Efficient and Deterministic Scheduling for Parallel State Machine Replication

Odorico M. Mendizabal*, Rudá S. T. De Moura†, Fernando Luís Dotti† and Fernando Pedone‡
*Universidade Federal do Rio Grande – FURG, Brazil
Email: odoricomendizabal@furg.br
†Pontifícia Universidade Católica do Rio Grande do Sul – PUCRS, Brazil
Email: ruda.moura@acad.pucrs.br, fernando.dotti@pucrs.br
‡Universitá della Svizzera italiana (USI), Switzerland
Email: fernando.pedone@usi.ch

***Abstract*—Many services used in large scale web applications should be able to tolerate faults without impacting their performance. State machine replication is a well-known approach to implementing fault-tolerant services, providing high availability and strong consistency. To boost the performance of state machine replication, recent proposals have introduced parallel execution of commands. In parallel state machine replication, incoming commands may or may not depend on other commands that are waiting for execution. Although dependent commands must be processed in the same relative order at every replica to avoid inconsistencies, independent commands can be executed in parallel and benefit from multi-core architectures. Since many application workloads are mostly composed of independent commands, these parallel models promise high throughput without sacrificing strong consistency. The efficient execution of commands in such environments, however, requires effective scheduling strategies. Existing approaches rely on dependency tracking based on pairwise comparison between commands, which introduces scheduling contention. In this paper, we propose a new and highly efficient scheduler for parallel state machine replication. Our scheduler considers batches of commands, instead of commands individually. Moreover, each batch of commands is augmented with a compact data structure that encodes commands information needed to the dependency analysis. We show, by means of experimental evaluation, that our technique outperforms schedulers for parallel state machine replication by a fairly large margin.**

***Keywords*-parallel state machine replication; fault tolerance; high throughput; deterministic scheduling**

I. Introduction

Several online services must be designed for both high availability and high throughput. State Machine Replication (SMR) [1], [2] is a well-known approach to increase service availability by tolerating replica failures. According to the SMR's execution model, every service replica receives and executes the same commands in the same order. Since replicas start with the same initial state and command execution must be deterministic, every replica will traverse the same sequence of states and produce the same output. Consequently, state machine replication ensures strong consistency (or more precisely, linearizability [3], [4]).

Many large online services use the SMR approach. Notable examples are Google's Chubby [5] and Apache Zookeeper [6]. Chubby is used by Borg [7], Google's cluster manager, Google File System (GFS) [8], and Bigtable [9], a distributed storage system. Apache Zookeeper is a popular service, offering a simple interface to support group messaging and distributed locking. It is used by HDFS [10], a Facebook file system, to implement a key-value store service, server replication, and concurrency control. Cassandra [11], a distributed data store, relies on Zookeeper for leader election and metadata management.

Although SMR provides configurable fault tolerance (i.e., by increasing the number of the replicas), it does not favor high throughput since the execution model is restricted to the sequential execution of commands. With servers typically running on multi-core architectures, the parallel execution of SMR commands has become an important research topic.

Since many application workloads are mostly composed of independent commands [12], command dependency tracking is a central aspect to both boost performance and keep strong consistency in SMR. In brief, commands are independent if they access disjoint portions of the replica's state or only read shared state and dependent otherwise. Dependent commands must be processed in the same relative order at every replica to avoid inconsistencies. Independent commands can be executed in parallel and benefit from multi-core servers.

Indeed, exploring command independency to allow parallel execution in SMR has recently received much attention (e.g., [13], [14], [15], [16], [17]). For instance, to parallelize the execution of independent commands, CBASE [13] adds a deterministic scheduler (also called parallelizer) to state machine replicas. In brief, the parallelizer at each replica receives commands in a total order (the same order at all replicas), examines command dependencies, and distributes the commands among a pool of worker threads for execution at the replica. The parallelizer handles a dependency graph to maintain a partial order across all pending commands. When a worker thread completes the execution of a command, it removes the command from the graph and responds to the client that submitted the command.

The design followed by CBASE deserves attention since it encapsulates dependency handling at the scheduling module. Furthermore, CBASE captures the exact conflict information needed to maximize concurrency and assure correct execution.

Under high load (i.e., possibly hundreds of thousands of commands per second), however, dependency tracking becomes itself a bottleneck as shown in this paper. To overcome this problem, we present a novel command handling and dependency tracking mechanism that favors high throughput in parallel state machine replication. Our proposed technique is based on the same design principles as CBASE: We encapsulate the complexity of command scheduling in the scheduling module. But differently from CBASE, we handle command batches instead of single commands at a time and introduce structures with low computational cost to aid in conflict detection at batch level. This leads our scheduler to outperform other schedulers by a fairly large margin. Since the low computational cost comes at the price of "false positives" (i.e., the detection of a conflict between two independent commands), we also investigate the impact of our techniques on throughput in the presence and absence of conflicts. Besides these contributions, we prove that our protocol guarantees linearizable executions and experimentally assess the performance of our scheduling technique using a full-fledged prototype comparing it to CBASE.

The rest of the paper is organized as follows. Section II presents the system model. Section III recalls the classical SMR approach, and Section IV discusses parallel approaches to SMR, focusing in CBASE [13]. Section V introduces the efficient PSMR protocol. Section VII experimentally assesses the performance of the proposed protocol. Section VIII surveys related work and Section IX concludes the paper.

II. System model

We assume a distributed system composed of interconnected processes. There is an unbounded set $C = \{c_1, c_2, ...\}$ of client processes and a bounded set $S = \{s_1, s_2, ..., s_n\}$ of server processes. The system is asynchronous: there is no bound on message delays and on relative process speeds. We assume the crash failure model and exclude malicious and arbitrary behavior (e.g., no Byzantine failures). A process is *correct* if it does not crash or *faulty* otherwise. We assume f faulty servers, out of $n = 2f + 1$ servers, i.e., f is the maximum number of server failures that can be tolerated.

Processes communicate by message passing, using one-to-one or one-to-many communication. One-to-one communication is through primitives send(m) and receive(m), where m is a message. If a sender sends a message enough times, a correct receiver will eventually receive the message. One-to-many communication is based on atomic broadcast, whose main primitives are *broadcast*(m) and *deliver*(i, m), where i refers to the consensus instance in which message m was decided. This definition implicitly assumes that atomic broadcast is implemented with a sequence of consensus instances identified by natural numbers $1, 2, 3, ...$ (e.g., [18], [19]).

Atomic broadcast ensures that (i) if a process broadcasts message m and does not fail, then there is some i such that eventually every correct process delivers (i, m); and if a process delivers (i, m), then (ii) all correct processes deliver (i, m), (iii) no process delivers (i, m') for $m \neq m'$, and

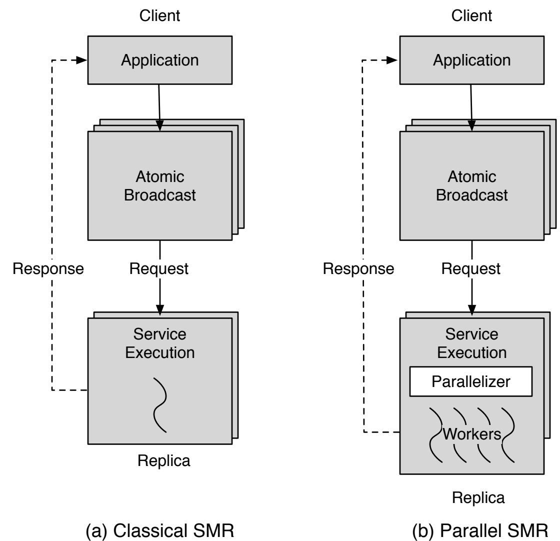

Fig. 1. Classical versus parallel state machine replication

(iv) some process broadcast m. We implement atomic broadcast using Paxos [19], a consensus protocol. Paxos requires additional synchronous assumptions but our protocols do not explicitly need these assumptions.

III. Classical state machine replication

State Machine Replication (SMR) renders a service fault-tolerant by replicating the server and coordinating the execution of client commands among the replicas [1], [2]. The service is defined by a state machine and consists of *state variables* that encode the state machine's state and a set of *commands* that change the state (i.e., the input). The execution of a command may (i) read state variables, (ii) modify state variables, and (iii) produce a response for the command (i.e., the output). Commands are *deterministic*: the changes to the state and response of a command are a function of the state variables the command reads and the command itself.

SMR provides clients with the abstraction of a highly available service while hiding the existence of multiple replicas. This last aspect is captured by *linearizability*, a consistency criterion: a system is linearizable if there is a way to reorder the client commands in a sequence that (i) respects the semantics of the commands, as defined in their sequential specifications, and (ii) respects the real-time ordering of commands across all clients [3], [4]. In classical SMR, linearizability can be achieved by having clients atomically broadcast commands and replicas execute commands sequentially in the same order (see Figure 1(a)). Since commands are deterministic, replicas will produce the same state changes and response after the execution of the same sequence of commands.

IV. Parallel state machine replication

Classical SMR makes poor use of multi-processor architectures since deterministic execution normally translates into (single-processor) sequential execution of commands.

Although (multi-processor) concurrent command execution may result in non-determinism, it has been observed that "independent commands" (i.e., those that are neither directly nor indirectly dependent) can be executed concurrently without violating consistency [2]. Two commands are *independent* if they either access different variables or only read variables commonly accessed; conversely, two commands are *dependent* if they access one common variable v and at least one of the commands changes the value of v. For example, two read commands are independent, while a read and an update command on the same variable are dependent.

A few approaches have been suggested in the literature to execute independent commands concurrently with the goal of improving performance (e.g., [13], [14], [20]). In this section, we describe CBASE, the approach proposed in [13] and the motivation for the techniques proposed in this paper. We recall other approaches to parallel SMR in Section VIII.

To parallelize the execution of independent commands, CBASE adds a deterministic scheduler, also known as parallelizer, to each replica (see Figure 1(b)). Clients atomically broadcast commands and the parallelizer at each replica delivers commands in total order, examines command dependencies, and distributes them among a pool of worker threads for execution. The parallelizer uses a dependency graph to maintain a partial order across all pending commands, where vertices represent commands and directed edges represent dependencies. While dependent commands are ordered according to their delivery order, independent commands are not directly connected in the graph. Worker threads receive independent commands from the parallelizer (i.e., vertices with no incoming edges) to be concurrently executed. When a worker thread completes the execution of a command, it removes the command from the graph and responds to the client that submitted the command.

Figure 2(a) depicts an illustrative dependency graph with six commands, delivered in the order $a, b, ..., f$. Commands a, c and e are the next ones to be scheduled for execution and can execute concurrently. Commands a and b are dependent but a was delivered first; so, a must execute before b. Intuitively, fewer interdependencies between commands in the dependency graph favor concurrency. However, the cost of adding a new command in the dependency graph is proportional to the number of commands in the graph that are independent of the new command. For example, a new command g will be first compared to commands d and f; if g is independent of d, it will be compared to c and b, and so on. If g is independent of every command in the graph, it will be compared against all vertices.

In the context of high-throughput state machine replication, the overhead of detecting conflicts between each new command against the dependency graph turns out to significantly hinder performance, a fact that we show experimentally in Section VII. In Section V, we describe more efficient ways to handle command dependencies.

V. Efficient Parallel State Machine Replication

In this section, we introduce a deterministic scheduler to efficiently handle command dependencies and schedule independent commands to execute concurrently. The main goal of this scheduler is to reduce the costs on handling a dependency graph.

A. Overall idea

In summary, our scheme combines the following strategies:

- The scheduler handles a batch of commands at a time, instead of one command at a time. The dependency graph stores batches of commands instead of single commands.
- Batch conflict detection uses an efficient mechanism allowing a single comparison to evaluate conflict among two batches.
- Worker threads execute entire batches. Commands in the same batch are executed sequentially, in the given order.

Batched commands. Clients submit commands through a client proxy, which groups commands from different clients and broadcasts the commands for execution as batches. When the proxy receives responses for all commands in a batch, it can submit a new batch of commands. There can be any number of client proxies, each one handling a group of clients.

The abridged dependency graph. The scheduler receives batches of commands and builds an abridged dependency graph, where vertices are command batches and edges are dependencies induced by the commands in the batches. More precisely, there is an edge from batch B_i to B_j in the graph if B_i is received before B_j and B_j contains at least one command that depends on a command in B_i (see Figure 2(b) and (c)).

Batch conflict detection. Detecting conflicts between any two batches B_i and B_j boils down to detecting a conflict between any $c_i \in B_i$ and any $c_j \in B_j$. In the case of no conflict this requires $O(B^2)$ conflict detection operations, where B is the batch size. Batch conflict detection succeeds when the first command conflict is found. Handling batches of commands is more efficient than processing one command at a time, but still requires a large number of conflict detection operations. In the following we show how to reduce this overhead.

Efficient batch conflict detection. We extend each batch of commands with a bitmap with a digest of the variables read and written by the commands in the batch. The idea is that given two bitmaps, $b(B_i)$ and $b(B_j)$, we want to be able to determine whether there is a command in batch B_i that conflicts with some command in batch B_j by comparing their bitmaps. The way bitmaps are encoded to satisfy this property is application dependent and can be achieved in different ways. In our prototype, we consider write commands in a database, where each operation includes the key of the entry written in the database. We create bitmaps by hashing the key provided in the command; the hashed value corresponds to a bit set in the bitmap. Checking whether two batches contain conflicting commands boils down to a bit-wise comparison of their bitmaps.

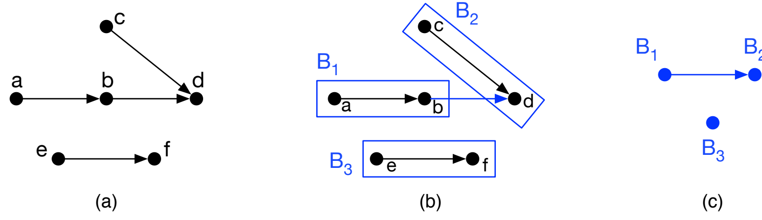

Fig. 2. Three representations of a dependency graph with six commands. (a) The original dependency graph, where edge $x \rightarrow y$ means that commands x and y are dependent and x was delivered before y. (b) The original graph grouped in batches of two commands. (c) The abridged dependency graph; notice that commands b and c are serialized in the abridged graph.

The overhead versus concurrency tradeoff, part 1. The abridged dependency graph establishes a tradeoff. On the one hand, batching reduces the overhead needed to handle commands (e.g., fewer system calls to deliver commands, fewer edges to store the graph, fewer comparisons to determine dependencies), which improves performance. On the other hand, the abridged dependency graph reduces concurrency since it may induce dependencies among independent commands. In Figure 2(b), independent commands a and c are serialized since commands in B_1 must execute before commands in B_2 because b is in B_1, d is in B_2 and b precedes d.

The overhead versus concurrency tradeoff, part 2. Besides the serialization imposed by batching above discussed, the need for fast scheduling decisions introduces a second source of serialization: since a bitmap encodes hashes of all command keys in the batch and bitmaps have limited size, the possibility of distinct keys mapping to the same bit exists. This leads to false positives in the detection of conflict between batches, sequentializing their execution. The false positive rate depends on the size of the bitmaps and the space of keys, as we discuss in Section VII. False positives may harm performance but not safety since conflicting batches obey the total order agreed among replicas. To keep safety, the encoding scheme should not generate false negatives (i.e., indicate no conflict when one exists). Our hashing scheme prevents false negatives.

The stored batch dependency graph. The batch dependency graph is stored in each replica, keeping complete dependency information among batches. A dependency graph among pending batches is $DG = (B, E)$ where:

- B is a set of batches.
- $E \in B \times B$ is a set of edges such that iff $(B_i, B_j) \in E$ then $b(B_i) \cap b(B_j) \neq \emptyset$, where $b(B_i)$ is the bitmap of B_i, and B_i was delivered before B_j. This means that B_i has to be executed before B_j.

The execution of commands. Each worker thread t_i executes commands following their order in DG, where commands in the same batch are handled in the order they appear in the batch. A batch B_i can be executed if $\nexists B_j$ such that $(B_j, B_i) \in E$. Since a batch under execution has to be taken into account for conflicts with new arriving batches, the worker thread does not exclude the batch under execution from the graph, but instead marks it as being processed such that no other thread takes the same batch. Therefore, batches have to be extended with the notion of $status(B_i) \in \{taken, notTaken\}$. Once the thread finishes the whole batch then it is removed from the graph along with the outgoing edges, which may lead to further batches free for execution. As a consequence of this procedure, the scheduler and the worker threads must access DG in mutual exclusion.

B. Algorithm in detail

Algorithm 1 details the behavior of the scheduler and the worker threads, respectively. The dependency graph DG (line 2) is accessed in mutual exclusion[1] by the scheduler and worker threads t_i. The initialization procedure (line 6) sets the initial values for the dependency graph, the number of threads and the next batch to be delivered. When a batch is delivered (line 14) it is inserted in the dependency graph. The insert operation includes all dependencies from existing batches in the graph (lines 18 to 20). In lines 28 to 29 and 30 to 31 two methods for batch conflict detection are described.

A worker thread (lines 44 to 46) requests a batch to execute using $dgGetBatch()$, processes it, and removes it from the dependency graph using $dgRemoveBatch()$. $dgGetBatch()$ finds the oldest among the free batches for execution and not taken by another thread (line 33) if there is one. In such a case, the batch is marked as taken (line 35). $dgRemoveBatch()$ removes dependencies between the executed batch and other batches (lines 39 to 41) and then removes the executed batch (line 42).

Why it works. Correctness is discussed in detail in the Appendix. Here we highlight the main intuition.

Replica consistency: conflicting batches are processed according to the total order. This holds since:

i) Safety I (preserving total order for conflicting batches in DG): all batches are inserted into the dependency graph in the order $<_B$ they arrive (lines 14 to 15). When a batch B_i is inserted, its conflict with all previously stored batches is calculated and edges meaning dependencies included in E from all the conflicting ones (lines 18 to 20). With this E encodes a directed acyclic graph (DAG) from $<_B$ with edges between conflicting batches.

ii) Safety II (preserving total order for conflicting batches

[1]For the sake of simplicity, we do not show the synchronization primitives for protection of DG. The reader can just assume that inserting, getting the next batch, and removing a batch are performed in mutual exclusion. Our prototype implements mutual exclusion with monitors.

Algorithm 1

1: **data structures and variables**
2: $DG = (B, E)$ {*the shared dependency graph*}
3: $int\ k$ {*the batch instance to be delivered*}
4: $int\ N$ {*the number of worker threads*}
5: $boolean\ useBitmap$ {*define conflict detection mechanism*}

6: **procedure** Initialization()
7: $N \leftarrow$ desired number of worker threads
8: $useBitmap \in \{True, False\}$
9: $DG \leftarrow (\emptyset, \emptyset)$
10: $k \leftarrow 1$
11: **for** $id = 1..N$ **do** {*for each worker thread...*}
12: create worker thread t_{id}

13: *The scheduler executes as follows:*
14: **when** deliver(k, B_k) {*when deliver new batch of commands*}
15: dgInsertBatch(B_k) {*schedule the delivered batch and*}
16: $k \leftarrow k + 1$ {*get ready to deliver the next batch*}

17: **procedure** dgInsertBatch(B_i)
18: $\forall B_j \in B$ {*any batch in the graph that*}
19: **if** $conflict(B_i, B_j)$ **then** {*... conflicts with incoming one*}
20: $E \leftarrow E \cup \{(B_j, B_i)\}$ {*has to be processed before*}
21: $status(B_i) \leftarrow notTaken$ {*no one is processing this batch*}
22: $B \leftarrow B \cup \{B_i\}$ {*insert batch in the graph*}

23: **procedure** boolean : conflict(B_i, B_j)
24: **if** $useBitmap$ **then** {*choose conflict mechanism*}
25: return bitmapConflict(B_i, B_j)
26: **else**
27: return cmmdKeyConflict(B_i, B_j)

28: **procedure** boolean : bitmapConflict(B_i, B_j)
29: **return** $b(B_i) \cap b(B_j) \neq \emptyset$

30: **procedure** boolean : cmmdKeyConflict(B_i, B_j)
31: **return** $\exists\ c_i \in B_i,\ c_j \in B_j$ such that $key(c_i) = key(c_j)$ {*searches two conflicting commands in the batches*}

32: **procedure** batch : dgGetBatch()
33: **let** freeBatches $= \{B_i \in B | \forall B_j \in B, (B_j, B_i) \notin E$
34: $\wedge status(B_i) = notTaken\}$ {*all free batches not yet taken*}
35: **let** $B_k \in$freeBatches$|\forall B_l \in B, k < l$ {*get the oldest free batch*}
36: $status(B_k) \leftarrow taken$ {*no other thread take it*}
37: $returnB_k$

38: **procedure** dgRemoveBatch(B_i)
39: $\forall B_j \in B$ {*any batch in the graph that*}
40: **if** $(B_i, B_j) \in E$ **then** {*... depends on the one being removed*}
41: $E \leftarrow E \setminus \{(B_i, B_j)\}$ {*does not depend anymore*}
42: $B \leftarrow B \setminus \{B_i\}$ {*remove processed batch from graph*}

43: *Each worker thread t_{id} executes as follows:*
44: **while** $B_i \leftarrow dgGetBatch()$ {*while there are commands to execute*}
45: execute commands in B_i in their order
46: $dgRemoveBatch(B_i)$

when taking a batch from DG): a batch is only taken to be processed when it has no incoming dependency edges in E (lines 33 to 35), and it is free to execute. Since each edge in E denotes a dependency, no batch is taken disrespecting the total order of conflicting batches.

iii) Non-deadlock: the first batch does not depend on any other batch and is free to execute. An executed batch B_i is removed and all edges in E from B_i as well (lines 39 to 42). Since a batch only depends on preceding ones, then a B_j, $B_i <_B B_j$ will necessarily have its last incoming dependency edge removed with the removal of B_i and will be free to execute: the graph has always a lowest element.

Consistency across replicas. This is granted since replicas deliver the same total order ($<_B$), implement the same algorithm, and this algorithm preserves the total order of conflicting batches irrespective of the relative speed of different replicas, which is argued as follows. Replicas may progress at different speeds and thus may have different sets of pending batches in their dependency graphs. When a batch B_j is delivered according to $<_B$ in two replicas R_a and R_b, B_i may belong to pending batches in R_a but not in R_b. In such case, R_b processes them in total order, while R_a will process in total order in case they conflict, and concurrently otherwise. In any case the total order of conflicting batches is preserved.

VI. Implementation

In order to evaluate our scheduling technique, we developed a key-value store service using CBASE, and the same service using our efficient version of CBASE scheduling, which includes batches and bitmaps. The service implements commands to create, read, update and remove keys from an in-memory database. Atomic broadcast is provided by the primitives broadcast and deliver implemented in Ring Paxos [21], a high-throughput atomic broadcast protocol.[2]

Client commands are forwarded to a client proxy, which is responsible for batching those commands in a single request. To alleviate the burden on the parallelizer, the bitmaps for a batch are computed by the client proxy. Client proxies broadcast a request to the replicas and wait for the first reply from a replica for every command in the batch before broadcasting another batch.

Upon receipt of a batch, a service replica proceeds as in Algorithm 1. To represent the original CBASE, the prototype is configured with batches of size 1, i.e., every batch contains a single command. Furthermore, two approaches are available to determine dependencies. The first one (lines 30 to 31) compares each command key in the incoming batch against keys of commands in the batches stored in the dependency graph. The second approach (lines 30 to 31) compares the bitmap of the incoming batch against bitmaps of batches stored in the dependency graph.

A. Graph implementation

Here we describe the graph implementation approach used. Each batch is a node of the graph. We say that a node n_i depends of another node n_j if their respective batches conflict and B_i is delivered after B_j. The dependency graph among nodes is implemented as an ordered list of nodes $nodeList$. Each node is inserted to $nodeList$ as the batch is delivered according to $<_B$, the total delivery order of batches. Each $node$ is a tuple $\langle batch, deps, bDeps, status\rangle$, where:

- $batch \in B$;

[2]We used the URingPaxos library (https://github.com/sambenz/URingPaxos).

- $deps$ is a set of nodes that depend on this node, it represents the edges E of the dependency graph;
- $bDeps$ (backward dependency) is a set of nodes this node depends of, used for implementation purposes;
- $status \in \{taken, notTaken\}$;

This graph has as atomic operations dgInsertBatch(B_i), dgGetBatch() and dgRemoveBatch(B_i) as described in Algorithm 1. Edges E from a node are represented by the node's $deps$. $bDeps$ is used to speed the process of removing edges from $deps$ of nodes that depend on the one being removed, further freeing nodes. Status is used as in Algorithm 1.

B. Bitmap implementation

Bitmaps are used to encode the keys of every command batched in a request. A hash function maps the keys into bitmap positions. Figure 3 illustrates a batch B and its respective bitmap representation. Batch B contains 2 commands and commands' keys are mapped into a bitmap of size m, $b(B)$. The second and the last but one $b(B)$ bits are set to 1, indicating that keys x and y belong to this bitmap.

The number of bits in a bitmap does not necessarily represents the number n of commands encoded in that batch. Since any two commands may access the same key, the same bit can map more than one command key. In addition, depending on the size of n and m, there is a certain probability that the hash function maps two different keys to the same bitmap position. This probability increases as n increases and m decreases. While this approach is subject to false positives (i.e., it may detect a conflict when none exists), it is not prone to false negatives (i.e., it does not miss real conflicts).

The bitmap structure is a Bloom filter with a single hash function for mapping keys into a bit array. While general purpose Bloom filters can be set with more than one hash function, for the dependency analysis proposed in this paper, the number of hash functions is limited to one. This restriction is necessary because our approach does not query Bloom filters for a given element. Instead, a conflict is detected when a non-empty intersection between the encoding information by a pair of Bloom filters is found, i.e., the same position set to 1 in both bit arrays. (Note that the intersection of bit arrays created with more than one hash function would increase the false positive rate.)

VII. Performance evaluation

In this section, we explain our assessment goals and methodology, describe the experiment environment, and present the results of our performance study.

A. Goals and methodology

The scheduling technique introduced in this paper aims to speed up SMR execution with minimal overhead. We wish to quantify the effects of enhancing CBASE with batches and bitmaps, and compare to the traditional CBASE on the following aspects:

- speed-up achieved with growing number of worker threads and increasing batch sizes;

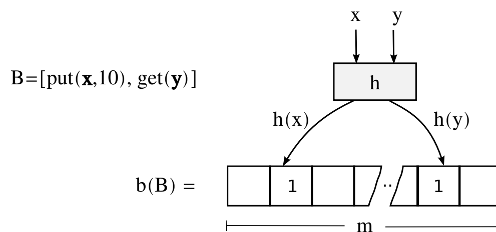

Fig. 3. Resulting bitmap for a batch of commands. The bitmap size is m and it encodes 2 keys.

- the impact of the scheduling overhead;
- the impact of false positives introduced by the dependency analysis based on bitmaps;
- the impact of conflicts in the overall throughput.

To investigate the first aspect, since we are interested in the effect of concurrent command execution in SMR, we provoke the maximum concurrency possible using conflict-free workloads and evaluate the throughput achieved with varying number of worker threads and batch size.

To investigate the second aspect, we vary the computational demand of request processing. Light request processing would show more clearly the impact of scheduling overhead while heavy request processing would dilute this overhead. We induce heavy request processing by increasing the batch size.

To investigate the third aspect, we vary both batch and bitmap size and measure the false positive rate for a mix of scenarios.

Finally, to investigate the fourth aspect, we vary the conflict rate and observe how the increasing of dependent batches in the graph impact in the overall throughput. The resulting experiments combine a coverage of these variables.

B. Environment and configuration

All experiments were executed on a cluster with two types of nodes: HP SE1102 nodes equipped with two quad-core Intel Xeon L5420 processors running at 2.5 GHz and 8 GB of main memory; and Dell PowerEdge R815 nodes equipped with four 16-core AMD Opteron 6366HE processors running at 1,8 GHz and 128 GB of main memory. The HP nodes were connected to an HP ProCurve switch 2920–48G gigabit network switch, and the Dell nodes were connected to another, identical network switch. The switches were interconnected by a 20 Gbps link. All nodes ran CentOS Linux 6.5 and had the Oracle Java SE Runtime Environment 8. Paxos' proposer, acceptors, and clients were deployed on HP nodes, while service replicas were deployed on Dell nodes.

C. Speed-up analysis

In this section we evaluate the scalability of the proposed approach with the increasing in the number of worker threads. We defined workloads with different batch sizes, varying from 1, 100, and 200 commands per batch. Furthermore, to concentrate on the impact caused by the scheduling overhead, in this experiment, we exercise contention-free workloads (i.e., without conflicts).

Figure 4 depicts the throughput of our key-value store prototype (given in kilo commands per second) for different CBASE configurations. Test scenarios vary according to the number of worker threads, batch size, and dependency analysis strategy (by comparing every key in a batch, or comparing the batch bitmaps). The traditional CBASE is represented by the first group of clustered bars (CBASE, batch size = 1). The maximum throughput observed is around 33000 commands per second. It is also observed that service does not scale well, i.e., the throughput is practically the same regardless of the number of worker threads (see 1, 2, 4, 8, and 16 threads' bars). Therefore, when the cost for processing a single command is very low, the scheduler becomes the bottleneck in the traditional CBASE.

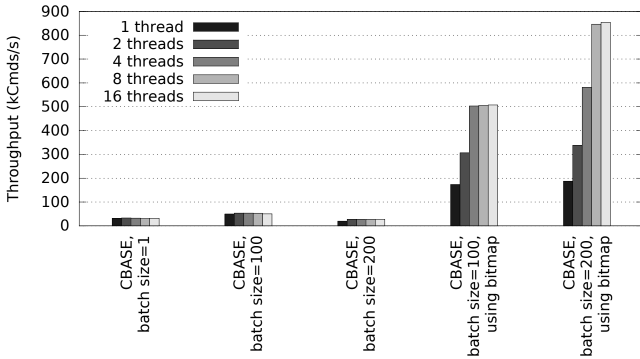

Fig. 4. Threads scalability for contention-free workloads.

In order to reduce the scheduling overhead impact, worker threads should spend more time processing commands or the scheduler technique should be optimized. Our approach covers these two aspects. While batching makes the task of executing commands heavier, the use of bitmaps in the dependency analysis reduces the number of comparisons needed to detect dependencies.

In Figure 4, the maximum throughput reaches 53000 commands/s when commands are grouped in batches of size 100 (see "CBASE, batch size = 100" bars). This represents an increase of 1.6 times on the throughput when compared to the traditional CBASE. However, when commands are grouped in batches of size 200 (see "CBASE, batch size = 200" bars), the throughput reaches 27600 commands/s, approximately 0.84 times the traditional throughput. The decrease on performance with batches of size 200 is explained by the large number of comparisons per dependency checking.

Grouping commands into batches has demonstrated limited scalability when dependencies are calculated key-by-key (see Figure 4, "CBASE, batch size = 100", and "CBASE, batch size = 200"). This lack of scalability is explained by the graph contention caused by a large number of comparisons performed during dependency analysis. When bitmaps are used to determine dependencies among batches, the synchronization cost caused by the scheduler is dramatically reduced. Furthermore, incoming batches become available for threads processing more quickly. Figure 4 shows a maximum throughput of 507000 commands/s for scenario "CBASE, batch size = 100, using bitmap", and 854000 commands/s for "CBASE, batch size = 200, using bitmap", i.e., throughput is 15.4 and 25.9 times higher than the traditional CBASE. It is also observed a better scalability when bitmaps are used, specially for scenarios with larger batches ("CBASE, batch size = 200, using bitmap").

D. *Conflict rate analysis*

As presented in Section V, the keys of all the commands belonging to a batch are encoded in a single bitmap associated to that batch. While comparing two bitmaps, the coincidence of at least one common bitmap position set as 1 in both bitmaps configures a conflict among batches (i.e., both bitmaps encoded the same command key). Notice that different keys may coincidentally be mapped to the same position, thus false positives are possible in our approach.

We evaluate the conflict rate produced by our approach by means of simulation. Our simulator represents incoming requests as single batches, and the dependency graph as a list of batches (each batch in the list corresponds to a vertex in the graph). The average graph size is given by the list size. To determine conflicts, our simulator compares an incoming batch against the list of batches (i.e., a representation of the dependency graph). If at least one common bitmap position is set as 1 in both bitmaps, then a conflict is computed. After checking conflicts involving the incoming batch and the list of batches, the incoming batch is added to the list of bitmaps and the oldest batch in the list is removed.

In our experiments we configure the bitmap size (in number of bits), the batch size (i.e., the number of keys encoded in a batch), the average graph size, the number of distinct keys, and the number of iterations. For all experiments, we adopted a large number of distinct keys (10^9), so the probability of two identical keys are encoded in a pair of bitmaps under comparison is very low. This means that conflicts detected in the simulation are predominantly caused by false positives. We also fixed the number of iterations to 10^6 for all executions. We choose values for batch size and average graph in accordance with the experiments performed in Section VII-C. From previous experiments, batch sizes were set to 100 and 200, so we adopt these values in our simulation. The average graph size computed throughout those scenarios execution were: 1 for "CBASE, batch size=1", 1 for "CBASE, batch size=100", 1 for "CBASE, batch size=200", 5 for "CBASE, batch size=100, using bitmap", and 7 for "CBASE, batch size=200, using bitmap". Thus, we set up the average graph size to 1, 5, and 7 in our simulations.

Table I shows how the conflict rate varies with the bitmap size, batch size, and the average graph size. The use of larger batches causes more conflicts since more commands are encoded and leading to a higher probability of a single conflict occurrence. The use of larger bitmaps reduces the occurrence of false positive, what can be observed by the reduction on the conflict rate as the bitmap size increases. For example, when

the bitmap size is equal to 1 Mbit, "CBASE, batch size=100" has average graph size of 5 batches and according to Table I around 4.75% conflict, while "CBASE, batch size=200" has average graph size of 7 batches and around 23.95% conflict.

TABLE I
CONFLICT RATE

Bitmap size (bits)	Average graph size	Conflict rate (batch size = 100)	Conflict rate (batch size = 200)
102400	1	9.29%	32.37%
102400	5	38.69%	85.85%
102400	7	49.50%	93.52%
1024000	1	0.96%	3.85%
1024000	5	4.75%	17.78%
1024000	7	6.61%	23.95%

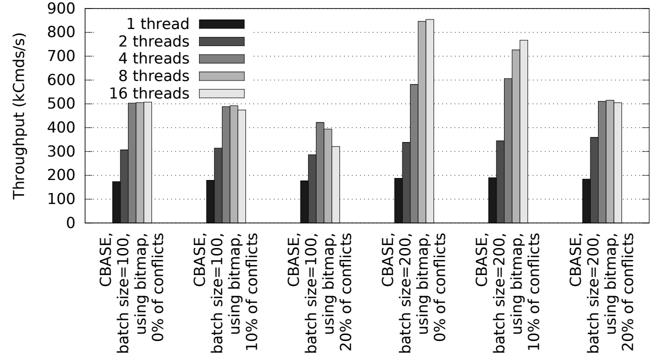

Fig. 5. Impact of conflicts on overall throughput.

E. Speed-up analysis for conflict-prone workloads

Once we surveyed the performance of our prototype for some scenarios of interest in context-free workloads, we introduce conflicts in our analysis. According to Moraru et al. [12], dependency probabilities between 0% and 2% are the most realistic. For instance, in Chubby, for traces with 10 minutes of observation, fewer than 1% of all commands could possibly generate conflicts [5]. In Google's advertising back-end, F1, fewer than 0.3% of all operations may generate conflicts [22]. Although the literature suggests a very low conflict rate, our dependency analysis strategy introduces false conflicts. For this reason, we re-run previous scenarios with conflict rates compatible to those discussed in Section VII-D.

Figure 5 shows the maximum throughput for scenarios "CBASE, batch size=100, using bitmap", and "CBASE, batch size=200, using bitmap" in the conflict-free, 10% of conflicts, and 20% of conflicts workloads. We choose 10% and 20% of conflicts because these rates are similar to those experienced when bitmap size is 1 Mbit. As expected, as the conflict rate increases, the lower is the throughput. Performance depends on the amount of concurrency allowed by the workload (related to the percentage of conflicts) and the overhead to synchronize threads. With a low number of worker threads (e.g., 1, 2, and 4) and 20% of conflicts, there are enough requests to keep threads busy. As the number of threads increases, the synchronization overhead outweighs the work that there is for these threads to execute. So, performance decreases slightly.

It is worth noting that even with the increase on the number of conflicts, our technique outperforms CBASE by a fairly large margin. For instance, with 20% of conflicts, the maximum throughput for "CBASE, batch size=200, using bitmap" is around 515000 commands/s, i.e., 15 times higher than CBASE throughput. Thus, although bitmaps detect false positives while calculating dependencies, the time taken by the dependency analysis is significantly reduced.

VIII. RELATED WORK

In this section, we review existing approaches to parallel SMR. To the best of our knowledge, CBASE [13] was the first proposal for parallel execution of SMR requests. By using an atomic broadcast protocol, all requests are delivered in the same order to every replica, exactly as in the classic SMR. In CBASE, however, replicas are augmented with a deterministic scheduler, as known as parallelizer. Based on application semantics, the parallelizer serializes the execution of dependent commands according to the delivery order and dispatches independent commands to be processed in parallel by a pool of worker threads. The consistency is provided by each individual replica, once requests are delivered in the same order and parallelizer adopts the same scheduling policies to establish a partial order for requests processing.

In Eve [14], authors propose a speculative strategy with the aim of avoiding the scheduling overhead. Eve replicas first execute commands and then verify the equality of their states through a verification stage. Before execution, a primary replica groups client commands into batches and transmits the batched commands to all replicas. Then, replicas speculatively execute batched commands in parallel. After the execution of a batch, the verification stage checks the validity of replica's state, as defined by the common state reached by a majority. If too many replicas diverge, replicas roll back to the last verified state and re-execute the commands sequentially. To avoid costly rollback procedures, the frequency in which replicas need to reconcile must be reduced. Eve minimizes divergence through a mixer stage that applies application-specific criteria to produce groups of requests that are unlikely to interfere.

In Rex [23], a single server receives requests and processes them in parallel. While executing, the server logs a trace of dependencies among requests based on the shared variables locked by each request. The server periodically proposes the trace for agreement to the pool of replicas. The other replicas receive the traces and replay the execution respecting the partial order of commands. The Execute-agree-follow model proposed by Rex resembles the passive replication model.

In [16], authors present Storyboard, an approach that supports deterministic execution in multi-threading environments. Their strategy enhances SMR with a forecasting mechanism that, based on application-specific knowledge, predicts the same ordered sequence of locks across replicas. While forecasts are correct, commands can be executed in parallel. If the forecast made by the predictor does not match the

execution path of a command, then the replica have to establish a deterministic execution order in cooperation with other replicas. In this case, Storyboard blocks the execution of the command and repredicts the command's execution path. The *repredict* command runs a consensus protocol to determine a consistent point in the execution order across all replicas. The reprediction will contain at least the lock which the command currently seeks to acquire, but possibly also further locks. All replicas will proceed according to the new forecast.

CRANE [17] is a parallel SMR system that transparently replicates general multi-threaded programs. Within each replica, CRANE intercepts POSIX socket and the Pthreads synchronization interface and implement deterministic versions of such synchronizing operations. To ensure total order delivery of synchronization commands across replicas, for each incoming socket call (e.g., `accept()` or `recv()`), CRANE runs a distributed consensus protocol, so that correct replicas see exactly the same sequence of calls. CRANE schedules synchronization commands using deterministic multithreading (DMT) [24], [25]. This technique maintains a logical time that advances deterministically on each thread's synchronization. The central idea of CRANE is to combine the input determinism of Paxos and the execution determinism of DMT.

In [15], authors propose a parallel SMR approach that uses multiple multicast groups to partially order commands across replicas, where each group leads to a different stream of commands delivered at each replica. In this approach the independent commands are not delivered by a single component and then scheduled for parallel execution. Instead, independent commands can be directly delivered by multiple worker threads by mapping command streams to multiple sockets. Therefore, the overhead associated with a parallelizer mechanism is minimized by this approach.

Our scheduler follows the same principle design of the parallelizer proposed in [13] and the mixer adopted in [14]. Compared to these works, our scheduler reduces the dependency graph contention, achieving higher throughput. Specially when compared to [14], our approach avoids the need of a verification and roll-back phases, which can be very costly (depending on the workload). Rex [23] also achieves very high throughput. However, Rex implements a passive replication strategy, where the total order delivery is no longer required. Thus, in case of primary failure, it suffers from high reconfiguration costs, and requests may be delayed during reconfiguration. In [16], [15] authors propose different architectural styles to enforce parallel execution of commands. For instance, although [15] may significantly improve the scalability of the delivery protocol, system designers should be able to partition SMR application to enjoy full benefits of the technique. Our approach offers a simple design, since it encapsulates the dependency handling complexity at a single scheduling module, keeping untouched other modules. In terms of design simplicity, CRANE [17] is possibly the best choice. Its drawback is that multithreaded applications with intense synchronization incur higher overhead due to the DMT approach. In our approach whenever the number of non-conflicting requests stored in the dependency graph is close to the number of worker threads, minor impact on throughput is observed.

IX. Conclusion

Current advances in parallel SMR allow independent commands to be executed concurrently in a replica. To keep replicas consistent, each replica has to carefully handle and respect dependencies among commands. This is a non-trivial task since it requires dependency detection on a possibly high volume of commands. As shown in this paper, for high command rates of light commands, existent schedulers for parallel SMR can become a bottleneck. In this paper, we have proposed mechanisms to efficiently represent and calculate dependency among commands, complemented by an efficient scheduling mechanism. We proved that our parallel SMR lends linearizable executions and experimentally assess the performance of our scheduling technique. Results demonstrated that the throughput achieved by our approach is 15 times higher than that observed by previous schedulers.

Acknowledgment

This work is partially supported by CAPES Brasil, PVE Project 88887.124751/2014-00.

References

[1] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.
[2] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys*, vol. 22, no. 4, pp. 299–319, 1990.
[3] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 3, pp. 463–492, 1990.
[4] H. Attiya and J. Welch, *Distributed Computing: Fundamentals, Simulations, and Advanced Topics*. Wiley-Interscience, 2004.
[5] M. Burrows, "The chubby lock service for loosely-coupled distributed systems," in *OSDI*, 2006.
[6] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "Zookeeper: wait-free coordination for internet-scale systems," in *ATC*, vol. 8, 2010.
[7] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *EuroSys*, 2015.
[8] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, 2003, pp. 29–43.
[9] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 1–26, 2008.
[10] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *MSST*, 2010.
[11] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, 2010.
[12] I. Moraru, D. G. Andersen, and M. Kaminsky, "There is more consensus in egalitarian parliaments," in *SOSP*, 2013.
[13] R. Kotla and M. Dahlin, "High throughput byzantine fault tolerance," in *DSN*, 2004.
[14] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin, "All about eve: execute-verify replication for multi-core servers," in *OSDI*, 2012.
[15] P. J. Marandi, C. E. B. Bezerra, and F. Pedone, "Rethinking state-machine replication for parallelism," in *ICDCS*, 2014.
[16] R. Kapitza, M. Schunter, C. Cachin, K. Stengel, and T. Distler, "Storyboard: Optimistic deterministic multithreading." in *HotDep*, 2010.
[17] H. Cui, R. Gu, C. Liu, T. Chen, and J. Yang, "Paxos made transparent," in *SOSP*, 2015.

[18] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM*, vol. 43, no. 2, pp. 225–267, 1996.
[19] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems*, vol. 16, no. 2, pp. 133–169, May 1998.
[20] P. J. Marandi, C. E. B. Bezerra, and F. Pedone, "Rethinking state-machine replication for parallelism," in *ICDCS*, 2014.
[21] S. Benz, P. J. Marandi, F. Pedone, and B. Garbinato, "Building global and scalable systems with atomic multicast," in *Middleware*, 2014.
[22] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild *et al.*, "Spanner: Google's globally-distributed database," in *OSDI*, 2012.
[23] Z. Guo, C. Hong, M. Yang, L. Zhou, L. Zhuang, and D. Zhou, "Rex: Replication at the speed of multi-core," in *EuroSys*, 2014.
[24] M. Olszewski, J. Ansel, and S. Amarasinghe, "Kendo: efficient deterministic multithreading in software," *ACM Sigplan Notices*, vol. 44, no. 3, pp. 97–108, 2009.
[25] N. Hunt, T. Bergan, L. Ceze, and S. D. Gribble, "DDOS: taming nondeterminism in distributed systems," in *ACM SIGPLAN Notices*, vol. 48, no. 4, 2013, pp. 499–508.

APPENDIX

Definition 1 (Batch, Batch Sequence)**.** A *batch* is a sequence of commands. A *batch sequence* is a pair $(B, <_B)$ where B is a set of batches and $<_B \subseteq B \times B$ is an irreflexive total order (this total order represents the atomic broadcast functionality).

Definition 2 (Conflict, Batch Sequence Dependency Relation)**.** Two commands *conflict* if any readset or writeset of one command intersects the writeset of the other. Given a batch sequence and the conflict relation $\#_C \subseteq C \times C$ among commands, the derived *batch sequence conflict relation* $\#_B$ is obtained by lifting the conflicts over C to conflicts between batches involving those commands. The *batch sequence dependency relation* $\prec_B$ is the transitive closure of $<_B \cap \#_B$, respecting the total order for conflicting batches.

Definition 3 (Execution)**.** An *execution* of $\prec_B$ is any total order that is compatible with $\prec_B$. By construction $\prec_B$ is an irreflexive partial order since $\prec_B \subseteq <_B$, $<_B$ is irreflexive and antisymmetric.

Definition 4 (Replica's Dependency Graph)**.** Given a batch sequence $(B, <_B)$ and its conflict relation $\#_B$, a *replica* R builds a dependency graph $DG = (B_{dg}, E)$, such that:

i) starting with $DG = (\emptyset, \emptyset)$ (line 9), each and all elements of B are taken for inclusion in DG following the order $<_B$ (see lines 14 to 15);

ii) while including a batch b_i in $DG(B_{dg}, E)$, b_i is included in B_{dg} and is checked for conflict with every other $b_j \in B_{dg}$ such that (b_j, b_i) is included in E in case of conflict (see dgInsertBatch()).

Proposition 1. *The dependency graph $DG = (B_{dg}, E)$ constructed for a batch sequence $(B, <_B)$ and a batch sequence conflict relation $\#_B$ is a directed acyclic graph (DAG). Proof:* Due to Definition 4(i), B_{dg} has all batches previous to b_i (a batch being included in DG) according to $<_B$, and eventually B_{dg} will have all batches from B. Due to Definition 4(ii) $\forall b_i, b_j \in B, (b_i, b_j) \in E$ iff $(b_i, b_j) \in <_B \wedge (b_i, b_j) \in \#_B$, or in other words iff $(b_i, b_j) \in (<_B \cap \ \#_B)$, the transitive closure of which is $\prec_B$. Since $\prec_B$ is a partial order, DG is a DAG.

Definition 5 (Replica's Multithreaded Execution)**.** Given a dependency graph $DG = (B, E)$ a *replica* R has a finite set of working threads that execute batches in B, such that:

i) DG is manipulated in mutual exclusion;

ii) a batch is chosen for execution according to dgGetBatch();

iii) after processing a batch, a thread removes it from DG using dgRemoveBatch(). This ensures that incoming batches are considered for conflict with batches both pending and under execution.

Proposition 2. *A replica's execution is compatible with $\prec_B$. Proof:* Since DG is a DAG compatible with $\prec_B$, item (ii) above ensures that any replica execution is compatible with $\prec_B$: the choice of next batch is always the lowest element w.r.t. $\prec_B$, a node of DG without incoming edges. If there are several such nodes, the lowest w.r.t. $<_B$ is chosen.

Proposition 3. *Non-deadlock: there is always a free batch to execute in DG. Proof:* Supposing DG is not empty and there are no $taken$ batches (batches in execution by worker threads), then there exists at least one lowest element that does not depend on any other batch, i.e., $\exists b_i \in B, \forall b_j \in B \ (b_j, b_i) \notin E$. This follows from $E \subseteq \prec_B$, $\prec_B$ is a partial order and B is finite.
Moreover, when a batch b_i is excluded form DG, then all dependencies from b_i are excluded. Let b_j be the next batch according to $<_B$, then either (a) b_i and b_j where independent or (b) the dependency $(b_i, b_j) \in E$ was excluded (see dgRemoveBatch()). In any case, since b_j is now the lowest element and there are no cycles, b_j does not depend on any batch and is free to execute, preserving this property.

Proposition 4. *No starvation of batches. Proof:* Since a batch b has an order in $<_B$ and there is no deadlock, b will eventually be processed.

Proposition 5. *Replicas are consistent.* Two replicas are consistent if, having processed a batch sequence, they converge to identical states. *Proof:* Follows from Proposition 2. Both replica runs are executions compatible with $\prec_B$, respecting total order of dependent batches.

Proposition 6. *A replica's execution is consistent with the linearizability criterion. Proof: linearizability* states that an execution respects the real-time ordering of commands across all clients and the semantics of the commands as defined in their sequential specifications. Commands c_i and c_j do not overlap in time if one command, say c_i is submitted by a client and responded by the service before c_j is submitted by another client. Linearizability holds since: (a) non overlapping commands are naturally submitted and responded sequentially; (b) overlapping in time, dependent commands are executed at all replicas in a same sequential order fixed by the ordering protocol; (c) overlapping in time, independent commands can be executed in different moments in different replicas, but preserve the semantics of their sequential specifications since they are independent on the concurrent batches being processed.