Replication-Based Fault-Tolerance for Large-Scale Graph Processing

Rong Chen, *Member, IEEE*, Youyang Yao, Peng Wang, Kaiyuan Zhang, Zhaoguo Wang, Haibing Guan, Binyu Zang, and Haibo Chen, *Senior Member, IEEE*

Abstract—The increasing algorithmic complexity and dataset sizes necessitate the use of networked machines for many graph-parallel algorithms, which also makes fault tolerance a must due to the increasing scale of machines. Unfortunately, existing large-scale graph-parallel systems usually adopt a distributed checkpoint mechanism for fault tolerance, which incurs not only notable performance overhead but also lengthy recovery time. This paper observes that the vertex replicas created for distributed graph computation can be naturally extended for fast in-memory recovery of graph states. This paper describes Imitator, a new fault tolerance mechanism, which supports cheap maintenance of vertex states by replicating them to their replicas during normal message exchanges, and provides fast in-memory reconstruction of failed vertices from replicas in other machines. Imitator has been implemented on Cyclops with edge-cut and PowerLyra with vertex-cut. Evaluation on a 50-node EC-2 like cluster shows that Imitator incurs an average of 1.37 and 2.32 percent performance overhead (ranging from −0.6 to 3.7 percent) for Cyclops and PowerLyra respectively, and can recover from failures of more than one million of vertices with less than 3.4 seconds.

Index Terms—Graph-parallel system, fault-tolerance, replication

1 Introduction

Graph-parallel abstractions have been widely used to express many machine learning and data mining (MLDM) algorithms, such as topic modeling, recommendation, medical diagnosis and natural language processing [1], [2], [3]. With the algorithm complexity and dataset sizes continuously increasing, it is now a common practice to run many MLDM algorithms on a cluster of machines or even in the cloud [4]. For example, Google has used hundreds to thousands of machines to run some MLDM algorithms [5], [6], [7].

Many graph algorithms can be programmed by following the "think as a vertex" philosophy [5], by coding graph computation as a vertex-centric program that processes vertices in parallel and communicates along edges. Typically, many MLDM algorithms are essentially iterative computation that iteratively refines input data until a convergence condition is reached. Such iterative and convergence-oriented computation has driven the development of many graph-parallel systems, including Pregel [5] and its open-source clones [8], [9], GraphLab [10], [11], GraphX [12], and PowerLyra [13].

- *R. Chen, Y. Yao, P. Wang, K. Zhang, H. Guan, B. Zang, and H. Chen are with the Shanghai Key Laboratory for Scalable Computing and Systems, Shanghai Jiao Tong University, Shanghai 200240, China. E-mail: {rongchen, youyangyao, peng-wp, kaiyuanzhang, hbguan, byzang, haibochen}@sjtu.edu.cn.*
- *Z. Wang is with New York University, New York, NY 10003. E-mail: zhaoguo@cs.nyu.edu.*

Manuscript received 4 Aug. 2016; revised 28 Mar. 2017; accepted 6 May 2017. Date of publication 15 May 2017; date of current version 13 June 2018. (Corresponding author: Haibo Chen.)
Recommended for acceptance by C. Carothers.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2017.2703904

Running graph-parallel algorithms on a cluster of machines essentially faces a fundamental problem in distributed systems: fault tolerance. With the increase of problem sizes (and thus execution time) and machine scales, the failure probability of machines would increase as well. Currently, most graph-parallel systems use a checkpoint-based approach. During computation, the runtime system will periodically save the runtime states into a checkpoint on some reliable global storage, e.g., a distributed file system. When some machines crash, the runtime system will reload the previous computational states from the last checkpoint and then restart the computation. Example approaches include synchronous checkpoint (e.g., Pregel and PowerGraph) and asynchronous checkpoint using the Chandy-Lamport algorithm [14] (e.g., Distributed GraphLab [10]). However, as the processes of checkpoint and recovery require saving and reloading from slow persistent storage, such approaches incur notable performance overhead during normal execution as well as lengthy recovery time from a failure. Consequently, though most existing systems have been designed with fault tolerance support, they are disabled during the production run by default [12].

This paper observes that many distributed graph-parallel systems require creating replicas of vertices to provide local access semantics such that graph computation can be programmed as accessing local memory [10], [11], [13], [15], [16]. Such replicas can be easily extended to ensure that there are always at least *K+1* replicas (including master) for a vertex across machines, in order to tolerate *K* machine failures.

Based on this observation, Imitator proposes a new approach that leverages existing vertex replication to tolerate machine failures, by extending existing graph-parallel

systems in three ways. First, Imitator extends existing graph loading phase with fault tolerance support, by explicitly creating additional replicas for vertices without replication. Second, Imitator maintains the freshness of replicas by synchronizing the *full states* of a master vertex to its replicas through extending normal messages. Third, Imitator extends the graph-computation engine with fast detection of machine failures through monitoring vertex states and seamlessly recovers the crashed tasks from replicas on multiple machines in a parallel way, inspired by the RAMCloud approach [17].

Imitator uses a randomized approach to locating replicas for fault tolerance in a distributed and scalable fashion. To balance load, a master vertex selects several candidates at random and then chooses among them using more detailed information, which provides near-optimal results with a small cost. Imitator currently supports two failure recovery approaches. The first approach, which is called Rebirth based recovery, recovers graph states on a new backup machine when a hot-standby machine for fault tolerance is available. The second one, the Migration-based recovery, redistributes graph states of the failed machines to multiple surviving machines.

Further, since different graph partitioning strategies (i.e., edge-cut [18], [19], [20], [21], [22] or vertex-cut [11], [12], [13], [23], [24], [25]) will treat edges in different ways, Imitator adopts differentiated approaches to tolerating the loss of edges. For graph-parallel systems using edge-cut, all edges are extended with full states of the masters and duplicated and synchronized to replicas upon updates. For systems using vertex-cut, all edges are partitioned and saved to persistent storage as multiple files at graph loading; the updates to the edges will be logged into persistent storage by overlapping with graph processing.

We have implemented Imitator on Cyclops [26] and PowerLyra [13], which use edge-cut and vertex-cut accordingly. To demonstrate the effectiveness and efficiency of Imitator, we have conducted a set of experiments using four popular MLDM algorithms on a 50-node EC2-like cluster (200 CPU cores in total). Experiments show that Imitator can recover from one machine failure in around 2 seconds. Performance evaluation shows that Imitator incurs an average of 1.96 percent (ranging from -0.6 to 3.7 percent) performance overhead for all evaluated algorithms and datasets. The memory overhead from additional replicas is also modest.

This paper makes the following contributions:

- A comprehensive analysis of existing checkpoint-based fault tolerance mechanisms for graph-parallel computation model (Section 2).
- A new replication-based fault tolerance approach for graph computation using either edge-cut or vertex-cut (Sections 3, 4, and 5).
- A detailed evaluation that demonstrates the effectiveness and efficiency of Imitator (Section 6).

2 Background and Motivation

This section first briefly introduces checkpoint-based fault tolerance in typical graph-parallel systems. Then, it examines performance issues during both normal execution and recovery.

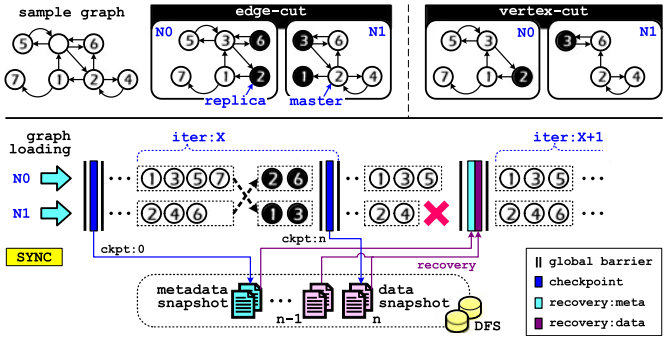


Fig. 1. The sample of checkpoint-based fault tolerance.

2.1 Graph-Parallel Execution

Many existing graph-parallel systems usually provide a shared memory abstraction[1] to a graph program. To achieve this, the input graph is first partitioned into multiple machines using edge-cut or vertex-cut, and then replicated vertices (vertex 1, 2, 3 and 6 under edge-cut, and vertex 2 and 3 under vertex-cut in Fig. 1) are created in machines where there are edges connecting to the original master vertex. To enable such an abstraction, a master vertex synchronizes its states to its replicas either synchronously or asynchronously through messages.

Graph partitioning plays a vital role in reducing communication cost, and highly impacts the design of all components of graph-parallel systems from computation engine to fault tolerance. Broadly, the algorithms are categorized into two groups: edge-cut and vertex-cut. The p-way edge-cut [18], [19], [20], [21], [22] evenly assigns *vertices* of a graph to p machines and replicates edges spanning machines to ensure that the master of each vertex locally connects with all of its edges. The vertices will also be replicated for edges spanning machines. In contrast, the p-way vertex-cut [11], [12], [13], [23], [24], [25] evenly assigns edges to p machines and replicates *vertices* for all edges. The main difference between edge-cut and vertex-cut is whether to replicate edges or not. Fig. 1 illustrates the edge-cut and vertex-cut on the sample graph.

The scheduling of computation on vertices can be synchronous (SYNC) or asynchronous (ASYNC). Fig. 1 illustrates the execution flow of synchronous mode on a sample graph, which is divided into two nodes (i.e., machines). Vertices are evenly assigned to two nodes with ingoing edges, and replicas are created for edges spanning nodes. In the synchronous mode, all vertices are iteratively executed in a fixed order within each iteration. A global barrier between consecutive iterations ensures that all vertex updates in the current iteration are simultaneously visible in the next iteration for all nodes through batched messages. The computation on vertex in the asynchronous mode is scheduled on the fly, and uses the new states of neighboring vertices immediately without a global barrier.

Some graph-parallel systems such as PowerGraph [11], PowerLyra [13], PowerSwitch [27], and GRACE [28], provide both execution modes, but usually use synchronous execution as the default mode. Hence, this paper only considers the synchronous mode. How to extend Imitator to asynchronous execution will be our future work.

1. Note that this is a restricted form of shared memory such that a vertex can only access its neighbors using shared memory abstraction.

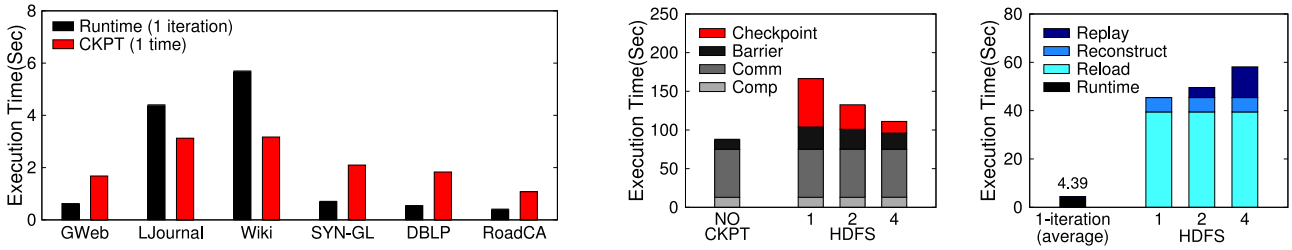


Fig. 2. (a) The performance cost of once checkpointing in Imitator-CKPT for different algorithms and datasets. (b) The breakdown of overall performance overhead of checkpoint-based fault tolerance for PageRank with LJournal [29] using different configuration. (c) The breakdown of recovery time for PageRank with LJournal using different configuration.

2.2 Checkpoint-Based Fault Tolerance

Existing graph-parallel systems implement fault tolerance through *distributed checkpointing* for both synchronous and asynchronous modes. After loading a graph, each node stores an immutable graph topology of its own graph part to a metadata snapshot on the persistent storage. Such information includes adjacent edges and the location of replicas. During execution, each node periodically logs updated data of its own part to the snapshots on the persistent storage, such as new values and states of vertices and edges. For the synchronous mode, all nodes will simultaneously do logging for all vertices in the global barrier to generate a consistent snapshot. While for asynchronous mode, all nodes initiate logging at fixed intervals, and perform a consistent asynchronous snapshot based on the Chandy-Lamport algorithm [14]. The checkpoint frequency can be selected based on the mean time between failures model [30] to balance the checkpoint cost against the expected recovery cost.[2] Upon detecting any machine failures, the graph states will be recovered from the last completed checkpoint. During recovery, all nodes first reload the graph topology from the metadata snapshot in parallel and then update states of vertices and edges through data snapshots. Finally, since the last checkpoint may be earlier than the latest state before the failure, all nodes should also replay all missing operations before resuming execution. Fig. 1 illustrates an example of checkpoint-based fault tolerance for synchronous mode.

2.3 Issues of Checkpoint-Based Approach

Though many graph-parallel systems provide checkpoint-based fault tolerance support, it is disabled by default due to notable overhead during normal execution and lengthy recovery time [12]. To estimate checkpoint and recovery cost, we evaluate the overhead of checkpoint (Imitator-CKPT) based on Apache Hama [9], [31],[3] an open-source clone of Pregel. Note that Imitator-CKPT is several times faster than Hama's default checkpoint mechanism (up to 6.5X for Wiki dataset [32]), as it can *periodically* launch checkpoint to create an *incremental* snapshot and avoid storing *messages* in snapshots due to vertex replication. Further, Imitator-CKPT only records the necessary states according to the behavior of graph algorithms. For example, Imitator-CKPT skips edge data for PageRank. Hence, Imitator-CKPT can be viewed as a near-optimal case of prior checkpoint-based approaches.

In the rest of this section, we will use Imitator-CKPT to illustrate the issues with checkpoint-based fault tolerance on a 50-node EC2-like cluster. The detailed experimental setup can be found in Section 6.1.

2.3.1 Checkpointing

Checkpointing requires time-consuming I/O operations to create snapshots of updated data on a globally visible persistent storage (we use HDFS [33] here). Fig. 2a illustrates the performance cost of one checkpoint for different algorithms and datasets. The average runtime of one iteration without checkpointing is also provided as a reference. The relative performance overhead of checkpointing for LJournal [29] and Wiki [32] is relatively small, since HDFS is more friendly to writing large data. Even for the best case (i.e., Wiki), creating one checkpoint still incurs more than 55 percent overhead.

Fig. 2b illustrates an overall performance comparison between turning on and off checkpointing on Imitator-CKPT for PageRank with the LJournal by 20 iterations. We configure Imitator-CKPT using HDFS to store snapshots and using different intervals from 1 to 4 iterations. Checkpointing snapshots to HDFS is not the only cause of overhead. The imbalance of global barrier also contributes a notable fraction of performance overhead, since the checkpoint operation must execute in the global barrier. In addition, though decreasing the frequency of intervals can reduce overhead, it may result in snapshots much earlier than the latest iteration completed before the failure, and increase the recovery time due to replaying a large amount of missing computation. The overall performance overhead for intervals 1, 2, and 4 iteration(s) reaches 89, 51 and 26 percent accordingly. Hence, such a significant overhead becomes the main reason to the limited usage of checkpoint-based fault tolerance for graph-parallel systems in practice.

2.3.2 Recovery

Though most fault-tolerance mechanisms focus on minimizing overhead in logging, the time for recovery is also an important metric of fault tolerance. The poor performance and scalability in recovery is another issue of checkpoint-based fault tolerance. In checkpoint-based recovery, all nodes, even the surviving nodes, need to reload states in the

2. The mean time between failures (MTBF) of a 50-node cluster is about 7.3 days [10]. According to Youngs model [30], the optimal checkpointing interval is more than 2 hours, which far exceeds the execution time of graph processing. In contrast, the overhead of pessimistic checkpointing and the lengthy recovery time may exceed the cost of just rerunning the bare execution when machine failures.

3. We extended and refined Hama's checkpoint and recovery mechanism as it currently does not support completed recovery.

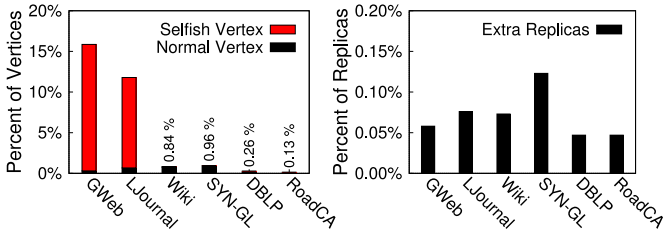


Fig. 3. (a) The percent of vertices without replicas, including normal and selfish vertex. (b) The percent of extra replicas for fault tolerance.

most recent snapshot from the persistent storage or even through network, since all nodes must rollback to a consistent state. Further, because the states of crashed node are stored on persistent storage, surviving nodes can not help to reload the states on the new node (*newbie*), which substitutes the crashed node. Consequently, the time for recovery is mainly limited by the I/O performance of each node. Even worse, typical optimizations in checkpointing, such as incremental snapshot and lower frequency of intervals, may further increase the recovery time.

Fig. 2c illustrates a comparison between the average runtime of one iteration and recovery on Imitator-CKPT for PageRank with the LJournal. The recovery consists of three steps, including *(reload)*ing (meta)data, *(reconstruct)*ing in-memory graph states, and *(replay)*ing the missing computation.[4] The reloading from snapshots on persistent storage incurs the major overhead, since all nodes are busy reloading their own states from persistent storage, not just the states of crashed nodes.

In addition, a *standby* node for recovery may not always be available, especially in a resource-scarce in-house cluster. It is also impractical to wait for rebooting of the crashed node. This constrains the usage scenario of such an approach. Further, as it only enables to migrate the workload on crashed node to a single surviving node, it may result in significant load imbalance and performance degradation of normal execution after recovery.

3 Replication-Based Fault Tolerance

This section first identifies challenges and opportunities in providing efficient fault tolerance, and then describes the design of Imitator.

3.1 Challenges and Opportunities

Low Overhead in Normal Execution. Compared to data-parallel computing models, the dependencies between vertices in graph-parallel models demand a fine-grained fault tolerance mechanism. Low overhead re-execution [34] and coarse-grained transformation [35] can hardly satisfy such a requirement. In contrast, checkpoint-based fault tolerance in existing graph-parallel systems sacrifices the performance of normal execution for fine-grained logging.

Fortunately, existing replicas for vertex computation in a distributed graph-parallel system open an opportunity for efficient fine-grained fault tolerance. Specifically, we observe that the replicas originally used for local access in vertex computation can be reused to backup data of vertices and edges, while the synchronization messages between a master vertex and its replicas can be reused to maintain the freshness of replicas.

To leverage vertex replicas for fault tolerance, it is necessary that each vertex has at least one replica; otherwise extra replicas for these vertices have to be created, which incurs additional overhead for communication during normal execution. Fig. 3a shows the percentage of vertices without replicas on a 50-node cluster for a set of real-world and synthetic graphs using the default hash-based (random) partitioning. Only GWeb [36] and LJournal [29] contain more than 10 percent of such vertices, while others only contain less than 1 percent vertices. The primary source of vertices without replicas is from selfish vertices, which have no out-edges (e.g., vertex 7 in Fig. 1). For most graph algorithms, the value of a selfish vertex has no consumer and only depends on ingoing neighbors. Consequently, there is no need to create extra replicas for selfish vertices. In addition, the performance cost in communication depends on the number of replicas, which is several times the number of vertices. Fig. 3b illustrates the percentage of extra replicas required for fault tolerance regardless of selfish vertices, which is less than 0.15 percent for all dataset.

One challenge is that, for vertex-cut, there are no replicated edges that can be used to tolerate the loss of edges during recovery. Fortunately, we observe that very few graph algorithms will update the state of edges during computation. This means that a graph-parallel system can hide the cost of storing edges to persistent storage by overlapping it with graph execution, which avoids the runtime overhead from synchronizing edge states.

Fast Recovery by Leveraging Scale. For checkpoint-based fault tolerance, recovery from a snapshot on the persistent storage cannot harness all resources in the cluster. The I/O performance of a single node becomes the bottleneck of recovery, which does not scale with the increase of nodes. Further, a checkpoint-based fault tolerance mechanism also depends on standby nodes to take over the workload on crashed nodes.

Fortunately, the replicas of a vertex scattered across the entire cluster provides a new opportunity to recover a machine failure in parallel, which is inspired by the fast recovery in DRAM-based storage system (e.g., RAMCloud [17]). Specifically, Imitator leverages a number of surviving nodes to recover states of a single crashed node in parallel. Actually, the time for recovery may be less with more nodes if the replicas selected for recovery can be evenly assigned to all nodes.

In addition, an even distribution of replicas for vertices on the crashed node further provides the possibility to support migrating the workload on crashed nodes to all surviving nodes without using additional standby nodes for recovery. This may also help reserve the load balance of execution after recovery.

3.2 Overall Design of Imitator

Based on the above observation, we propose Imitator, a replication-based fault tolerance scheme for graph-parallel systems. Unlike prior systems, Imitator employs replicas of a vertex to provide fault tolerance rather than periodically

4. For brevity, we assume that the failure occurs at the middle of an interval. The failure point only impacts on the *replay* time, which is usually proportional to the number of lost iterations.

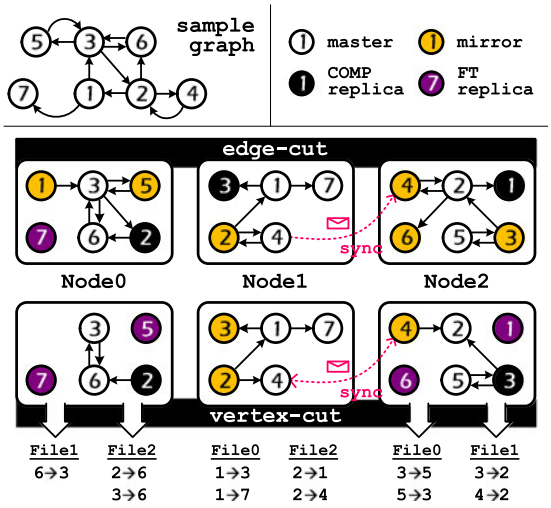


Fig. 4. A sample of replicas in Imitator.

checkpointing graph states to persistent storage. The replicas of a vertex inherently provide a remote consistent backup, which is synchronized during each global barrier. When a node crashes, its workload (vertices and edges) will be reconstructed on a standby node or evenly migrated to all surviving nodes.

Note that Imitator assumes a fail-stop model where a machine crash will not cause wild or malicious changes to other machines. How to extend Imitator to support more complicated faults like byzantine faults [37] will be our future work.

Execution Flow. Imitator extends existing synchronous execution flow with the detection of potential node failures and seamless recovery. As shown in Algorithm 1, each iteration consists of three steps. First, all vertices are updated using neighboring vertex states in the computation phase (line 5). Second, an update of vertex states is synchronized from a master vertex to its replicas in the communication phase through message passing (line 6). Note that all messages have been received before entering the global barrier. Finally, all new vertex states are consistently committed within a global barrier (lines 14 and 15). Imitator employs a highly available and persistent distributed coordination service (e.g., Zookeeper [38]) to provide barrier-based synchronization and distributed shared states among nodes, whichi is inherited from Apache Hama [9], [31].[5] Node failures will be detected before (line 7) and after (line 16) the global barrier. Before recovery, each node must enforce the consistency of its local graph states. If a failure occurs before the global barrier, each surviving node should roll back its states (line 9) and execution flows (line 12) to the beginning of the current iteration, since messages from crashed nodes may be lost. Imitator provides two alternative recovery mechanisms: *Rebirth* and *Migration*. For Rebirth, a standby node will join the global synchronization (line 2), and reconstruct the graph states of the crashed nodes from all surviving nodes (lines 3, 10 and 18). For Migration, the vertices on crashed nodes will be reconstructed on all surviving nodes (lines 11 and 19). Note that Imitator no needs to timely detect machine failures, since the recovery is always delayed to the next global barrier of the current iteration. At that time, all surviving nodes have finished their own tasks, and can help recover the states of the crashed node. Therefore, Imitator uses heartbeat communications to a central master node with a conservative interval (e.g., 500 ms) which can safely determine the machine failures.

5. Each node will create a file in a shared directory of Zookeeper and check if the number of files equals the number of nodes. If not, the node will wait for the notification. The last node will ask Zookeeper to wake up all waiting nodes.

4 Managing Replicas

Many graph-parallel systems [10], [11], [13], [26], [39] construct a local graph on each node by *replicating* vertices to avoid remote accesses during vertex computation. As shown in Fig. 4, the sample graph is partitioned to three machines using random edge-cut or vertex-cut. Vertices (with their edges) or edges are evenly assigned to three machines, and replicas are created to provide local vertex accesses. These replicas will be synchronized with their master vertices to maintain consistency. Imitator reuses these *replicas* as consistent backups of vertices for recovery from failures. However, replication-based fault tolerance requires that every vertex has at least one replica with exactly the same states with the master vertex, while existing replicas are only with partial states. Further, not all vertices have replicas. Finally, since the master vertex in vertex-cut will not be co-located with all of its edges, replicating edges with the master vertex is not always feasible to tolerate the loss of edges.

Algorithm 1. Imitator Execution Model

```
   Input: Date Graph G = (V, E, D)
   Input: Initial active vertex set 𝕍
 1 if is_newbie() then                                   // new node
 2     newbie_enter_leave_barrier();
 3     iter = newbie_rebirth_recovery();
 4 while iter < max_iter do
 5     compute();
 6     send_msgs();
 7     state = enter_barrier();
 8     if state.is_fail() then                           // node failure
 9         rollback()
10         if is_rebirth() then rebirth_recovery(state);
11         else migration_recovery(state);
12         continue;
13     else                                              // normal execution
14         commit_state();
15         iter ++;
16     state = leave_barrier();
17     if state.is_fail() then                           //node failure
18         if is_rebirth() then rebirth_recovery(state);
19         else migration_recovery(state);
```

This section describes extensions for fault-tolerance oriented replication, creating full-state replicas, replicating edges and an optimization for selfish vertices. Here, we only focus on creating at least one replica to tolerate one machine failure for brevity; creating more replicas can be done similarly.

4.1 Fault Tolerant (FT) Replica

The original replication optimized for local accesses may leave some vertices without replicas. For example, the

internal vertex (e.g., vertex 7 in Fig. 4) has no replica as all its edges are stored at the same node. A failure of Node1 may cause an irrecoverable state for vertex 7.

For such an internal vertex, Imitator creates an additional *fault tolerant replica* on another machine when loading the graph. There is no constraint on the location of these replicas, which provides an opportunity to balance the workload among nodes to hide the performance overhead. Before assignment, the number of replicas and internal vertices are exchanged among nodes. Each node proportionally assigns FT replicas to the remaining nodes. For example, vertex 7 has no computation replicas and its additional FT replica is assigned to Node0, which has fewer replicas, as shown in Fig. 4.

4.2 Full-State Replica (Mirror)

The replica to provide local access does not have full states to recover the master vertex, such as the location of replicas. However, it is not efficient to upgrade all replicas to be homogeneous with their masters, which will cause excessive memory and communication overhead. Imitator selects one replica to be the homogeneous replica with the master vertex, namely *mirror*.

Most additional states in mirrors are static, such as the location of replicas, which are replicated during graph loading. The remaining states are dynamic, such as whether a vertex is active or inactive in next iteration, and should be transferred with synchronization messages from a master to its mirrors in each iteration of computation.

As mirrors are responsible to recover their masters on a crashed node, the distribution of mirrors may affect the scalability of recovery. Since the locations of mirrors are restricted by the locations of all candidate replicas, each machine adopts a greedy algorithm to evenly assign mirrors on other machines independently: each machine maintains a counter of existing mirrors, and the master always assigns mirrors to replicas whose hosting machine has the least mirrors so far.

Note that the FT replica is always the mirror of a vertex. As shown in Fig. 4, the mirrors of vertex 1 and 4 on Node1 are assigned to Node0 and Node2 accordingly, and the mirror of vertex 7 has to be assigned to Node0.

4.3 Replicating Edges

To recover the lost edges upon a crash, Imitator also requires replicating edges. For systems using edge-cut, since the master vertex locally connects with all of its edges (see Fig. 4), it would be natural to include all edges into the full states of the masters and replicate them to the mirrors. However, for a system using vertex-cut, there is no replicated edges and the edges of the same vertex may connect to replicas on multiple nodes, such as the vertex 2 in Fig. 4.

A simple way for vertex-cut is to accumulate all edges on the master as the edge-cut and replicate them to mirrors. Nevertheless, it will incur high communication cost, excessive memory consumption, and even load imbalance, especially for natural graphs [11], [13].

Fortunately, we observe that very few graph algorithms will update the states of edges during computation. Hence, we let each node replicate edges of its own graph part to persistent storage (e.g., HDFS [33]) during the graph loading phase. For cases where the state of edges is updated during execution, Imitator will incrementally log the updates to the corresponding edge-ckpt files by overlapping the logging with the graph computation on vertices. Since the edges on the crashed nodes will be evenly migrated to all surviving nodes during Migration-based recovery (Section 5.2), the edges of each node are further partitioned and stored to multiple edge-ckpt files. Note that each node simply assumes that all others are survived and will exclusively receive one edge-ckpt file during recovery. To reduce communication cost during recovery, the edge will be assigned to the edge-ckpt file corresponding to the node hosting the master or mirror of the target vertex. As shown in Fig. 4, all four edges on Node2 are assigned to two edge-ckpt files (i.e., File0 and File1), and the edges (3,2) and (4,2) are stored to File1.

4.4 Optimizing Selfish Vertices

The main overhead of Imitator during normal execution is from the synchronization of additional FT replicas. According to our analysis, many vertices requiring FT replicas have no neighboring vertices consuming their vertex data (selfish vertices). For example, vertex 7 has no out-edges in Fig. 4. Further, for some algorithms (e.g., PageRank), the new vertex data is computed only according to its neighboring vertices.

For such vertices, namely *selfish vertices*, Imitator only creates an FT replica for recovery, and never synchronizes them with their masters during normal execution. During recovery, the static states of selfish vertices can be obtained from its FT replicas, and dynamic states can be re-computed using the neighboring vertices.

5 Recovery

The main issue of recovery is knowing which vertices, either master vertices or other replicas, have been assigned to the crashed node. A simple approach is adding a layer to store the location of each vertex. This, however, may become a new bottleneck during the recovery. Fortunately, when a master vertex creates its replicas, it knows its replicas' locations. Thus, by storing its replicas' locations, a master vertex knows if its replicas are assigned to the crashed node. As the mirror (i.e., a full-state replica) is responsible for recovery when its master vertex is lost, the master vertex needs to synchronize its own location to its mirror as well. Further, all edges are replicated with the full-state of mirrors for edge-cut or into the *edge-ckpt* files for vertex-cut.

During recovery, each surviving node will check in parallel whether master or replica vertices and edges related to the failed nodes have been lost and reconstruct such lost graph states accordingly. As each remaining node has the complete information of its related graph states, such checking and reconstruction can be done in a decentralized way and in parallel.

Imitator provides two strategies for recovery: Rebirth-based recovery, which recovers graph states in crashed nodes to standby ones; Migration-based recovery, which scatters vertices on the crashed nodes to surviving ones.

5.1 Rebirth-Based Recovery

During recovery, the location information of vertices will be used by master vertices or mirrors to check whether there are some vertices to recover. All states of edges will be

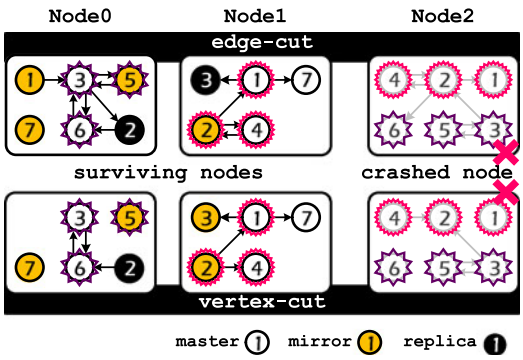


Fig. 5. A sample of Rebirth-based recovery in Imitator.

obtained from adjacent vertices on surviving nodes or *edge-ckpt* files on persistent storage. Rebirth-based recovery comprises three steps: *Reloading*, where the surviving nodes send the recovery messages to the standby nodes to help it recover states; *Reconstruction*, which reconstructs the states (mainly the graph topology) necessary for computation; and *Replay*, which redoes some operations to get the latest states of some vertices.

5.1.1 Reloading

First, through checking the location of its replicas, a master vertex will know whether there are any of its replicas located in the crashed nodes. If so, the master vertex will generate messages to recover such replicas. If a master vertex is on the crashed nodes, its mirror will be responsible to recover this crashed master. Based on this rule, each surviving node can just use the information from its local vertices to decide whether it needs to participate in the recovery process.

For the sample graph in Fig. 5, Node2 crashed during computation. After a new standby node (i.e., machine) awakes Node0 and Node1 from a global barrier, these two nodes will check whether they have some vertices to recover. Node1 will check its master vertices (master vertex 1, 4, and 7), and find that there are some replicas (replica 1 and mirror 4) on the crashed node. Hence, it needs to generate two recovery messages to recover replica 1 and mirror 4 on the new node. Further, Node1 will also check its mirrors to find whether there is any mirror whose master was lost. It then finds that the master of vertex 2 was lost, and thus generates a message to recover master 2. Node0 will act the same as Node1.

Second, for systems using edge-cut, all edges on crashed nodes are also stored within the masters or mirrors on surviving nodes, which can be reloaded with the vertices. For example, the five edges connected with vertex 2 will be included within the message from the mirror of vertex 2 on Node1. For a system using vertex-cut, the edge-ckpt files with all edges of the crash node can be directly reloaded from persistent storage, which can be done through overlapping with the reloading vertices from the surviving nodes.

The surviving nodes also need to send some global states to the new node, such as the number of iterations so far. All the recovery messages are sent in a batched way to reduce communication cost.

5.1.2 Reconstruction

For the new node, there are three types of states to reconstruct: the graph topology, runtime states of vertices, and global states (e.g., number of iterations so far). The last two types of states can be retrieved directly from recovery messages. The graph topology is a complex data structure, which is non-trivial to recover.

A simple way to recover the graph topology is to use the raw edge information (the "points-to" relationship between vertices) and redo operations of building topology in the graph loading phase. In this way, after receiving all the recovery messages, the new node will create vertices based on the messages (which contain the vertex types, the edges and the detailed states of a vertex). After creating all vertices, the new node will use the raw edge information on each vertex to build the graph topology. One issue with this approach is that building the graph topology can only start after creating all vertices. Further, due to the complex "points-to" relationship between vertices, it is not easy to parallelize the topology building process.

To expose more parallelism, Imitator uses enhanced edge information for recovery. Since all vertices are stored in an array in each machine, the topology of a graph is represented by the array index. This means that if there is an edge from vertex A to vertex B, vertex B will have a field to store the index of vertex A in the array. Hence, if Imitator can ensure a vertex is placed at the same position of the vertex array in the new node, reconstruction of graph topology can be done in parallel on all the surviving nodes.

To ensure this, Imitator also replicates the master's position to its mirror with other states in the graph loading phase. When a mirror recovers its master, it will create the master vertex and its edges, and then encode the vertex and the master position into the recovery message. On receiving the message, the new node just needs to retrieve the vertex from the message and put it at the given position. Recovering replicas can be done in the same way.

Since every crashed vertex only needs one vertex to do the recovery, there is only one recovery message for one position. Thus, there is *no contention* on the array (which is thus lock-free) and can be done immediately when receiving the message. Hence, it is completely decentralized and can be done in parallel. Note that there is no explicit reconstruction phase for this approach because the reconstruction can be done during the reloading phase when receiving recovery messages.

5.1.3 Replay

Imitator can recover most states of a vertex directly from the recovery message, except the activation state, which cannot be timely synchronized between masters and mirrors. The reason is that a master vertex may be activated by some neighboring vertices that are not located on the same node. When a master replicates its states to its mirrors, the master may still not be activated by its remote neighbors. Hence, the activation state can only be recovered by replaying the activation operations. However, the neighboring vertex of a master vertex might also locate at the crashed node. As a result, a master vertex needs to replicate its activation information (which vertices it should activate) to its mirrors. A vertex (either master or mirror) doing recovery will attach the corresponding activation information to the recovery message. The new node will re-execute the activation operations according to these messages on all the vertices.

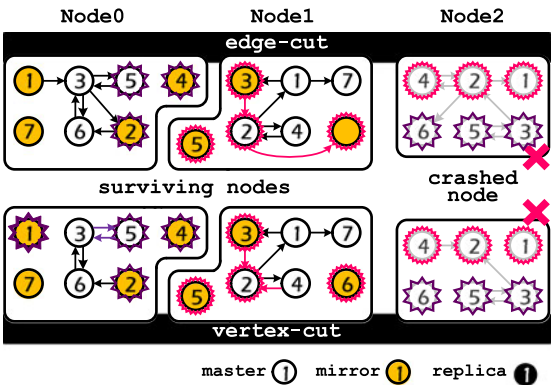


Fig. 6. A sample of Migration-based recovery in Imitator.

5.2 Migration-Based Recovery

When there are no standby nodes for recovery, Migration based recovery will scatter graph computation from the crashed nodes to surviving ones. Fortunately, the mirrors, which are isomorphic with their masters, provide a convenient way to migrate a master vertex from one node to another. Other data to be used by the new master in future computation can be retrieved from its neighboring vertices.

The Migration approach also consists of three steps: *Reloading*, *Reconstruction*, and *Replay*, of which the processes are only slightly different from the Rebirth approach. In the following, we will use the example in Fig. 6 as a running example to illustrate how Migration based recovery works.

5.2.1 Reloading

The main differences between Rebirth and Migration for *reloading* is that mirror vertices will be "promoted" to masters and take over the computation tasks for future execution.

On detecting a failure, all surviving nodes will get the information about the crashed ones from the master node of a cluster. Surviving nodes will scan through all of their mirrors to find whose masters were on the crashed nodes. In Fig. 6, they are mirror 5 on Node0 and mirror 2 on Node1. These mirrors will be "promoted" as new masters.

For systems using edge-cut, the "promoted" mirrors already have the information of their edges. For systems using vertex-cut, the surviving nodes will reload edges within edge-ckpt files of crashed nodes in parallel, which can well overlap with vertex promotion. For example, in Fig. 6, Node1 will reload the edge (3,2) and (4,2) from File1 of Node2.

After recovering vertices and edges, the "promoted" mirrors send the new location information to their surviving replicas, create additional FT replicas to retain the original fault tolerance level and select new mirrors. In Fig. 6, FT replica 4 is created on Node0 and selected as the mirror. In addition, due to reloading edges on a different node, some new replicas are necessary to retain local access semantics for computation. Replica 6 on Node1 under edge-cut in Fig. 6 illustrates this case. All surviving nodes will cooperate to create such replicas.

5.2.2 Reconstruction

During *reconstruction*, all surviving nodes will assemble new graph states from the recovery messages sent in the *reloading* phase. After the *reconstruction* phase, the topology of the graph and most of the states of the vertices (both masters and replicas) are migrated to the surviving nodes.

5.2.3 Replay

The Migration approach also needs to fix the activation states for new masters. However, the Rebirth approach needs to fix such states for all recovered masters, while the Migration approach only needs to fix the states of newly promoted masters, which are only a small portion of all master vertices on one machine. Hence, we choose to treat these new masters especially instead of redoing the activation operation on all the vertices. Imitator checks whether a new master is activated by some of its neighbors or not. If so, Imitator will correct the activation states of the new master. When finishing the *Replay* phase, the surviving nodes can now resume the normal execution.

5.3 Additional Failure Models

5.3.1 Multiple Machine Failures

To tolerate multiple nodes failure at the same time, Imitator needs to ensure that the number of mirrors for each vertex in Imitator is equal or larger than the expected number of machines to fail. When a single machine failure happens, if all mirrors participate the recovery, it is a waste of network bandwidth. Hence, during graph loading, each mirror is assigned with an ID; only the surviving mirrors with the lowest ID will do the recovery work. Since a mirror has the location information of other mirrors and the new coming node's logic ID of this job, mirrors need not communicate with each other to elect a mirror to do recovery.

5.3.2 Other Types of Failures

When a failure happens during the system is loading graph, since the computation has not started, we just restart the job. If a failure happens during recovery, such a failure is almost the same as the failure happening during the normal execution. Hence, we just restart the recovery procedure.

There is a single master node for a cluster, and it is only in charge of job dispatching and failure handling. It has nothing to do with the job execution. Since there are a lot of prior work to address the single master failure, we do not consider the failure of master node in this paper.

6 Evaluation

We first incorporated the design of Imitator into Cyclops [26], which extends Apache Hama [9], [31] (an open source clone of Pregel) by vertex replication instead of pure message passing as the communication mechanism. It provides multiple state-of-the-art edge-cuts. The support of fault tolerance requires no source code changes to graph algorithms. To measure the efficiency of Imitator, we use four typical graph algorithms (PageRank, Alternating Least Squares (ALS), Community Detection (CD) and Single Source Shortest Path (SSSP)) to compare the performance and scalability of different systems and configuration. We also provide a case study to illustrate the effectiveness of Imitator by illustrating the execution of different recovery approaches under injected machine failures.

TABLE 1
A Collection of Real-World and Synthetic Graphs

Algorithm	Graph	$\|V\|$	$\|E\|$
PageRank	GWeb [36]	0.87 M	5.11 M
	LJournal [29]	4.85 M	70.0 M
	Wiki [32]	5.72 M	130.1 M
ALS	SYN-GL [11]	0.11 M	2.7 M
CD	DBLP [26]	0.32 M	1.05 M
SSSP	RoadCA [36]	1.97 M	5.53 M

$|V|$ and $|E|$ denote the number of vertices and edges respectively.

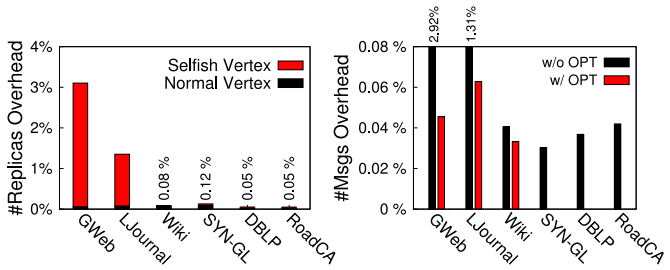


Fig. 8. The overhead of (a) #replicas and (b) #msgs for Imitator w/ and w/o selfish optimization.

To demonstrate the generality, we further incorporated Imitator into PowerLyra [13], which provides a set of state-of-the-art vertex-cuts, and also performed a detailed experiment with various workloads and configurations during normal execution and recovery. It should be noted that our implementation of checkpoint-based fault tolerance on both Cyclops and PowerLyra can be viewed as near-optimal cases, which are several times faster than the default checkpoint implementations [40], [41].

6.1 Experimental Setup

All experiments are performed on a 50-node EC2-like cluster. Each node has four AMD Opteron cores, 10 GB of RAM, four 500 GB SATA HDDs, and is connected via a 1 GigE network. We use HDFS on the same cluster as the distributed persistent storage to store input files and checkpoints. The replication factor of HDFS is set to three.

Table 1 lists a collection of algorithms and large graphs for our experiments on Cyclops. The SSSP algorithm requires the input graph to be weighted. Since the RoadCA [36] graph is not originally weighted, we synthetically assign a weight value to each edge, where the weight is generated based on a log-normal distribution ($\mu = 0.4$, $\sigma = 1.2$) from the Facebook user interaction graph [42].

For brevity, we directly report the results of checkpoint-based fault tolerance with the interval of one iteration unless otherwise stated. This means that the runtime overhead is the upper bound, while the recovery time is the lower bound. Since the checkpoint time is commonly inversely proportional to the interval, readers can roughly estimate the performance overhead of checkpoint-based approaches for different intervals.

6.2 Runtime Overhead

Fig. 7 shows the runtime overhead due to applying different fault tolerance mechanisms on the baseline system (Cyclops [26]). The overhead of Imitator is less than 3.7 percent for all algorithms with all datasets, while the overhead of the checkpoint-based fault tolerance is very large, varying from 65 percent for PageRank on Wiki to 449 percent for CD on DBLP. Even using in-memory HDFS, the checkpoint-based approach still incurs performance overhead from 33 to 163 percent partly due to the cross machine triple replication in HDFS. In addition, writing to memory also causes significant pressure on memory capacity and bandwidth to the runtime, occupying up to 42.1 GB extra memory for SSSP on RoadCA.

The time of checkpointing once is from 1.08 to 3.17 seconds for different size of graphs, since the write operations to HDFS can be batched and are insensitive to the data size. The overhead of each iteration in Imitator is lower than 0.05 seconds, except 0.22 seconds for Wiki, which is still several tens of times faster than checkpointing.

6.3 Overhead Breakdown

Fig. 8a shows the extra replicas among all the replicas used for fault tolerance. The rates of extra replicas are all very small without selfish vertices, even the largest rate is only 0.12 percent. Fig. 8b shows the redundant messages among the total messages during the normal execution. Since the rate of extra replicas is very small, the additional messages rate is very small, with only 2.92 percent for the worst case. When enabling the optimization for selfish vertices, the messages overhead is lower than 0.1 percent.

6.4 Efficiency of Recovery

Replication-based fault tolerance provides a good opportunity to fully utilize the entire resources of the cluster for recovery. As shown in Table 2, the replication-based recovery outperforms checkpoint-based recovery by up to 6.86X (from 3.93X) and 17.67X (from 3.55X) for Rebirth and Migration approaches accordingly. Overall, Imitator can

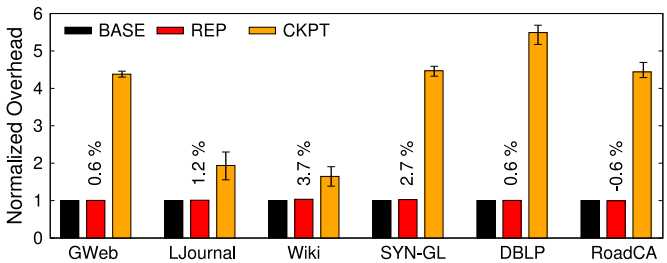


Fig. 7. A comparison of runtime overhead between replication (REP) and checkpoint (CKPT) based fault tolerance over baseline (Cyclops) w/o fault tolerance (BASE).

TABLE 2
The Recovery Time (Seconds) of Checkpoint (CKPT), Rebirth (REB) and Migration (MIG)

Algorithm	Graph	CKPT	REB	MIG
PageRank	GWeb	8.17	2.08	1.20
	LJournal	41.00	8.85	2.32
	Wiki	55.67	14.12	3.40
ALS	SYN-GL	6.86	1.00	1.28
CD	DBLP	3.88	0.67	1.09
SSSP	RoadCA	12.06	2.27	1.57

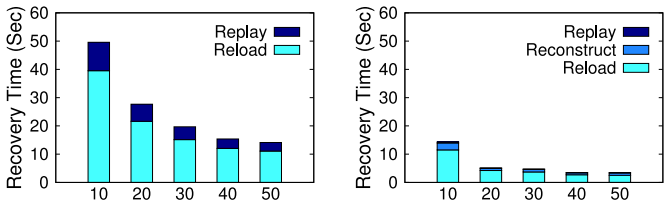


Fig. 9. The recovery time of Rebirth (a) and Migrate (b) on Imitator for PageRank with the increase of nodes.

recover 0.95 million and 1.43 million vertices (including replicas) from one failed node in just 2.32 and 3.4 seconds for LJournal and Wiki dataset accordingly.

For large graphs (e.g., LJournal and Wiki), the performance of Migration is relatively better than that of Rebirth, since it avoids data movement (e.g., vertex and edge values) in the reloading phase and distributes replaying operations to all surviving nodes rather than on the single new node. On the other hand, for small graphs (e.g., SYN-GL and DBLP), the performance of Rebirth is relatively better than that of Migration, since there are multiple rounds of message exchanges in Migration. This causes the slowdown to recovery, ranging from 28 to 63 percent, compared with Rebirth.

6.5 Scalability of Recovery

We evaluate the recovery scalability of Imitator for PageRank with the Wiki dataset using different numbers of nodes that participate in the recovery. As shown in Fig. 9, both recovery approaches scale with the increase of nodes, since all nodes can participate in the reloading phase. Because the local graph has been constructed in the reloading phase, there is no explicit reconstruction phase for Rebirth. Further, the replay operations are only executed in the new node for Rebirth, while are distributed to all surviving nodes for Migration.

6.6 Impact of Graph Partitioning

To analyze the impact of different graph partitioning algorithms, we implemented Fennel [20] on Imitator, which is a heuristic graph partitioning algorithm. As shown in Fig. 10a, compared to the default hash-based partitioning, Fennel significantly decreases the replication factor for all datasets, reaching 1.61, 3.84 and 5.09 for GWeb, LJournal and Wiki respectively.

Fig. 10b illustrates the overhead of Imitator under the Fennel partitioning. Due to lower replication factor, Imitator requires more additional replicas for fault tolerance, which also result in an increase of message overhead.

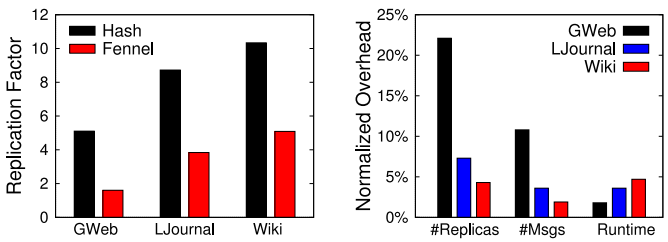


Fig. 10. (a) The replication factor of different partitioning schemes. (b) The overhead of Imitator using Fennel algorithm.

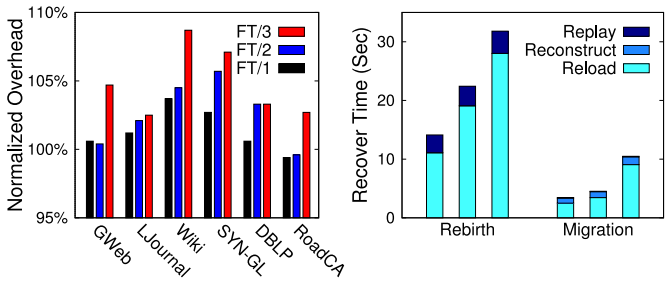


Fig. 11. (a) The runtime overhead, and (b) The recovery time for tolerating 1, 2, and 3 machine failure(s).

However, the runtime overhead is still small, ranging from 1.8 to 4.7 percent.

6.7 Handling Multiple Failures

When Imitator is configured to tolerate multiple machine failures, there will be more extra replicas to add. The overhead tends to be larger. Fig. 11 shows the overall cost when Imitator is configured to tolerate 1, 2 and 3 machine failure (s). As shown in Fig. 11a, the overhead of Imitator is less than 10 percent even when it is configured to tolerate 3 nodes failures simultaneously.

Fig. 11b shows the recovery time of the largest dataset, Wiki, when different numbers of nodes crashed. For Rebirth, since the surviving nodes need more messages to exchange when the crashed nodes increase, the time to send and receive recovery messages increases. However, the time to rebuild graph states and replay some pending operations is almost the same as that of a single machine failure. Since Migration strategy harnesses the cluster resource for recovery, the time of all operations in Migration is relatively small.

6.8 Memory Consumption

As Imitator needs to add extra replicas to tolerate faults, we also measure the memory overhead. We use *jstat*, a memory tool in JDK, to monitor the memory behavior of the baseline system and Imitator. Table 3 illustrates the result of one node of the baseline system and Imitator on our largest dataset Wiki. If Imitator is configured to tolerate one machine failure during execution, the memory overhead is modest, and the memory usage of the baseline system and Imitator is comparable.

6.9 Case Study

Fig. 12 presents a case study for running PageRank using LJournal with none or one machine failure during the execution of 20 iterations. Different recovery strategies are applied to illustrate their performance. The symbols, BASE,

TABLE 3
Memory and GC Behavior of Imitator with Different Fault Tolerance Setting for PageRank on Wiki

Config	Max Cap (GB)	Max Usage (GB)	Young/Full GC	
			Number	Time (Sec))
w/o FT	3.85	2.76	40/15	13.7/13.4
FT/1	5.05	3.70	50/29	19.9/21.7
FT/2	6.24	4.51	55/29	23.6/26.1
FT/3	6.99	4.91	58/30	25.7/29.7

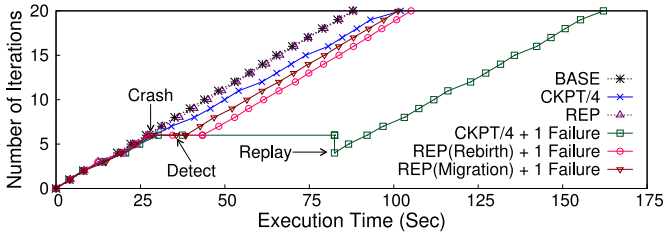


Fig. 12. An execution of PageRank of LJournal with different fault tolerance settings. One failure occurs between the 6th iteration and the 7th iteration.

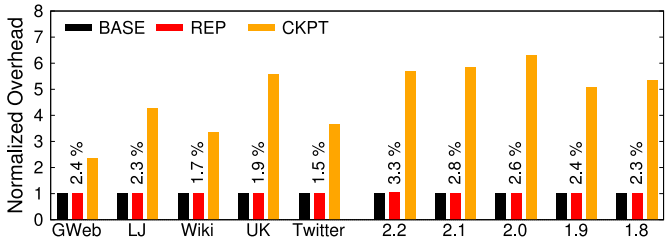


Fig. 13. A comparison of runtime overhead between replication (REP) and checkpoint (CKPT) based fault tolerance over baseline (PowerLyra) w/o fault tolerance (BASE).

REP, and CKPT/4, denote the execution of the baseline, replication and checkpoint-based fault tolerance systems without failure accordingly, where others illustrate the cases with a failure between the 6th and 7th iterations. Note that the interval of checkpointing is four iterations.

The scheme of failure detection is the same for all strategies, of which the time span is about 7 seconds. For the recovery speed, the Migration strategy, of which recovery time is about 2.6 seconds, is the fastest due to the fact that it harnesses all resources and minimizes data movements. The Rebirth strategy has a time span of 8.8 seconds. This still outperforms the 45 seconds recovery time of CKPT/4, which does the periodic checkpoint with an interval of 4 iterations, due to fully exploiting network resources and without accessing distributed file system.

After the recovery has finished, REP with Rebirth can still execute at full speed, since the execution environment before and after the failure is the same in this approach. On the other hand, the REP with Migration is slower since the available computing resource has decreased, but only slightly. For the CKPT/4, it still has to replay 2 lost iterations after a long time recovery.

6.10 Effectiveness on Vertex-Cut

To study the performance of replication-based fault tolerance for systems using vertex-cut, we further evaluation the runtime overhead and recovery time of Imitator with various configuration, compared to the baseline system (PowerLyra [13]). Unless mentioned, the partitioning algorithm adopts hyrbid-cut as default. Table 4 lists a collection of real-world and synthetic graphs for experiments on PowerLyra.

Runtime Overhead. Fig. 13 illustrates the performance cost to applying different fault tolerance mechanisms for PageRank (20 iterations) with various real-world and synthetic graphs. Compared to the baseline without fault tolerance, the normalized overhead of Imitator is always less than 3.3 percent (from 1.5 percent), while that of checkpoint-based approach is very large, varying from 531 to 135 percent.

Efficiency of Recovery. Imitator also provides a good recovery performance due to fully utilizing the entire resources of the cluster. As shown in Table 5, the replication-based recovery outperforms checkpoint-based recovery by up to 7.66X (from 1.70X) and 7.18X (from 1.29X) for Rebirth and Migration approaches accordingly.

Overall, Imitator can recover 5.2 million vertices (including replicas) and 26.9 million edges from one failed node in just 42.0 and 33.4 seconds for Twitter using Rebirth and Migration approaches accordingly.

For large graphs (e.g., Twitter), the performance of Migration is relatively better than that of Rebirth, since it can read edge-ckpt files from persistent storage in parallel by all surviving nodes in the reloading phase. On the other hand, for small graphs (e.g., GWeb), the performance of Rebirth is better, because it can overlap the reloading of edges from persistent storage with that of vertices from surviving nodes. Further, Migration requires more rounds of message exchanges.

TABLE 4
A Collection of Real-World Graphs and Synthetic Graphs with Varying Power-Law Constant (α) and Fixed 10 Million Vertices

Graph	$\|V\|$	$\|E\|$
GWeb [36]	0.9 M	5.1 M
LJ [43]	5.4 M	79 M
Wiki [32]	5.7 M	130 M
UK-2005 [44]	40 M	936 M
Twitter [45]	42 M	1.47 B

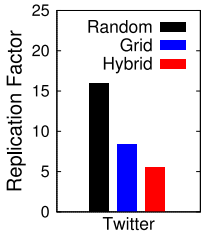
α	$\|V\|$	$\|E\|$
2.2	10 M	39 M
2.1	10 M	54 M
2.0	10 M	105 M
1.9	10 M	249 M
1.8	10 M	673 M

$|V|$ and $|E|$ denote the number of vertices and edges respectively.

TABLE 5
The Recovery Time (Seconds) of Checkpoint (CKPT), Rebirth (REB) and Migration (MIG) for Real-World and Synthetic Graphs

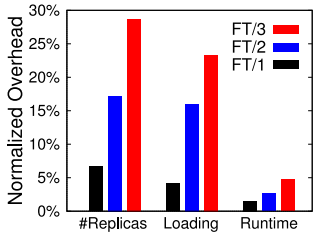
Graph	CKPT	REB	MIG
GWeb	1.8	0.8	1.4
LJ	12.5	3.1	4.7
Wiki	13.1	3.9	4.8
UK	216.0	28.2	30.1
Twitter	183.7	42.0	33.4

α	CKPT	REB	MIG
2.2	8.0	2.1	4.6
2.1	9.2	2.8	5.0
2.0	10.6	3.8	7.1
1.9	14.3	6.0	8.6
1.8	22.3	13.1	11.9



	Random	Grid	Hybrid
Rep-Factor	0.01%	0.02%	6.71%
Loading	4.10%	5.35%	6.26%
Runtime	0.16%	0.73%	1.49%
REB	63.8s	47.4s	42.0s
MIG	45.6s	42.8s	33.4s

Fig. 14. (a) The replication factor and (b) The overhead and recovery time of Imitator using different partitioning algorithms for PageRank with Twitter.



Config	Rebirth	Migration
FT/1	42.0	33.4
FT/2	50.6	45.9
FT/3	63.1	70.0

Fig. 15. (a) The runtime overhead of normal execution, and (b) the recovery time (seconds) of Rebirth (REB) and Migration (MIG) on PowerLyra for tolerating 1, 2, and 3 machine failure(s).

Impact of Graph Partitioning. The default graph partitioning schemes of PowerLyra is the Hybrid-cut [13], which is highly optimized for natural graphs. It leads to quite fewer replicas and better performance. As shown in Fig. 14a, the replication factor of Hybrid-cut for Twitter on our cluster is only 5.56, which is lower than that of Grid-cut [23] and Random-cut [11] (i.e., 8.34 and 15.96).

In fact, using Hybrid-cut can be regarded as a worst case for Imitator due to fewer candidate replicas for fault tolerance. As shown in Fig. 14b, the runtime overhead of Imitator using Random-cut and Grid-cut is just 0.16 and 0.73 percent. However, as shown in Table 6, we believe that the tiny difference of runtime overhead will not impact the advantages of using a better graph partitioning like Hybrid-cut. Further, compared to Hybrid-cut, the higher replication factor of Random-cut and Grid-cut will also cause slowdown to recovery time due to larger replication factors, reaching 51.9 and 12.9 percent for Rebirth and 36.5 and 28.1 percent for Migration respectively.

Multiple Failures. To tolerate multiple machine failures, Imitator needs more extra FT replicas, which will increase runtime overhead and recovery time. As shown in Fig. 15a, the runtime overhead of Imitator for PageRank with Twitter is only 4.69 percent even when it is configured to tolerate three machine failures simultaneously.

Fig. 15b shows the recovery time when different numbers of nodes crashed. For Rebirth, since all of the new nodes can reload edges from persistent storage in parallel, the time is almost the same as that of a single machine failure. However, for Migration, all surviving nodes need to reload more edges from persistent storage with the increase of crashed nodes. Therefore, the increase of recovery time of Rebirth is relatively smaller than that of Migration.

TABLE 6
A Comparison of Runtime Overhead and Communication Cost per Iteration Between Different Partitioning Algorithms for Tolerating 1, 2, and 3 Machine Failure(s) on PageRank with Twitter

	Random	Grid	Hybrid
	Execution Time (Sec)		
w/o FT	59.3	29.2	16.5
FT/1	59.4 (+0.16%)	29.5 (+0.73%)	16.8 (+1.49%)
FT/2	59.7 (+0.60%)	29.6 (+1.27%)	17.0 (+2.66%)
FT/3	60.0 (+1.14%)	29.9 (+2.27%)	17.3 (+4.69%)
	Communication Cost (GB)		
w/o FT	1.85	0.46	0.21
FT/1	1.86 (+0.92%)	0.47 (+1.82%)	0.22 (+5.59%)
FT/2	1.89 (+2.22%)	0.48 (+3.93%)	0.24 (+13.05%)
FT/3	1.91 (+3.31%)	0.49 (+6.70%)	0.26 (+21.49%)

We further study the impact of graph partitioning on tolerating multiple failures. As shown in Table 6, the runtime overheads of Random-cut and Grid-cut to tolerate multiple failures are slightly smaller than that of Hybrid-cut due to introducing fewer FT replicas. For example, the runtime overhead to tolerate three machine failures simultaneously is 1.14, 2.27, and 4.69 percent respectively. However, the impact of fault-tolerance on runtime overhead is negligible compared to that of adopting different partitioning algorithms, which is confirmed by our evaluation on communication cost. It means that Imitator can almost preserve the behavior and not interfere with the choice of partitioning algorithms.

Memory Consumption. As Imitator needs to add extra replicas to tolerate faults, we also measure the memory overhead. Different to edge-cut, vertex-cut will not replicate edges, which dominate the memory consumption for most graphs. For example, the number of edges ($|E|$) is about 35 times larger than the number of vertices ($|V|$) in Twitter (see Table 4). Therefore, the memory overhead of Imitator is much lower than the increase of replication factor. Table 7 shows the total memory consumption on our 50-node cluster with different partitioning algorithms. The memory overhead for Hybrid-cut is only 1.87 percent even when it is configured to tolerate three machine failures simultaneously.

6.11 Efficiency of Fault-Tolerance Schemes

Readers might be interested in the theoretical efficiency of different fault-tolerance schemes with the optimal interval according to Young's model [30]. The efficiency is the minimum time (w/o fault-tolerance) divided by the expected time (w/fault-tolerance, including performance overhead and recovery cost for an expected number of failures). We assume the MTBF of a 50-node cluster is about 7.3 days [10]. For brevity, we use the results on PowerLyra for PageRank with Twitter as an example to estimate the efficiency of checkpoint-based (CKPT) and replication-based (REP)

TABLE 7
A Comparison of Total Memory Consumption (GB) on Our 50-Node Cluster Between Different Partitioning Algorithms for Tolerating 1, 2, and 3 Machine Failure(s) on PageRank with Twitter

	Random	Grid	Hybrid
w/o FT	223.8	136.7	173.1
FT/1	223.8 (+0.01%)	136.7 (+0.01%)	173.9 (+0.42%)
FT/2	223.9 (+0.04%)	136.8 (+0.08%)	175.0 (+1.08%)
FT/3	224.1 (+0.14%)	137.0 (+0.26%)	176.4 (+1.87%)

approaches. First, the optimal intervals of CKPT and REP are 9,768 and 623 s respectively, which match the significant difference of performance costs (75.63 and 0.31 s). Second, the efficiency of CKPT and REP are 98.44 and 99.90 percent respectively, which is relatively close since the failure rate is low. However, the negligible overhead and fast recovery of our replication-based approach are still of significant importance for graph processing since the execution time of an interval is usually short.

7 Related Work

Checkpoint-based fault tolerance is widely used in graph-parallel computation systems. Pregel [5] and its open-source clones [8], [9] adopt synchronous checkpoint to save the graph state to the persistent storage, including vertex and edge values, and incoming messages. GraphLab [10] designs an asynchronous alternative based on the Chandy-Lamport [14] snapshot to achieve fault tolerance. PowerGraph [11] and PowerLyra [13] provide both synchronous and asynchronous checkpointing for different modes. Yan et al. [46] propose an optimized lightweight checkpoint approach, which only stores vertex states and incremental edge updates to persistent storage. Shen et al. [47] combine checkpoint-based and log-based mechanisms to reduce the frequency of checkpointing and parallelize the recovery process. Zorro [40] also proposes a replication-based fault tolerance for distributed graph processing, but which trades off the accuracy of results while reducing the overhead during normal execution. Greft [48] is a distributed graph processing system that tolerates accidental arbitrary (e.g., Byzantine) faults.

Piccolo [49] is a data-centric distributed computation system, which provides user-assisted checkpoint mechanism to reduce runtime overhead. However, the user needs to save additional information for recovery. MapReduce [34] and other data-parallel models [50] adopt simple re-execution to recover tasks on crashed machines, since they suppose all tasks are deterministic and independent. Unfortunately, graph-parallel models do not satisfy such assumptions. Spark [35] and Discretized Streams [51] propose a fault-tolerant abstraction, namely Resilient Distributed Datasets (RDD), for coarse-grained operations on datasets, which only logs the transformation used to build a dataset (lineages) rather than the actual data. However, it is hard to apply RDD to graph-parallel models, since the computation on the vertex is a fine-grained update. Tiled MapReduce [52] backs up the results of sub-jobs to save the associated computation during a global failure. GraphX [12] claims to use in-memory distributed replication to reduce the amount of recomputation on failure, but it does not explain how to guarantee the replication and recover from machine failures.

Replication is widely used in large-scale distributed file systems [33], [53] and streaming systems [54], [55] to provide high availability and fault tolerance. In these systems, all replicas are full-time for fault tolerance, which may introduce high-performance cost. RAMCloud [17] is a DRAM-based storage system, it achieves a fast recovery from crashes by scattering its backup data across the entire cluster and harnessing all resources of the cluster to recover the crashed nodes. Distributed storage only provides simple abstraction and does not consider data dependency and computation on data. SPAR [15] is a graph-structured middleware to store social data. It also briefly mentions of storing more ghost vertices for fault tolerance. However, it does not consider the interaction among vertices, and only provides eventual consistency between master and replicas, which does not fit for graph-parallel computation systems.

8 Conclusion

This paper presented a replication-based approach called Imitator to provide low-overhead fault tolerance and fast crash recovery for large-scale graph-parallel systems with either edge-cut or vertex-cut. The key idea of Imitator is leveraging and extending an existing replication mechanism with additional mirrors and complete states as the master vertices, such that vertices in a crashed machine can be reconstructed using states from its mirrors. The evaluation showed that Imitator incurs very small normal execution overhead, and provides fast crash recovery from failures. Currently, we currently only consider graph-parallel computation; our future work will extend Imitator to support graph-structured stream processing and querying [56].

Acknowledgments

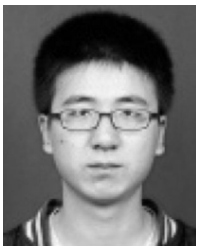
A preliminary version of this paper focused on graph-parallel systems using edge-cut is published in [57]. This work was supported in part by the National Key Research & Development Program (No. 2016YFB1000500), the National Natural Science Foundation of China (No. 61402284, 61572314, 61525204), the National Youth Top-notch Talent Support Program of China, the Zhangjiang Hi-Tech program (No. 201501-YP-B108-012) a research grant from the Singapore NRF (CREATE E2S2).

References

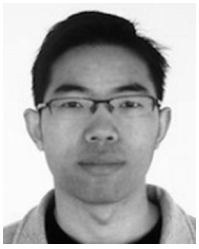
[1] S. Brin and L. Page, "The anatomy of a large-scale hypertextual Web search engine," in *Proc. 7th Int. Conf. World Wide*, 1998, pp. 107–117.
[2] J. E. Gonzalez, Y. Low, C. Guestrin, and D. O'Hallaron, "Distributed parallel inference on large factor graphs," in *Proc. 25th Conf. Uncertainty Artif. Intell.*, 2009, pp. 203–212.
[3] A. Smola and S. Narayanamurthy, "An architecture for parallel topic models," *Proc. VLDB Endowment*, vol. 3, pp. 703–710, 2010.
[4] R. Chen, M. Yang, X. Weng, B. Choi, B. He, and X. Li, "Improving large graph processing on partitioned graphs in the cloud," in *Proc. 3rd ACM Symp. Cloud Comput.*, 2012, Art. no. 3.
[5] G. Malewicz, et al., "Pregel: A system for large-scale graph processing," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 135–146.
[6] Q. V. Le, et al., "Building high-level features using large scale unsupervised learning," in *Proc. 29th Int. Conf. Mach. Learn.*, 2012, pp. 507–514.
[7] J. Dean, et al., "Large scale distributed deep networks," in *Proc. 25th Int. Conf. Neural Inf. Process. Syst.*, 2012, pp. 1232–1240.
[8] Apache Giraph. (2016). [Online]. Available: http://giraph.apache.org/
[9] Apache Hama. (2015). [Online]. Available: http://hama.apache.org/
[10] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed GraphLab: A framework for machine learning and data mining in the cloud," *Proc. VLDB Endowment*, vol. 5, pp. 716–727, 2012.
[11] J. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "PowerGraph: Distributed graph-parallel computation on natural graphs," in *Proc. 10th USENIX Conf. Operating Syst. Des. Implementation*, 2012, pp. 17–30.
[12] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica, "GraphX: Graph processing in a distributed dataflow framework," in *Proc. 11th USENIX Conf. Operating Syst. Des. Implementation*, 2014, pp. 599–613.

[13] R. Chen, J. Shi, Y. Chen, and H. Chen, "PowerLyra: Differentiated graph computation and partitioning on skewed graphs," in *Proc. 10th Eur. Conf. Comput. Syst.*, 2015, Art. no. 1.
[14] K. Chandy and L. Lamport, "Distributed snapshots: Determining global states of distributed systems," *ACM Trans. Comput. Syst.*, vol. 3, no. 1, pp. 63–75, 1985.
[15] J. Pujol, et al., "The little engine (s) that could: Scaling online social networks," in *Proc. ACM SIGCOMM Conf.*, 2010, pp. 375–386.
[16] J. Yan, G. Tan, Z. Mo, and N. Sun, "Graphine: Programming graph-parallel computation of large natural graphs for multicore clusters," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 6, pp. 1647–1659, Jun. 2016.
[17] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum, "Fast crash recovery in RAMCloud," in *Proc. 23rd ACM Symp. Operating Syst. Principles*, 2011, pp. 29–41.
[18] K. Schloegel, G. Karypis, and V. Kumar, "Parallel multilevel algorithms for multi-constraint graph partitioning (distinguished paper)," in *Proc. 6th Int. Euro-Par Conf. Parallel Process.*, 2000, pp. 296–310.
[19] I. Stanton and G. Kliot, "Streaming graph partitioning for large distributed graphs," in *Proc. 18th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2012, pp. 1222–1230.
[20] C. Tsourakakis, C. Gkantsidis, B. Radunovic, and M. Vojnovic, "FENNEL: Streaming graph partitioning for massive scale graphs," in *Pro. 7th ACM Int. Conf. Web Search Data Mining*, 2014, pp. 333–342.
[21] X. Zhu, W. Han, and W. Chen, "GridGraph: Large-scale graph processing on a single machine using 2-level hierarchical partitioning," in *Proc. USENIX Conf. USENIX Annu. Techn. Conf.*, 2015, pp. 375–386.
[22] D. Margo and M. Seltzer, "A scalable distributed graph partitioner," *Proc. VLDB Endowment*, vol. 8, pp. 1478–1489, 2015.
[23] N. Jain, G. Liao, and T. L. Willke, "GraphBuilder: Scalable graph ETL framework," in *Proc. 1st Int. Workshop Graph Data Manage. Experiences Syst.*, 2013, Art. no. 4.
[24] R. Chen, J. Shi, B. Zang, and H. Guan, "Bipartite-oriented distributed graph partitioning for big learning," in *Proc. 5th Asia-Pacific Workshop Syst.*, 2014, Art. no. 14.
[25] K. Zhang, R. Chen, and H. Chen, "NUMA-aware graph-structured analytics," in *Proc. 20th ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2015, pp. 183–193.
[26] R. Chen, X. Ding, P. Wang, H. Chen, B. Zang, and H. Guan, "Computation and communication efficient graph processing with distributed immutable view," in *Proc. 23rd Int. Symp. High-Performance Parallel Distrib. Comput.*, 2014, pp. 215–226.
[27] C. Xie, R. Chen, H. Guan, B. Zang, and H. Chen, "SYNC or ASYNC: Time to fuse for distributed graph-parallel computation," in *Proc. 20th ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2015, pp. 194–204.
[28] G. Wang, W. Xie, A. J. Demers, and J. Gehrke, "Asynchronous large-scale graph processing made easy," in *Proc. 6th Biennial Conf. Innovative Data Syst. Res.*, 2013, http://dblp.uni-trier.de/db/conf/cidr/cidr2013.html
[29] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan, "Group formation in large social networks: Membership, growth, and evolution," in *Proc. 12th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2006, pp. 44–54.
[30] J. W. Young, "A first order approximation to the optimum checkpoint interval," *Commun. ACM*, vol. 17, no. 9, pp. 530–531, 1974.
[31] K. Siddique, Z. Akhtar, Y. Kim, Y.-S. Jeong, and E. J. Yoon, "Investigating Apache Hama: A bulk synchronous parallel computing framework," *J. Supercomputing*, vol. 73, no 9, pp. 4190–4205, 2017.
[32] H. Haselgrove, "Wikipedia page-to-page link database," 2010. [Online]. Available: http://haselgrove.id.au/wikipedia.htm
[33] Hadoop distributed file system. (2017). [Online]. Available: http://hadoop.apache.org/
[34] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, 2008.
[35] M. Zaharia, et al., "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. 9th USENIX Conf. Netw. Syst. Des. Implementation*, 2012, pp. 2–2.
[36] J. Leskovec, K. Lang, A. Dasgupta, and M. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," *Internet Math.*, vol. 6, pp. 29–123, 2009.
[37] M. Castro and B. Liskov, "Practical Byzantine fault tolerance," in *Proc. 3rd Symp. Operating Syst. Des. Implementation*, 1999, pp. 173–186.
[38] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: Wait-free coordination for internet-scale systems," in *Proc. USENIX Conf. USENIX Annu. Tech. Conf.*, 2010, pp. 11–11.
[39] I. Hoque and I. Gupta, "LFGraph: Simple and fast distributed graph analytics," in *Proc. 1st ACM SIGOPS Conf. Timely Results Operating Syst.*, 2013, Art. no. 9.
[40] M. Pundir, L. M. Leslie, I. Gupta, and R. H. Campbell, "Zorro: Zero-cost reactive failure recovery in distributed graph processing," in *Proc. 6th ACM Symp. Cloud Comput.*, 2015, pp. 195–208.
[41] J. Xue, Z. Yang, Z. Qu, S. Hou, and Y. Dai, "Seraph: An efficient, low-cost system for concurrent graph processing," in *Proc. 23rd Int. Symp. High-Performance Parallel Distrib. Comput.*, 2014, pp. 227–238.
[42] C. Wilson, B. Boe, A. Sala, K. P. Puttaswamy, and B. Y. Zhao, "User interactions in social networks and their implications," in *Proc. 4th ACM Eur. Conf. Comput. Syst.*, 2009, pp. 205–218.
[43] F. Chierichetti, R. Kumar, S. Lattanzi, M. Mitzenmacher, A. Panconesi, and P. Raghavan, "On compressing social networks," in *Proc. 15th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2009, pp. 219–228.
[44] P. Boldi, B. Codenotti, M. Santini, and S. Vigna, "UbiCrawler: A scalable fully distributed web crawler," *Softw.: Practice Experience*, vol. 34, no. 8, pp. 711–726, 2004.
[45] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?" in *Proc. Int. Conf. World Wide*, 2010, pp. 591–600.
[46] D. Yan, J. Cheng, and F. Yang, "Lightweight fault tolerance in large-scale distributed graph processing," *arXiv:1601.06496*, 2016, http://arxiv.org/abs/1601.06496
[47] Y. Shen, G. Chen, H. V. Jagadish, W. Lu, B. C. Ooi, and B. M. Tudor, "Fast failure recovery in distributed graph processing systems," *Proc. VLDB Endowment*, vol. 8, pp. 437–448, 2014.
[48] D. Presser, L. C. Lung, and M. Correia, "Greft: Arbitrary fault-tolerant distributed graph processing," in *Proc. IEEE Int. Congr. Big Data*, 2015, pp. 452–459.
[49] R. Power and J. Li, "Piccolo: Building fast, distributed programs with partitioned tables," in *Proc. 9th USENIX Conf. Operating Syst. Des. Implementation*, 2010, pp. 1–14.
[50] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *Proc. 2nd ACM SIGOPS/EuroSys Eur. Conf. Comput. Syst.*, 2007, pp. 59–72.
[51] M. Zaharia, T. Das, H. Li, T. Hunter, S. Shenker, and I. Stoica, "Discretized streams: Fault-tolerant streaming computation at scale," in *Proc. 24th ACM Symp. Operating Syst. Principles*, 2013, pp. 423–438.
[52] R. Chen, H. Chen, and B. Zang, "Tiled-MapReduce: Optimizing resource usages of data-parallel applications on multicore with tiling," in *Proc. 19th Int. Conf. Parallel Archit. Compilation Techn.*, 2010, pp. 523–534.
[53] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *Proc. 19th ACM Symp. Operating Syst. Principles*, 2003, pp. 29–43.
[54] M. Balazinska, H. Balakrishnan, S. R. Madden, and M. Stonebraker, "Fault-tolerance in the borealis distributed stream processing system," *ACM Trans. Database Syst.*, vol. 33, no. 1, 2008, Art. no. 3.
[55] M. A. Shah, J. M. Hellerstein, and E. Brewer, "Highly available, fault-tolerant, parallel dataflows," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2004, pp. 827–838.
[56] J. Shi, Y. Yao, R. Chen, H. Chen, and F. Li, "Fast and concurrent RDF queries with RDMA-based distributed graph exploration," in *Proc. 12th USENIX Conf. Operating Syst. Des. Implementation*, 2016, pp. 317–332.
[57] P. Wang, K. Zhang, R. Chen, H. Chen, and H. Guan, "Replication-based fault-tolerance for large-scale graph processing," in *Proc. 44th Annu. IEEE/IFIP Int. Conf. Depend. Syst. Netw.*, 2014, pp. 562–573.

Rong Chen received the PhD degree in computer science from Fudan University, in 2011. He is currently an associate professor in the School of Software, Shanghai Jiao Tong University. His current research interests include large-scale graph processing, in-memory computing, and operating systems. He is a member of the IEEE.

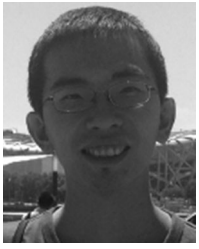


Youyang Yao is currently working toward the undergraduate student at Shanghai Jiao Tong University. His research interests include large-scale graph processing and graph query processing.



Peng Wang received the MS degree in computer science from Shanghai Jiao Tong University, in 2015.

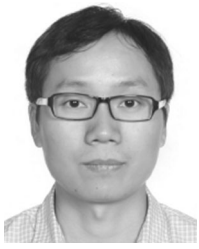
Kaiyuan Zhang received the BS degree in computer science from Shanghai Jiao Tong University, in 2015. He is currently a graduate student at the University of Washington.



Zhaoguo Wang received the PhD degree in computer science from Fudan University, in 2014. He is currently a research scientist in the Systems Group, New York University. His research interests include distributed systems, parallel and distributed processing.

Haibing Guan received the PhD degree from Tongji University, in 1999. He is a professor in the School of Electronic, Information and Electronic Engineering, Shanghai Jiao Tong University. His research interests include distributed computing, network security, network storage, green IT, and cloud computing.

Binyu Zang received the PhD degree in computer science from Fudan University, in 2000. He is currently a professor in the School of Software, Shanghai Jiao Tong University. His research interests include systems software, compiler design, and implementation.



Haibo Chen received the PhD degree in computer science from Fudan University, in 2009. He is currently a professor in the School of Software, Shanghai Jiao Tong University. His research interests include operating systems and parallel and distributed systems. He is a senior member of the IEEE.