Abstract—The MapReduce programming model simplifies the design and implementation of certain parallel algorithms. Recently, several work-groups have extended MapReduce’s application domain to iterative and on-line data processing. Despite having different data access characteristics, these extensions rely on the same storage facility as the original model, but propagate data updates using additional techniques. In order to benefit from large main memories, fast data access and stronger data consistency, we propose to employ in-memory storage for extended MapReduce. In this paper, we describe the design and implementation of EMR, an in-memory framework for extended MapReduce. To illustrate the usage and performance of our framework, we present measurements of typical MapReduce applications.

Keywords—MapReduce; parallel programming; data storage; scalability; distributed applications; design; experimentation

I. INTRODUCTION

High-performance computing benefits from parallelization as a means to increase application performance. In the recent years, the processing and storage capacity made available by Cloud computing has boosted the interest in large-scale parallel applications. Parallel programming often requires much more effort than sequential programming because of difficulties such as synchronization, data consistency and fault tolerance. Computing models and programming frameworks simplify the development of new parallel applications and help avoid code duplication.

The MapReduce programming model [7] is a popular paradigm for large-scale parallel computations. Since Dean and Ghemawat from Google proposed the model in 2004, many companies and open-source projects have built frameworks and applications which implement the model [6].

The key idea behind the MapReduce model is to map input data into an adequate domain where the problem is solved in parallel, and to reduce the intermediate results to the final solution, again in parallel. The model defines a computation to consist of two phases, map and reduce. In the first phase, a dedicated master node splits input data into map jobs that are mutually independent, thus several worker nodes can execute the jobs in parallel. After the workers have finished the map jobs, the master gathers the intermediate results. In the second phase, the master creates and assigns reduce jobs, which the worker nodes process to transform the intermediate results into the final result. Google’s MapReduce framework stores its data in the proprietary Google File-System (GFS). [7]

The original MapReduce model allows efficient and simple parallelization of many computations with static input data. However, the model does not apply to iterative or online computations. Therefore, extensions of the model have been proposed, such as Twister [9] and the Hadoop online protocol (HOP) [6]. These frameworks do not replace GFS, but require additional facilities for workers to notify their peers of modified input data. For example, Twister promotes data updates using publish-subscribe messaging [9], and HOP sends intermediate data directly from map to reduce workers using RPC [6].

The GFS is well-suited for storing static data which is written once, such as input, intermediate results and output of original MapReduce. It does not provide consistency in case of concurrent random-access write operations, which occur in the extended MapReduce models.

Therefore, we propose to use in-memory storage for extended MapReduce. In-memory storage avoids the latency of disk accesses and can support consistent data updates. Hasso Plattner, the founder of SAP, even says that “main memory is the new disk” [14]. We have designed and implemented EMR, an in-memory framework supporting extended MapReduce.

The EMR framework bases on the ECRAM in-memory storage [8]. ECRAM implements a distributed transactional memory (DTM) [16] for strong data consistency, while also supporting data accesses outside transactions if weaker consistency suffices. Compared to DBMS, which partition data among database servers, ECRAM rather replicates data among all computing nodes. Therefore, EMR does not need to execute distributed transactions, which would require consensus among all nodes, for example using the expensive two-phase commit protocol. By storing all data in RAM, EMR avoids the access latency of hard-disks. Our framework implements job management for extended MapReduce on top of ECRAM, using distributed condition variables to signal availability and completion of jobs.

In summary we make the following contributions: First, we clarify how extensions of the MapReduce model depend on the consistency guaranteed by the underlying data store. Second, we describe the design of our in-memory framework for extended MapReduce applications. Third, we detail the implementation of the EMR framework and selected applications on an in-memory storage that supports transactions.

This paper is structured as follows. In Section 2, we investigate the dependency of MapReduce frameworks on the data
stores underneath them. Section 3 presents the design and implementation of EMR. Section 4 illustrates the implementation of conventional, iterative and online MapReduce algorithms using EMR, and Section 5 evaluates our framework. Finally, Section 6 presents related research, and Section 7 concludes with an outlook on future work.

II. CONSISTENCY AND FAULT TOLERANCE FOR MAPREDUCE COMPUTATIONS

In this section, we analyze the data access patterns in MapReduce and the consistency and fault tolerance requirements of extended MapReduce models.

A. The MapReduce Model

The MapReduce computing model has been suggested by Dean and Ghemawat to simplify parallelization, failure handling and communication of data-intensive computations [7]. MapReduce’s popularity is rooted in two of its properties: the simplicity of parallelizing adequate computations and its applicability to many common data processing problems. The model simplifies parallel programming by relieving the application developer from typical parallel programming tasks such as assigning data to computing nodes, handling node failures and gathering partial results from computing nodes. This simplicity comes at the cost of restricting the programming environment by imposing a fixed algorithmic structure.

In case a problem is not embarrassingly parallel, data dependencies between subproblems impede independent solution, such that the problem cannot be solved using the conventional MapReduce pattern. To widen the applicability of the MapReduce model, several extensions to the original model have been proposed. These extended MapReduce models enable online aggregation of data by making approximate results available early, and they allow continuous queries for online processing of data streams [6]. The extensions also support long-running computations with stateful jobs, requiring stronger data consistency and fault tolerance [9].

B. Data Accesses in MapReduce

MapReduce typically processes vast amounts of data in the magnitude of several terabytes, requiring high availability and resilience from the underlying data store. Moreover, users expect MapReduce applications to yield correct results, such that the data store must provide appropriate guarantees for data consistency. However, a data store cannot guarantee consistency, availability and partition-tolerance in equal measure [11]. Google’s MapReduce implementation is backed by the Google Filesystem, a scalable data store adhering to a relaxed consistency model [10], which we describe in the following subsection.

Accesses to shared data occur in MapReduce in two cases. First, during the framework’s map phase, worker nodes read input data in order to transform it into the intermediate key-value representation. The map phase assumes its input data remains constant. Second, during the reduce phase, worker nodes concurrently write the intermediate data to the respective output files.

Concurrent accesses by MapReduce’s worker nodes necessitate data synchronization. However, MapReduce is free of data dependencies other than the mere presence or absence of data. The Google Filesystem provides atomic rename and append-at-least-once operations that are tailored towards MapReduce’s requirements on consistency, fault tolerance and scalability.

C. Google Filesystem

The highly scalable Google Filesystem (GFS) adheres to a relaxed consistency model: Directory entries are kept strictly consistent by a single master node, enabling atomic file renames. To capture the semantics of concurrent file mutations, Google Filesystem distinguishes between consistent and defined file regions. It guarantees that successful write operations leave the file consistent, but undefined, whereas successful append-at-least-once operations leave the file defined but potentially inconsistent. [10]

The atomic append-at-least-once operation writes a record to all replicas of file, however possibly more often than once on individual replicas. The filesystem considers excess records on individual replicas as inconsistent. This behavior allows for highly parallel write and append operations.

In some use cases, duplicate replicas trigger non-idempotent operations. Applications that depend on stricter consistency requirements need to take special precautions such as application-level checkpointing and writing self-validating records. [10]

Using the atomic append-at-least-once operation helps the MapReduce framework achieve fault tolerance. Failed writes and appends leave file regions inconsistent. Including some task identification into the output record enables the coordinator to detect duplicate records. MapReduce masks worker node failures by detecting missing output records and restarting tasks. [7]

D. Consistency and Fault Tolerance in Extended MapReduce

The extended MapReduce frameworks increase the original MapReduce model in several respects: First, compared to immutable input data and written-once intermediate data in original MapReduce, extended MapReduce may update data. Second, workers do not necessarily run map and reduce phases in lock-step. In contrast, phases can be scheduled as soon as all input data are available.

Like the original MapReduce model, the MapReduce extensions are implemented using message-passing techniques. The computational structure of MapReduce workflow simplifies the interaction of nodes and has low communication overhead. Message-based communication requires explicit coordination with respect to distributing computational tasks and propagating data updates. More complex patterns of interaction, such as with extended MapReduce models, require hand-tailored messaging. Data and execution dependencies increase the complexity of the communication pattern, complicating message-passing based task distribution and data consistency. Therefore, we propose to implement extended MapReduce models using data-oriented communication.
In-memory storage enables data-oriented communication for extended MapReduce. With respect to mutable data, in-memory storage allows for adaptive caching and makes intermediate results accessible to all nodes without requiring explicit exchange of messages. In a dynamic environment, distributed in-memory storage can elastically adapt to workload. To be resilient in cases of memory shortage, network or power outages, in-memory storage systems must provide persistency by means of replication or writing snapshots to permanent storage.

We are convinced that in-memory storage benefits from flexible consistency control. By using transaction semantics, MapReduce applications can synchronize data without requiring the developer to implement locking correctly, which is a major source of lost updates, deadlocks and related programming errors in distributed applications. By bundling operations on data into transactions, an in-memory store avoids excess communication and enables transferring data in bulk.

As we will detail in the following section, a programming framework can implement job management for MapReduce-style computations based on transactional update notifications.

III. THE EMR FRAMEWORK

In this section, we describe the design of EMR, our in-memory framework for extended MapReduce, which includes job management, storage abstractions and synchronization facilities. The framework stores shared input, output and intermediate data in ECRAM. Storage descriptors offers data access in a generic manner, regardless of the different MapReduce flavors. The different execution models require specialized synchronization for either one-pass, lock-stepped or asynchronous execution.

We have designed EMR for deployment in large-scale cloud computing environments, by accounting for node churn, scalable allocation of storage objects and transparent access synchronization.

A. Execution Model

EMR implements an extended MapReduce model. Figure 1 illustrates the execution model of in-memory extended MapReduce in a notation similar to the one used by Dean and Ghemawat [7]. Arrows from nodes to objects denote write accesses, converse arrows represent read accesses.

In contrast to the original MapReduce execution model, the master as well as the workers can create storage objects dynamically. Map and Reduce jobs emerge on demand, such that the master assigns jobs to workers using a job queue. To handle execution dependencies in the same way as data dependencies, EMR reifies computational tasks and stores job descriptors, the job queue and node information blocks in ECRAM.

EMR defines a flexible API consisting of type definitions for storage descriptors, and prototypes for the application-defined hotspot functions to split, map, shuffle and reduce data. Storage descriptors comprise the location of a file, the associated data object (if the file is mapped into shared space), and a condition variable for reader-writer synchronization.

The framework’s main function is called mapreduce. It takes configuration parameters similar to the original MapReduce’s specification object [7], namely the input and output descriptors and the hotspot functions. The application on the master node calls mapreduce to start a MapReduce run. For iterative processing, the master waits until the last reduce job has finished, then starts another iteration to create additional jobs. For online data processing, applications can also create map and reduce jobs by themselves. However, overlapping map and reduce phases depend on the application to manage the synchronization and ordering of jobs.

Worker nodes call the function job_run to wait for jobs by synchronizing on the work queue’s non-empty condition. Whenever a job arrives in the work queue, an idle node awakes, dequeues the job and runs the specified function, passing the corresponding input and output objects. The job_run function loops until a special function called terminate removes the node from the set of available workers and finalizes the application.

B. Framework Architecture

The EMR framework for in-memory extended MapReduce consists of the extended MapReduce run-time environment, the underlying ECRAM in-memory storage and a utility library. Figure 2 depicts our framework’s structure.

The EMR run-time environment includes job management, abstractions for input, output and intermediate storage and synchronization facilities for the different types of MapReduce.

By storing virtually all data in ECRAM, the framework enables MapReduce applications to access data in a location-transparent and fault-tolerant way. Compared to pessimistic synchronization and message passing, ECRAM’s optimistic synchronization scheme and replication layer hide communication latency.

The utility library implements generic data structures such as linked lists and hash tables. The data types are not only used by the runtime environment, they are also available to
application-specific implementations of the hotspot functions. Furthermore, the utility library provides an abstraction for in-memory files.

C. Job Scheduler

Extended MapReduce requires a flexible job scheduling subsystem. Our framework implements scheduling based on ECRAM’s update notifications.

The fundamental concept in our scheduler is the reification of all activities. The scheduler represents all processing nodes and all jobs as in-memory objects. Nodes have work queues attached, and there is a global work queue for jobs which can be scheduled on any processor.

By storing the work queues in ECRAM, the runtime system takes advantage of the wait/notify mechanism to block until a condition becomes true. The function \texttt{ecram\_wait(object\_id, value, comparator)} blocks the calling thread until the specified object satisfies the comparison with the desired value. The comparator may be a test for equality, inequality, greater-than etc. Whenever the node receives an update for the object in question, it checks whether updated value fulfills the condition. In that case, it resumes the previously blocked thread.

Besides storing descriptors of jobs waiting for being processed, each work queue also stores the number of jobs it contains. Using the \texttt{ecram\_wait} function, nodes wait until at least one job is inserted into the work queue. Note that, by virtue of transaction properties, inserting a job into a work queue and incrementing the attached job counter is an atomic operation. Thus, the job scheduler is free of race conditions even without implementing lock management.

The described wait/notify mechanism allows our framework to synchronize execution of different MapReduce flavors. To execute conventional MapReduce jobs, the scheduler on the master node creates an integer object to hold the number of finished map jobs. It splits input data, enqueues the map jobs into the global queue and waits for the integer object to equal the number of jobs submitted. The worker nodes dequeue and execute the jobs, and afterwards each worker transactionally increments the integer object by one. After having shuffled the intermediate results, the master node creates an integer object to numerate the reduce jobs, enqueues the reduce jobs and waits for the completion of all jobs.

Scheduling of iterative MapReduce proceeds similarly. However, the master node creates map and reduce jobs alternately until it has run a specified number of iterations, or a terminating condition is fulfilled. The worker nodes run exactly the same code as with conventional MapReduce.

Online MapReduce requires less job synchronization by the runtime system. The application can create map and reduce jobs as soon as input data is available, or it can create long-running jobs that check for pending data by themselves. However, long-running compute jobs bear internal state. But if internal state is completely stored in ECRAM, the application needs not take special precautions for cases of failures.

D. Optimistic Synchronization

The ECRAM in-memory storage provides optimistic synchronization similar to distributed transactional memory (DTM) [2], [12], [16]. Optimistic synchronization relieves developers from the hassle with distributed locking. Nonetheless, it guarantees strong consistency of all objects accessed during the same transaction.

Implementing parallel algorithms on DTM requires relatively little effort compared to implementation using message-passing. By aggregating data accesses into transactions and resolving access conflicts transparently, DTM simplifies handling data dependencies, which occur frequently in many extended MapReduce algorithms. DTM does not restrict execution flow, such that it supports weakly synchronous and asynchronous parallel execution. On the downside, extended MapReduce still features execution dependencies between jobs, necessitating a job scheduler as detailed above.

The ECRAM transactional in-memory storage has a simple programming interface. The function \texttt{ecram\_bot} starts speculative execution of a transaction. All subsequent accesses to shared objects are recorded in the transaction’s read and write set, until the function \texttt{ecram\_eot} is called to determine the transaction’s validity. If the transaction’s read and write set does not intersect with transactions committed in the meantime, the transaction is said to be valid and commits its changes atomically.

ECRAM allows transparently restarting an invalid transaction by restoring all objects and execution to the initial state at begin of the transaction. Given that ECRAM replicates storage anyways for performance and fault-tolerance reasons, keeping the initial state does not consume any extra space. Algorithm that can be implemented as extended MapReduce applications are well parallelizable, such that the contention even on large amounts of shared data is low, and therefore transaction aborts are infrequent.

Accessing objects outside transactions is possible with ECRAM, however read accesses may return out-of-date content, and changes are not stored permanently. The
E. Scalable Object Allocation

The EMR framework uses ECRAM objects for input, output and intermediate data, for job descriptors, job queues and node info blocks (see Figure 2). Furthermore, applications can allocate distributed objects of arbitrary size and number depending on their storage requirements. ECRAM objects are created dynamically using the functions `ecram_alloc` for malloc-style objects respective `ecram_mmap` for file-based objects. The function `ecram_free` deletes an object.

To avoid global synchronization for object allocation, ECRAM aggregates objects in heaps that belong to exactly one node. Nodes satisfy object allocations from one of their heaps. Freeing an object that has been created by a remote node is allowed. However, entire heaps cannot be freed, but may change their owner.

F. Dynamic Participation

MapReduce usually executes in a large-scale cloud computing environment, where resources are often paid based on their actual usage (pay-as-you-go principle) and failures are frequent. On the one hand, users may wish to add more computing nodes to speed up computation and access results earlier. On the other hand, Amdahl’s law [1] limits the degree of parallelization, such that additional nodes may increase costs with negligible effect on processing speed. Therefore, the cloud platform needs to dynamically configure the participation of computing nodes.

To support extended MapReduce in a cloud-computing environment, EMR supports dynamic configuration depending on available resources and on the workload. We discuss how EMR masks node failures in the next subsection.

Each node is represented by a node information block, which serves to identify the node and contains the node’s private job queue and the node’s current status. Our framework assumes a single master node. Worker nodes organize their node information blocks in a queue. When joining, a worker nodes enqueues its node info block and when leaving, it dequeues the block. To determine how to configure and schedule jobs, the job scheduler on the master node scans the worker queue periodically, for example before each map and reduce phase.

The EMR framework supports nodes joining or leaving during runtime. However, once map and reduce jobs have been scheduled for execution, the set of computing nodes is assumed to remain constant. Newly joined nodes are considered only at the next scheduler invocation. Unexpectedly leaving nodes are handled as failures. To automatically allocate resources depending on imposed workload, EMR must take into account the potential benefit and the overhead of reconfiguration. We plan to include this feature into a future release of EMR.

G. Fault Tolerance

The large-scale computing environment of MapReduce necessitates handling unexpected node failures. In contrast to conventional MapReduce, extended MapReduce can contain stateful compute jobs, such that detecting missing output data does not suffice for fault tolerance [9]. In accordance to original MapReduce, we do not consider failures of the master node.

ECRAM implements fault tolerance using atomic transactions and replicated object. Our framework builds upon ECRAM’s fault tolerance mechanisms. Operations on the job queue and on the worker queue are implemented as transactions. When a job is scheduled to run on a node, our framework assigns it a timestamp. If an ancient timestamp identifies a lagging job, the node will be removed from the worker queue, and the job will be re-scheduled on another node.

IV. IMPLEMENTATION OF DISTRIBUTED ALGORITHMS

Our framework makes implementing distributed extended MapReduce algorithms straightforward. The client nodes simply call `job_run`, passing an array of functions they are able to run as map or reduce jobs. The master node prepares storage descriptors for input and output, for example by mapping input files as shared objects. Then it calls `mapreduce` with the parameters described above. When the last worker sets the output descriptor’s ready condition, the master node can write output objects back to the file system.

The map and reduce jobs can execute transactions to retrieve and modify distributed objects, such that they will always read the most current version, and stores are replicated to guarantee their durability. Note that jobs can alternatively access objects without running transactions, if they contend to access out-of-date data.

A. Word Frequency Analysis

A popular example for MapReduce is word frequency analysis (“word-count”), which determines the frequency of all words contained in a text corpus [7]. The master node distributes the input documents evenly to the worker nodes, which in turn compute intermediate frequency of the words in their subset of documents. In the shuffle phase, the master collects the intermediate histograms and generates an index table. Finally, in the reduce phase, the master directs the workers to sum up the frequency for a specified range of words.

Iterative word frequency analysis makes intermediate results available early. Online computation of word frequencies allows for example tracking word frequencies for a collection of web sites.

We have implemented word-count using our MapReduce framework and in-memory tries (prefix trees). The trie data-structure almost entirely avoids false transaction conflicts, because two insertions of words in a trie collide only if one word is a prefix of the other word. During the map phase, workers build up their own tree, such that collisions are impossible and the map phase can run as a single transaction.
In the reduce phase, each worker scans all trees for a certain subset of prefixes.

B. Real-time Raytracing

Raytracing transforms a 3D scene graph into a 2D image by tracing the path of light through the scene. Most implementations trace each ray separately, making the computation embarrassingly parallel.

Raytracing can improve an image iteratively by replacing interpolated pixels with more exactly calculated pixels, or by increasing the accuracy of effects such as reflection and transparency. Real-time raytracing makes a case for online MapReduce, because nodes can update the image of a changing scene.

In our implementation, the master node splits the 2D image plane into equally-sized partitions. Each node traces the rays in the partition assigned to him during the map phase. The reduce phase simply collates the distinct partitions to a complete image.

C. K-Means Clustering

The heuristic k-means clustering algorithm aims at partitioning points in an Euclidean space into k sets under the constraint that the distance between nodes and cluster centers is minimized. The well-parallelizable algorithm has been implemented as an instance of MapReduce [17] and as an application for transactional memory [4]. Initially, k cluster centers are chosen randomly. Each map job calculates the distance between one point and the k cluster centers. The reduce jobs then determine for each point which cluster center is nearest and, if necessary, re-assign the point to the new center.

Again, iterative k-means clustering allows accessing approximate results fast, with more precise results available later. Running the algorithm online enables asynchronous updates of cluster centers and assignment of points to cluster centers.

V. Evaluation

We have evaluated our framework using the word-frequency and raytracer applications. The experiments were run on a cluster system consisting of 33 computing nodes, each equipped with 2 AMD Opteron processors (16 x Opteron 246 @ 2 GHz, 16 x Opteron 244 @ 1.8 GHz) and 2 GB ccNUMA RAM. The nodes were configured to boot Debian Squeeze with Linux x86-64 kernel 2.6.32 in disk-less mode via NFS.

The experiments ran with one master node and a varying numbers of workers. We kept the problem size (i.e. input text for word-frequency respective output image for raytracer) constant, such that the amount of work per node was inverse to the number of workers. The word-frequency application processed novel Ulysses by James Joyce, which contains about 268000 words. The raytracer rendered a scene consisting of 228 objects on an image with a dimension of 640x480 pixels.

The first experiment resolves the total run-time of one-pass MapReduce runs into the latency of the map and reduce phases and on the overhead imposed by the framework (see Figures 3 and 4).

The run-time analysis of word-frequency shows that the framework’s overhead, which is mainly caused by job configuration and synchronization, is negligible. The map phase, during which each node scans his partition of the input file, scales quite well in the number of nodes. The reduce phase does not scale as well, because all nodes write to the final results concurrently. We have also observed an increase in the transaction conflict rate.

Concerning the raytracer, the reduce phase, during which one node gathers the image data from its peers and stores it locally, is almost constant. As we had expected, the duration of the image computation during the map phase is almost inverse to the number of nodes. This demonstrates that the raytracer scales very well.

We have also measured the number of transactions each worker executes (see Figures 5 and 6). An increase in the number of workers should not result in an higher number of transactions per node, because this would eventually limit the scalability of transaction validation. However, doubling the number of workers approximately halves the input size per node. Our measurements confirm that the number of transactions per worker is approximately inverse to the number of workers. As detailed above, a map job in the wordcount application only runs a single transaction, because conflicts are impossible. The raytracer runs a constant number of transactions for map jobs and one transaction for reduce jobs, such that most transactions are run by the EMR runtime system when managing jobs.

Finally, we have evaluated the consumption of distributed ECRAM storage for all nodes including the master node (see Figure 7). The raytracer application requires a constant amount of storage, independent of the number of workers. The storage required by the wordcount application increases with the number of nodes, because, during the map phase, every node builds up an intermediate data structure whose size depends rather on the variety than on the amount of words in its input split.
VI. RELATED WORK

Our EMR framework implements a MapReduce model similar to Twister and HOP. In contrast to EMR, Twister promotes data updates using multicast and publish/subscribe messaging. To achieve high performance despite the communication overhead of data updates, Twister uses relatively large-grained stateful map tasks. Based on our experience with EMR, we confirm this heuristic. Furthermore, Twister inserts a local combine operation before the global reduce operation to determine whether to continue iteration. Using EMR, applications control the number of iterations to run.

MapReduce Online modifies the original MapReduce model to allow for online aggregation and continuous queries [6]. In addition to storing intermediate data temporarily on disk, the Hadoop Online Prototype (HOP) pipelines data directly between computing nodes, effectively converting MapReduce’s original data-centric model to a message-oriented model. MapReduce Online broadens the field of application of the original MapReduce model and increases the degree of parallelism even for legacy workload. On the downside, MapReduce Online fundamentally changes the MapReduce design. The HOP implementation involves buffer management, RPC-based communication and progress monitoring. On the one hand, our data-centric approach does not need explicit messaging. On the other hand, EMR’s performance and fault tolerance cannot profit from hand-tailored communication.

Our framework has some similarities with Google’s Percolator project [13], which enables incremental processing of large data sets as a replacement for MapReduce. In contrast to our framework, Percolator is integrated in Google’s closed-source infrastructure, entailing Google File System and the tabular data store Bigtable. Percolator’s transactions are scale impressively well, almost linearly for more than five thousand cores. However, transactions cannot embrace more than a single row or column in Bigtable.

To enable modifications to trigger transaction execution, Percolator’s designers have defined a notification mechanism. Notifications are similar in functionality to database triggers.
in that they are executed after a data update, but the observer runs as a separate transaction. Notifications serve a purpose similar to ECRAM’s conditions, which also continue execution after a transaction has updated certain data. While Perlocator’s observers randomly scan the table for pending notifications, our condition mechanism is driven by commit messages.

There exist several MapReduce implementations using data-centric communication, most prominently Phoenix [15], [17] and Ostrich [5]. However, unlike our framework, these implementations target multicore processors with shared memory access. Ostrich optimizes Phoenix with respect to memory reuse, data locality and overlapping map and reduce phases. There is some preliminary information available about online MapReduce on a shared-memory architecture [3]. Unfortunately, the authors do not detail the consistency and reliability guarantees of their implementation.

VII. CONCLUSION

The MapReduce model has been proposed to relieve developers of highly parallel applications from handling data consistency and node failures. The alleviation of these hassles was successful, but the original MapReduce model does not apply to certain computational problems. Extensions of the original MapReduce model necessitate caring for consistency and reliable execution once again. In-memory storage benefits consistency and fault handling, giving rise to the idea of an in-memory framework for extended MapReduce.

We have proposed the EMR framework and described its architecture. After detailing how we have implemented the framework on the ECRAM in-memory storage, we have given several examples of applications using EMR. The performance evaluation confirms the scalability of our in-memory framework.

Future work will focus on improving EMR’s ability to adapt elastically to imposed workload and available resources. We also plan to implement more extended MapReduce algorithms on our framework. The insights gained will be valuable to improve the EMR framework and the ECRAM in-memory storage.

REFERENCES


