a single map and reduce operator. The map operator extracts the stock symbol and price from a plain-text string to be used as the key and value respectively. The reduce operator simply takes the average from the set of values associated with a key for a particular window. We used NYSE stock data from the TAQ3 data release of January 2006[2] whereas data streams were reconstructed and replayed at an accelerated rate.

5.1 Combine Effectiveness

The addition of the combine step between map and reduce allows for incremental processing, increased task par-

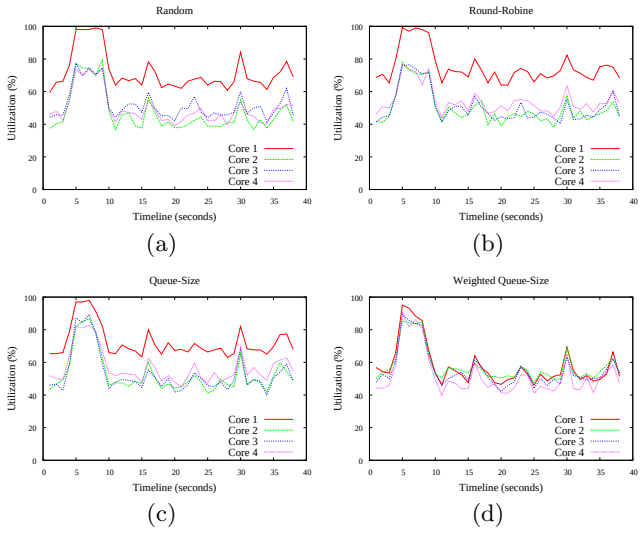

Figure 6: Of the four load distribution strategies used, Weighted Queue-Size is able to best equalize core utilization to help decrease end-to-end latency.

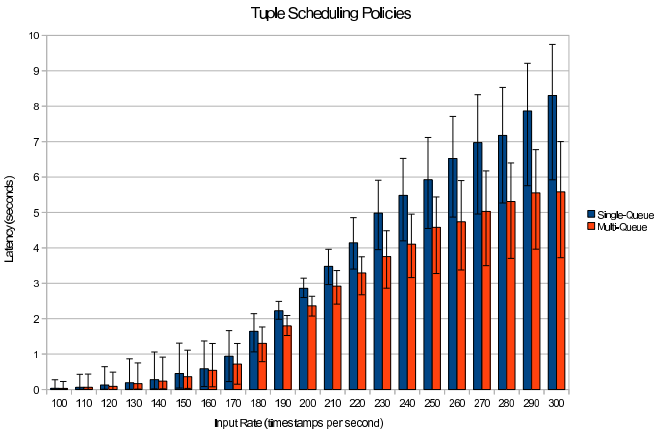

Figure 7: As the stream input rate increases, node input queues are populated more rapidly causing contention for processing time. Multi-Queue is able to prioritize the oldest data, whereas bursts in the input causes Single-Queue to neglect the older data for longer periods of time resulting in higher latencies.

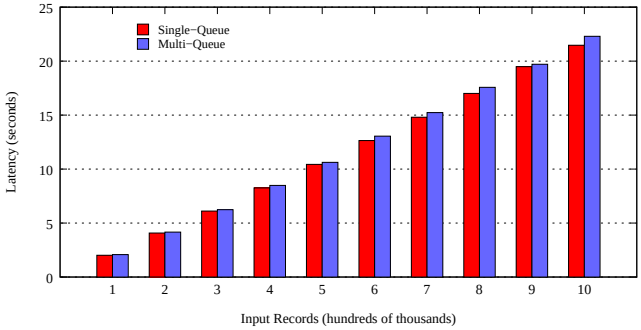

Figure 8: Analyzing the batch processing latency of a workload allows us to see the relative overhead costs incurred by both strategies. Here we see that Multi-Queue has proportionately higher cost for various input sizes.

allelism, and fewer redundant computations. However, it comes with the cost of extending the application workflow and thereby increasing the number of intermediate buffering points at which we must wait for punctuations to materialize windows.

Figure 4 shows the results of the moving average stock price application compared to the same application augmented with a combine phase between the mappers and reducers. The figure depicts the average latency observed from either method as window sizes increase with a fixed window slide (incurring potentially redundant computations).

As can be seen from the figures, the benefits from the addition of a combine step are obvious in the presence of redundant data processing with respect to observed latencies as well as memory usage. As we would imagine, when there is little to no redundant data, the Combine strategy under-performs the No-Combine strategy. This is due to the increased overhead accompanied by the extension of the workflow and introduction of new window materialization points.

5.2 Load Distribution Strategies

To analyze the effectiveness of the load distribution strategies mentioned in the paper, we replayed the stored data stream to instances of C-MR running the mentioned strategies. We again used the moving stock price averaging application but without a combine step, this time, showing only map and reduce workload assignments to computing nodes.

In Figure 6, we depict the timeline associated with the test run of each strategy and the corresponding core utilizations for each of the computing nodes. Of particular note is that, in 3 of the 4 strategies, Core 1 is visibly more burdened than the other cores. This is due to inadequate counterbalancing of work accompanied with the fact that Core 1 not only behaves as a computing node but also manages the input stream. Random and Round-Robin are oblivious to contention for computing resources and are similarly unable to equally distribute computation workloads and equalize core utilization. Queue-size is able to close the gap between the utilization of Core 1 and that of Cores 2, 3, and 4, however, we observed that bursts in the input stream caused the Queue-Size method to fall behind in its processing workload. Weighted Queue-Size was able to perform the best at equalizing work spent among all available cores due to profiling the complexity of input stream management.

5.3 Workload Scheduling

In Figure 7, we compare the Single-Queue and Multi-Queue node scheduling methods outlined in the paper for the moving average stock price application. We see that as input rate of the data increases, the Single-Queue strategy is less graceful in handling the increased load and bursts that are observed when compared to the Multi-Queue strategy. While both process the same exact number of data items, Single-Queue is unable to prioritize the order in which data items are consumed. This has the adverse affect of not push-

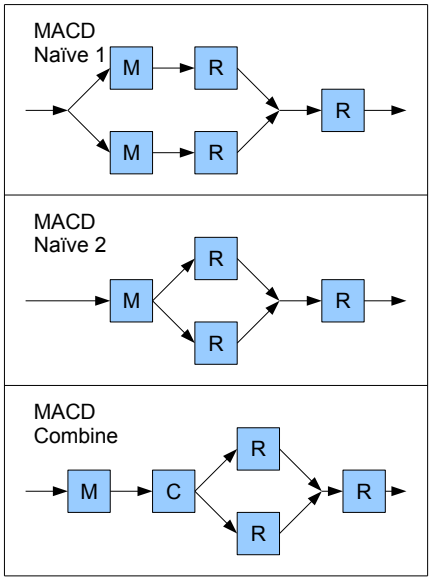

Figure 9: Three levels of workflow optimization which produce identical results with varying end-to-end latencies.

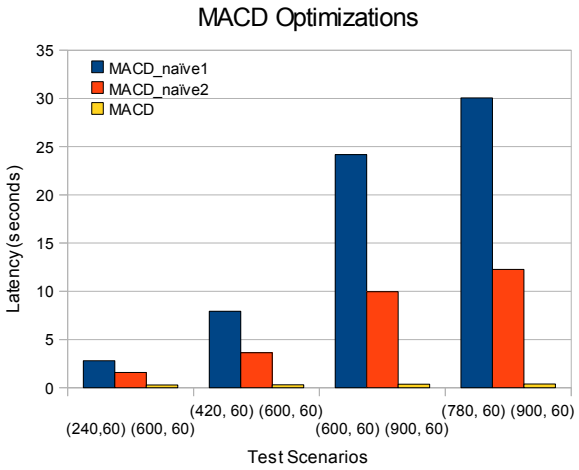

Figure 10: The naive1 strategy provides no optimizations and scales extremely poor as window sizes and window overlap regions increase. Naive2 performs better than naive1 due to the sharing of map computations, however, it still scales poorly. The alternative, MACD combine results in excellent latencies for all tests and scales extremely well due to the savings in redundant computations of both map and aggregation operations.

ing the oldest data items (those that have been in the system longest) through the workflow first to decrease their latency. Instead, bursts in the input cause the Single-Queue method to ignore the older data items for extended periods of time.

We also performed the batch processing of a stored data stream, by loading the entire stream at once, to show the effects of the overhead of Multi-Queue in relation to Single-Queue. In this scenario, since we ignore per-tuple latencies, we observe the latency it takes to process all data such that the order of consumption by a node doesn't matter at all. From these tests, as seen in Figure 8, in which we load incrementally larger data sets, we observe a slight constant overhead of about 3% regardless of the amount of data processed. This small and constant overhead of managing multiple input-queues, however, is not enough to impact the ability of Multi-Queue to provide lower per-tuple latencies than the Single-Queue method.

5.4 MACD Optimizations

To show additional optimizations that C-MR is capable of achieving, we performed another experiment in which we fed the same stock data through a continuously executing moving average convergenece/divergence (MACD) query. This query, common to financial trading applications, performs moving averages of differing window sizes but evaluated at the same interval over the same stream. The differences of the two moving averages is then returned. In Figure 9, we depict three different levels of optimizations for which we can represent the MACD workflow.

C-MR allows us to combine input streams and fork output streams to create complex workflow and to optimize workflows as we see in this case. Additionally, we see in the MACD Combine case that we are able to leverage the ability of combine to aggregate over sub-windows that are common to both moving-average operations in order to prevent a large amount of the computation from occurring twice.

The end-to-end latency results for the three MACD workflows can be seen in Figure 10 in which we compare end-to-end latencies for each of the workflows given various window sizes and overlaps as test scenarios. From the results we can see that the benefit achieved from sharing the map operation is substantial, yet the sharing of the aggregation operation is even more substantial and allows the optimized workflow to scale incredibly well for all tests.

6. CONCLUSIONS AND FUTURE WORK

In summary, we described the Continuous MapReduce processing model that enables the continuous execution of MapReduce jobs on data streams. By incorporating the use of windows and generic computing nodes we were able to leverage pipelined parallelism while achieving fine-grained workload distribution to improve node utilization and make end-to-end latency an optimization goal.

Additionally, we introduced an extra combine phase that sits between the map and reduce steps which allows for incremental evaluation of windows and also provides a means to share the results of overlapping regions between windows to reduce redundant computations and conserve memory.

In the future, we plan to extend the C-MR prototype to leverage clusters of multi-core and multi-processor systems. Such an endeavor will require considering fault-tolerance mechanisms for recovering data that can't be replayed from disks and load distribution algorithms that can handle efficient distribution of many keys to a large number of computing nodes.

7. ACKNOWLEDGEMENTS

This work has been partially supported by the National Science Foundation under grant No. IIS-0916691 and IIS-0448284.

8. REFERENCES

[1] Hadoop MapReduce, http://hadoop.apache.org/mapreduce/.

[2] Monthly TAQ, http://www.nysedata.com/nysedata/, New York Stock Exchange, Inc.

[3] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. B. Zdonik. The design of the borealis stream processing engine. In *CIDR*, pages 277–289, 2005.

[4] R. H. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. E. Culler, J. M. Hellerstein, D. Patterson, and K. Yelick. Cluster I/O with River: Making the fast case common. In *Proceedings of the Sixth Workshop on Input/Output in Parallel and Distributed Systems*, pages 10–22, Atlanta, GA, 1999. ACM Press.

[5] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. Technical Report UCB/EECS-2009-136, EECS Department, University of California, Berkeley, Oct 2009.

[6] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[7] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: a mapreduce framework on graphics processors. In *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 260–269, New York, NY, USA, 2008. ACM.

[8] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *European Conference on Computer Systems (EuroSys)*, pages 59–72, Lisbon, Portugal, March 21-23 2007. also as MSR-TR-2006-140.

[9] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-aware operator placement for stream-processing systems. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*, page 49, Washington, DC, USA, 2006. IEEE Computer Society.

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

[11] M. Shah, J. Hellerstein, S. Chandrasekaran, and M. Franklin. Flux: An adaptive partitioning operator for continuous query systems. Technical Report UCB/CSD-2-1205, U.C. Berkeley, 2002.

[12] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE Trans. on Knowl. and Data Eng.*, 15(3):555–568, 2003.

[13] Y. Xing, J.-H. Hwang, U. Çetintemel, and S. Zdonik. Providing resiliency to load variations in distributed stream processing. In *VLDB '06: Proceedings of the 32nd international conference on Very Large Data Bases*, pages 775–786. VLDB Endowment, 2006.

[14] Y. Xing, S. Zdonik, and J.-H. Hwang. Dynamic load distribution in the borealis stream processor. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering*, pages 791–802. IEEE Computer Society, 2005.