# Boosting concurrency in Parallel State Machine Replication

Ian Aragon Escobar
PUCRS
Brazil

Fernando Luís Dotti
PUCRS
Brazil

Eduardo Alchieri
Universidade de Brasília
Brazil

Fernando Pedone
Università della Svizzera italiana (USI)
Switzerland

## Abstract

State machine replication (SMR) is a well-known approach to implementing fault-tolerant services, providing high availability and strong consistency. To boost the performance of SMR, some proposals execute independent commands concurrently, while dependent commands execute sequentially in the total delivery order. The most general approach to handling command dependencies resorts to a directed acyclic graph (DAG), where nodes represent commands and edges represent dependencies. In this paper we show that due to the command arrival and multithreaded execution rates of SMR, a highly concurrent implementation of a DAG is needed. We show that a typical coarse-grained DAG implementation, where the whole graph is a critical section, results in a bottleneck in the replica. We propose two improvements to the coarse-grained DAG approach: fine-grained algorithms, using lock-coupling, and lock-free algorithms. Our fine-grain algorithms lock individual vertices in the DAG. The lock-free algorithms use nonblocking synchronization, with atomic operations, and lazy synchronization to postpone physical removal of nodes. All algorithms were integrated in a parallel SMR prototype. Experimental evaluation revealed that the fine-grained algorithms are also subject to a bottleneck. The lock-free implementation, however, sports linear speedup with the number of working threads, in some cases scaling up to 64 threads.

***CCS Concepts*** • **Computer systems organization** → **Dependable and fault-tolerant systems and networks**; • **Computing methodologies** → *Distributed algorithms*;

***Keywords*** State Machine Replication, Concurrent Algorithms, Fault Tolerance

**ACM Reference Format:**
Ian Aragon Escobar, Fernando Luís Dotti, Eduardo Alchieri, and Fernando Pedone. 2019. Boosting concurrency in Parallel State Machine Replication . In *Middleware '19: Middleware '19: 20th International Middleware Conference, December 8–13, 2019, Davis, CA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3361525.3361549


*Middleware '19, December 8–13, 2019, Davis, CA, USA*

ACM ISBN 978-1-4503-7009-7/19/12…$15.00
https://doi.org/10.1145/3361525.3361549

## 1 Introduction

Designing applications that tolerate failures and scale performance is important but challenging. One way to face this challenge is to rely on fundamental abstractions. Classic state machine replication (SMR) is a simple, yet effective approach to fault tolerant and strongly consistent applications [22, 29]. The idea is to replicate the application server and execute client commands at the replicas in the same order and deterministically. By starting in the same initial state and executing the same sequence of commands, replicas transition through the same sequence of states and produce the same responses for each command. SMR has been successfully used in many online services (e.g., [5, 17]).

Classic SMR, however, provides limited performance. This happens because every replica executes all commands sequentially, to ensure determinism. Sequential command execution at a replica is detrimental to performance and a poor use of modern hardware. To boost performance, replicas must execute commands concurrently and thereby benefit from modern multi-processor servers. Parallel SMR techniques are based on the observation that independent commands can execute concurrently while only dependent commands must be serialized and executed in the same order by the replicas [29]. Two commands are *dependent* (or *conflict*) if they access common state and at least one of the commands changes the state, and *independent* otherwise. Executing dependent commands concurrently may result in inconsistent states across replicas. Although the performance of a parallel SMR technique depends on specifics of the technique and the workload mix of independent and dependent commands, it has been shown that parallel approaches result in substantial improvements in performance (e.g., [2, 19, 21, 23, 24]).

This paper considers a class of parallel SMR protocols that rely on a scheduler thread to assign commands to worker threads at each replica. For example, CBASE, a protocol in this category, uses a *dependency graph* (in fact, a directed acyclic graph or DAG) at each replica to encapsulate dependencies [21]. For each new incoming command at a replica, the replica scheduler includes the command and its dependencies in the graph; worker threads remove commands from the graph respecting these dependencies and execute the commands. There is an edge from command $r_i$ to command $r_j$ in the DAG if $r_i$ is ordered before $r_j$, and both commands conflict. Intuitively, an edge from $r_i$ to $r_j$ means that $r_i$ must be executed before $r_j$.

Modeling and tracking command dependencies using a dependency graph is an elegant solution to the problem of command scheduling in parallel SMR. It poses a new challenge, however: as we show in this paper, under high load (e.g., hundreds of thousands of commands per second), dependency tracking may become itself the performance bottleneck. This is particularly the case if the scheduler and worker threads access the graph using coarse-grain locks in exclusive mode (e.g., in CBASE there is a single exclusive lock on the whole graph) and the cost of executing application commands is low, compared to the cost of executing graph operations to include and remove nodes and edges.

In this paper, we explore efficient implementations of the scheduler-workers approach. We capture fundamental requirements of parallel SMR with a new abstract data type, the *Conflict-Ordered Set (COS)*. COS generalizes dependency graph-based techniques: the scheduler includes new commands in the COS (i.e., *insert* operation), respecting their incoming order; worker threads select commands in the COS for execution (i.e., *get* operation), according to command dependencies, and then remove the commands from the COS once executed (i.e., *remove* operation). From this perspective, CBASE is a simple implementation of COS: it essentially serializes the execution of COS operations.

We present in the paper three implementations of COS operations. The first implementation, our baseline, uses coarse-grain locking, as CBASE's single-lock approach. The second implementation uses fine-grain locking, at the granularity of graph nodes, instead of a lock on the whole dependency graph. Even though there are a few concurrent graph proposals in the literature [7, 18, 28], this is a relatively unexplored topic [7]. We use the hand-over-hand locking (or lock coupling) technique [3, 26] to ensure that concurrent operations do not violate SMR ordering constraints and do not corrupt the graph structure.

Our third implementation introduces lock-free DAG operations. We use both nonblocking synchronization and lazy synchronization to implement lock-free operations. Nonblocking synchronization relies on atomic operations; it is used to include nodes in the graph and reserve nodes for execution. Lazy synchronization postpones graph remove operations. In more detail, nodes in the graph are first logically removed while physical node removal is performed during the insert operation, using a helping approach. The technique simplifies the coordination of the remove operation with other operations, at the cost of a slightly more complex insert operation.

We implemented all three techniques and assessed their performance in two environments. In the first environment, we consider the performance and scalability of the techniques using a shared dependency graph in a single node. In the second environment, we use BFT-SMaRt [4], a state machine replication framework, to compare the techniques in a deployment with three replicas. While the first two blocking algorithms impose important throughput bottlenecks both in the standalone dependency graph and in BFT-SMaRt deployements, the third algorithm has shown linear speedup with the number of working threads, outperforming the first algorithms in all scenarios investigated, showing more than 2.5× performance increase in some cases.

This paper makes the following contributions: (a) we identify the requirements of parallel SMR as the COS abstract data type; (b) we propose fine-grained locking and lock-free algorithms to implement COS; (c) we experimentally evaluate the performance and scalability of the proposed algorithms on the dependency graph data structure alone and in BFT-SMaRt.

## 2 System model and definitions

We assume a distributed system composed of interconnected processes. There is an unbounded set of client processes and a bounded set of $n$ server processes (replicas). 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. A process is *correct* if it does not fail, or *faulty* otherwise. There are up to $f$ faulty replicas, out of $n = 2f + 1$ replicas.

Processes communicate by message passing, using one-to-one or one-to-many communication. One-to-one communication uses 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 uses atomic broadcast, defined by primitives $broadcast(m)$ and $deliver(m)$, where $m$ is a message. Atomic broadcast ensures the following properties [9, 13][1]:

- *Validity*: If a correct process broadcasts a message $m$, then it eventually delivers $m$.
- *Uniform Agreement*: If a process delivers a message $m$, then all correct processes eventually deliver $m$.

[1] Atomic broadcast needs additional synchrony assumptions to be implemented [6, 10]. These assumptions are not explicitly used by the protocols proposed in this paper.

- *Uniform Integrity*: For any message $m$, every process delivers $m$ at most once, and only if $m$ was previously broadcast by a process.
- *Uniform Total Order*: If both processes $p$ and $q$ deliver messages $m$ and $m'$, then $p$ delivers $m$ before $m'$, if and only if $q$ delivers $m$ before $m'$.

Our consistency criterion is *linearizability* [16]. An execution is linearizable if there is a way to total order the operations such that (a) it respects the semantics of the objects accessed by the operations, as expressed in their sequential specifications; and (b) it respects the real-time ordering of the operations in the execution. (There exists a real-time order among two operations if one operation finishes at a client before the other operation starts at a client.)

# 3 Parallel state machine replication

## 3.1 Background

State Machine Replication (SMR) renders a service fault-tolerant by replicating the server and coordinating the execution of client commands among the replicas [22, 29]. 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 (defined in §2). In classical SMR, linearizability can be achieved by having clients atomically broadcast commands and replicas execute commands sequentially in the same delivery order (see Figure 1(a)). Since commands are deterministic, replicas will produce the same state changes and responses after the execution of the same sequence of commands.

Classic 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, independent commands (i.e., those that are neither directly nor indirectly dependent) can be executed concurrently without violating consistency [29]. Two commands are *independent* (or *non-conflicting*) if they either access different variables or only read variables commonly accessed; conversely, two commands are *dependent* (or *conflicting*) 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.

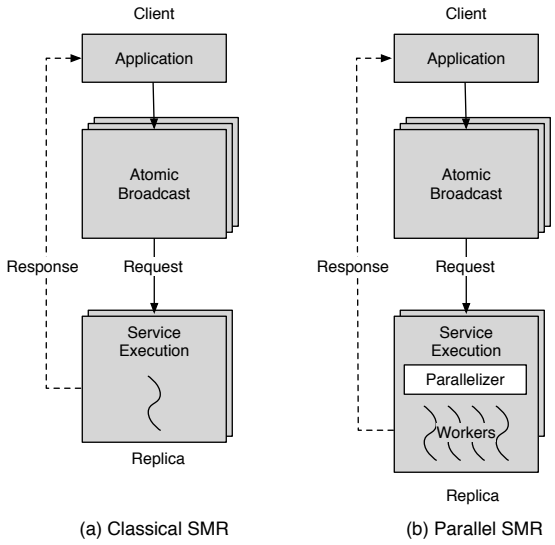



**Figure 1.** Classical versus parallel state machine replication

## 3.2 The basics of parallel execution

A few approaches have been suggested in the literature to execute independent commands concurrently with the goal of improving performance (e.g., [2, 19, 21, 23, 24]). In this section, we describe CBASE, the approach proposed in [21] and the motivation for the techniques proposed in this paper. We recall other approaches to parallel SMR in §8.

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 for execution. The parallelizer at each replica delivers commands in total order, examines command dependencies, and includes delivered commands in 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 get independent commands from the graph (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 (together with the edges that represent its dependencies) and responds to the client that submitted the command.

## 3.3 Fundamental requirements

We can generalize the requirements for parallel execution of commands with an abstract data type that keeps track of the order among conflicting commands. We call this data structure a *Conflict-Ordered Set (COS)*. This data structure is defined by three primitives with sequential specifications as follows. We assume $C$ to be the set of possible commands and $\#_C \subseteq C \times C$ the conflict relation between commands.

- $insert(c \in C)$ inserts command $c$ in the data structure;
- $c \in C : get()$ returns $c$ if and only if:
  - $c$ is in the data structure,
  - no previous $get()$ has returned $c$, and
  - there is no $c'$ in the data structure inserted before $c$ such that $(c, c') \in \#_C$;
- $remove(c \in C)$ removes $c$ from the data structure.

### 3.4 Generic P-SMR algorithm

In Algorithm 1 we detail the behavior of the scheduler and worker threads based on COS. When a command is delivered, it is inserted in the data structure (line 10). A varying number of worker threads get free commands to execute (line 13). When executed (line 14), commands are removed from the structure (line 15).

**Algorithm 1** Scheduler and worker threads

```
1:  constants and data structures
2:      T : number of working threads
3:      COS : the conflict-ordered set
4:  procedure Init()
5:      for all id ∈ 1..T do                          {for each worker thread...}
6:          start workingThread t_id
7:      start scheduler
8:  scheduler works as follows:
9:  when deliver(c)                                   {when a new command arrives}
10:     COS.insert(c)                                 {insert it in the stucture}
11: workingThread t_id executes as follows:
12: loop                                              {forever...}
13:     c ← COS.get()                                 {get a command with do dependencies}
14:     execute(c)                                    {execute c and then}
15:     COS.remove(c)                                 {remove c from the structure}
```

In the next three sections, we present different implementations of the COS abstract data type.

## 4 Scheduling based on a sequential graph

Algorithm 2 uses a directed acyclic graph (DAG) to implement the *COS* primitives. In the DAG, we represent commands as nodes and command dependencies as edges. Edge $(c_i, c_j)$ indicates that $c_i$ has to execute before $c_j$. The DAG is shared by the scheduler and worker threads, which access the graph in mutual exclusion. The algorithm relies on a monitor for synchronization among these primitives, using public procedures and condition variables. The same graph structures and synchronization technique are used both in [21] and [25].

Procedure *insert* inserts a node in the DAG with bounded number of nodes (lines 12–12). During the insertion of a new node $c_i$, *insert* considers all previous nodes in the graph for conflict and inserts edges from conflicting nodes to $c_i$ (lines 13–15). Finally, the status of $c_i$ is set to indicate it is waiting (line 16), $c_i$ is included in the graph (line 17) and checked if

**Algorithm 2** Coarse-grained DAG - COS primitives

```
1:  constants and data structures
2:      DG = (N, E) ← (∅, ∅)                          {the shared DAG}
3:      condition nFull, hasReady                     {monitor condition variables}
4:      maxN ← desired graph size                     {maximum number of nodes}
5:      st(n ∈ N) ∈ {wtg, exe}                        {state is waiting or executing}
6:  private procedure boolean : conflict(c_i, c_j)
7:      returns (c_i, c_j) ∈ #_C
8:  private procedure testReady(c_i)
9:      if {c_j ∈ N|(c_j, c_i) ∈ E} = ∅ then          {has no dependencies}
10:         signal(hasReady)
11: public procedure insert(c_i)                      {i in c_i denotes the order}
12:     if |N| = maxN then                            {if full, then wait}
13:         wait(nFull)
14:     ∀c_j ∈ N                                      {any command in the graph that conflicts}
15:         if conflict(c_i, c_j) then                {with the one being inserted}
16:             E ← E ∪ {(c_j, c_i)}                  {should be processed first}
17:     st(c_i) ← wtg                                 {no worker thread took it}
18:     N ← N ∪ {c_i}                                 {command is inserted in the graph}
19:     testReady(c_i)                                {check if ready to be executed}
20: public procedure command : get()
21:     while true do                                 {find node waiting, with no dependencies}
22:         let rdySet = {c_i ∈ N|st(c_i)=wtg∧∀c_j ∈ N, (c_j, c_i) ∉ E}
23:         if |rdySet| ≠ 0 then
24:             let c_k ∈ rdySet|∀c_l ∈ N, k < l       {take the oldest one}
25:             st(c_k) ← exe                         {mark it}
26:             return c_k
27:         else                                      {if there is no ready node ...}
28:             wait(hasReady)                        {... wait}
29: public procedure remove(c_i)                      {c_i assumed to be in N}
30:     ∀c_j ∈ N                                      {any node in the graph that}
31:         if (c_i, c_j) ∈ E then                    {... depends on this}
32:             E ← E \ {(c_i, c_j)}                  {has this dependency removed}
33:             testReady(c_j)                        {check if became ready}
34:     N ← N \ {c_i}                                 {remove processed node}
35:     signal(nFull)                                 {signal there is space for new nodes}
```

ready for execution (lines 18, 7–9). If $c_i$ has no dependencies, then condition *hasReady* is signaled to release a possible worker waiting for a ready command (line 28).

Procedure *get* retrieves ready commands, that is, commands that neither have dependencies nor are under execution (lines 21–22). More precisely, it returns the oldest ready command $c_k$ and marks $c_k$ as in execution. If no command is ready, it waits for an insertion (line 18) or a removal (line 33) that modifies the DAG, possibly freeing a command for execution.

In order to *remove* a command $c_i$ from the DAG, all outgoing edges (dependencies) from $c_i$ are first removed. During this process, other commands may become ready (lines 30–33). Then $c_i$ is removed from the set of nodes ($N$), thus freeing space and allowing a waiting insertion operation to continue (line 35).

Correctness of the algorithm follows from the fact that all operations in the graph are serialized (more details can be found in [21, 25]). Progress derives mainly from the fact that dependencies form a directed acyclic graph and therefore inductively there is always one command that can execute and free other commands to be executed.

## 5 Scheduling based on a fine-grained locking concurrent graph

A natural way to enhance concurrency during access to the dependency graph is to implement locking at node level instead of graph level. This opens the possibility of simultaneous operations in different nodes of the graph. To allow concurrent access to the graph, we use hand-over-hand locking (also known as lock coupling) [15, 26]. This technique is based on a total ordering of the nodes, which in our case is induced by the delivery order of atomic broadcast. We keep the same basic structure of nodes and directed edges as in the coarse-grained graph, and include additional edges to capture delivery order. According to the hand-over-hand locking strategy, an operation first has to lock the lowest node in the graph. Then, to traverse the ordered set, the operation will step-by-step lock the successor node before unlocking the previous one.

**Algorithm 3** Fine-grained DAG - Preliminaries

```
1:  data type
2:      Node{c : Command, mx : Mutex,
3:           st : {wtg, exe}}                    {state is waiting or executing}
4:      procedure Node createNode(c : Command)
5:          return new Node{c, new Mutex(), wtg}
6:  data structures and variables initialized
7:      Ordered set of Node : N, initially empty
8:      E = N × N, initially empty               {set of Edges}
9:      DG = (N, E)                              {the directed graph}
10:     space ← new Semaphore(maxSize)           {graph space}
11:     ready ← new Semaphore(0)                 {free commands to execute}
12:     head = createNode(⊥, null)               {lowest element and}
13:     tail = createNode(⊤, null)               {highest element}
14: procedure private boolean : conflict(n_i, n_j : Node)
15:     returns (n_i.c, n_j.c) ∈ #_C
16: procedure private boolean : isFree(n : Node)
17:     returns ({n_j ∈ N|(n_j, n) ∈ E} = ∅)     {no one referring to n}
18: procedure private testReady(n : Node)
19:     if isFree(n) then                        {has no dependencies}
20:         ready.up()
```

We now detail the algorithms for the scheduler and the working threads. The main structure is the ordered set of nodes (see Algorithm 3), each node having its own lock. There is a known lowest and highest element in this set that are special control nodes.

**Algorithm 4** Fine-grained DAG - COS primitives

```
1:  procedure insert(c:Command)
2:      space.down()                             {wait for space available}
3:      n ← createNode(c)                        {create node structure}
4:      n.lock()                                 {locks node being inserted}
5:      head.lock()                              {locks lowest element in N}
6:      aux ← head                               {aux starts in lowest node}
7:      ∀n_j ∈ N, in order                       {consider each node, in order}
8:          n_j.lock()                           {next n_j and previous are locked}
9:          aux.unlock()                         {unlocks previous: no overtake}
10:         if conflict(n, n_j) then
11:             E ← E ∪ {(n_j, n)}               {n depends of n_j}
12:         aux ← n_j                            {keeps previous with aux}
13:     N ← N ∪ {n}                              {becomes a visible node of the graph}
14:     testReady(n)                             {signal if n is free}
15:     aux.unlock()                             {release node locks}
16:     n.unlock()
17: procedure Node : get()
18:     ready.down()                             {wait free nodes to execute}
19:     head.lock()                              {locks lowest element in N}
20:     aux ← head                               {aux starts in lowest node}
21:     ∀n_i ∈ N, in order                       {consider each node, in order}
22:         n_i.lock()                           {next n_j and previous are locked}
23:         aux.unlock()                         {unlocks previous: no overtake}
24:         if isFree(n_i) ∧ (n_i.st = wtg) then
25:             n_i.st ← exe                     {mark it to execute}
26:             n_i.unlock()
27:             return n_i                       {deliver it for execution}
28:         aux ← n_i                            {not found, try next one}
29: procedure remove(n : Node)                   {n assumed in the graph}
30:     head.lock()                              {locks lowest element in N}
31:     aux ← head                               {aux starts in lowest node}
32:     ∀n_i ∈ N, in order                       {consider each other node, in order}
33:         n_i.lock()                           {next n_j and previous are locked}
34:         if aux ≠ n then                      {keeps lock in the node to be deleted}
35:             aux.unlock()                     {unlocks previous: no overtake}
36:         if (n, n_i) ∈ E then                 {n_i depends on n being removed}
37:             E ← E \ {(n, n_i)}               {delete dependency edges}
38:             testReady(n_i)                   {signal if n_i became free}
39:         aux ← n_i                            {keeps previous with aux}
40:     n_i.unlock()                             {unlocks last node of the walk}
41:     N ← N \ {n}                              {and finally remove n from the graph}
42:     n.unlock(), n.delete()
43:     space.up()                               {there is one more place in the graph}
```

The *insert* procedure (see Algorithm 4) is called sequentially upon total delivery of commands (see Algorithm 1). While traversing the nodes in order, conflicts from each existing node with the new node are checked and corresponding edges from the existing nodes to the new one added if needed (lines 7 to 12). Both the new node and each visited existing node are locked when edges among them are added (see line 11). After traversing all existing nodes (line 7), the node is added to the ordered set. At this point, all edges involving the new node and previous nodes have been already inserted.

The *get* procedure simply hand-over-hand traverses the list of nodes from lowest (oldest) to highest, checking if a node is free to execute. When a free node is found, it is marked so that no other thread chooses the node. The chosen node is then returned to the demanding worker thread.

The *remove* procedure is requested to remove an existing node from the graph, previously returned by *get*. For each visited node, the procedure checks whether there is an edge between the requested node and the visited node. When the node to be removed is found in the graph (line 34), it is left locked. From then on, edges from the node being removed to newer nodes can be found and deleted, possibly freeing other nodes for execution (lines 36 to 38). At the end the node is deleted.

Correctness stems from the fact that (a) nodes in the graph are visited in the order in which they are delivered (i.e., total order), (b) every visited node is first locked, and (c) a visited node is only unlocked after the next node is locked. Essentially, the first node in the graph, with respect to the delivery order, serializes operations. Multiple operations may execute concurrently, however, if they do not access common nodes in the graph. Finally, since operations only lock nodes following the total order, there is no deadlock.

# 6 Scheduling based on a lock-free concurrent graph

To maximize concurrency, in this section we present non-blocking algorithms to implement COS. COS primitive *get* is naturally blocking since it depends on the existence of commands to be executed in the graph (ready nodes). Also, the presented algorithms consider the realistic situation where the memory space is limited and thus *insert* may block if the graph is "full" (i.e., reaches a configurable maximum size). For clarity, in the following we separate the handling of these blocking conditions in one layer of operations and propose an underlying layer with lock-free operations. This is both to exploit maximum concurrency and to state clearly that the concurrent graph operations (when enabled) are lock-free.

## 6.1 Algorithms - blocking layer

The blocking layer deals with full graph and lack of ready node conditions. A full graph blocks *insert* operations, while absence of ready nodes in the graph blocks *get* operations. When these conditions are satisfied, the underlying layer with lock-free algorithms can be used. We measure graph space in number of "node slots" and use two counting semaphores to record the number of free slots in the graph and the number of nodes ready to execute (see lines 2 and 3 in Algorithm 5). Inserts need one free slot in the graph; *remove* increments the number of free slots in the graph.

**Algorithm 5** Blocking layer - COS primitives

```
1:  variables initialized
2:      space ← new Semaphore(maxSize)              {graph space}
3:      ready ← new Semaphore(0)                    {ready nodes}
4:  public procedure insert(c : Command)            {invoked in arrival order}
5:      space.down()                                {wait for space available}
6:      rdyNodes = lfInsert(c)                      {insert a node and inform if c is ready}
7:      ready.up(rdyNodes)                          {allow get to retrieve c if it is ready}
8:  public procedure NodeRef : get()
9:      ready.down()                                {wait free nodes to execute}
10:     return lfGet()
11: procedure remove(n : NodeRef)
12:     rdyNodes = lfRemove(n)                      {assumed: n.st = exe}
13:     ready.up(rdyNodes)                          {allow get to retrieve ready nodes}
14:     space.up()
```

## 6.2 Algorithms - lock-free layer

As in the previous techniques, we assume that *insert*s are called sequentially (see Algorithm 1). This order is important since it must be followed by conflicting commands to keep replicas consistent. While *lfInsert* operations are sequential (among themselves), *lfGet* and *lfRemove* operations are concurrent with any operations.

Another important aspect is that when a node is inserted, all edges representing dependencies involving this node and previous nodes have to be inserted too. If this restriction fails, a node could be wrongly considered ready for execution due to missing dependencies under insertion.

In Algorithm 6, line 2, a *Node* of the graph holds the respective command *c* and has an atomic field with its state, which may be: *wtg* waiting, meaning there are dependencies to be solved before executing this command; *rdy* the command is ready to execute; *exe* executing; *rmd* the node has been logically removed. Each node traverses these states in this order. The list of references to other nodes that depend on this node is *depMe* and the list of references to nodes it depends on is *depOn*. The node list is defined in line 11. The *nxt* field in line 7 represents the total order among commands.

Algorithm 6 has five private procedures used in three main operations of Algorithm 7, *lfInsert*, *lfGet*, and *lfRemove*. In the first three, we use atomic constructions to explicitly state the expected semantics of the procedures *compareAndSet*, *atomicAssign* and *atomicRead*, as well as to make clear in the algorithms were atomic structures are accessed.

The *lfRemove* operation to remove a node (Algorithm 7, line 33) marks the node as logically removed by atomically assigning *rmd* to its state. It uses *testReady*, line 1, to check wether nodes that depend on the one removed (line 36) became ready to execute, marking them (line 2). List modifications due to removal are performed using a helping approach during insertion (see line 20). The number of

nodes that turned ready is returned. This allows the blocking layer to free *get* operations that want to visit the graph. An $lfRemove(n)$ operation is only started when $n$ was first taken for execution with *lfGet* and therefore it is always true that $n.st = exe$.

The *lfGet* operation (Algorithm 7, line 27) is in charge of returning a node ready to execute from the graph. When this operation is called, a ready node in the graph must exist. The fist node atomically tested to be equal to $rdy$ (i.e., ready) is set to $exe$ (i.e., executing), returning the search.

**Algorithm 6** Lock-Free DAG - Preliminaries

```
1:  data type
2:    Node{
3:      c : Command,    { may be waiting, ready, executing, removed}
4:      atomic st : {wtg, rdy, exe, rmd}
5:      depOn : set of NodeRef          {nodes this one depends on}
6:      depMe : set of NodeRef          {nodes that depend on this}
7:      nxt : NodeRef                   {next node in arrival order}
8:    }
9:    NodeRef : atomic reference to Node
10: data structures and variables initialized
11:   N : NodeRef, initially ⊥          {the list of nodes in the graph}
12: procedure private boolean : compareAndSet(a, b, c)
13:   atomic { if a = b then a ← c; r ← true else r ← false }
14:   return r
15: private procedure atomicAssing(a, b)
16:   atomic { a ← b }
17: private procedure anyType: atomicRead(a)
18:   atomic { x ← a }
19:   returns x
20: private procedure Node createNode(c : Command)
21:   return new Node{c, wtg, ∅, ∅, ⊥}      {waiting by default}
22: private procedure boolean : conflict(n_i, n_j : Node)
23:   return (n_i.c, n_j.c) ∈ #_C
```

Finally, the *lfInsert* operation (Algorithm 7, line 12) assumes there is room in the graph to create a node. The new node is created with command $c$. The ordered list of nodes is then visited from the oldest to the newest node in the graph (line 18). Each node is tested if it was logically removed (line 19); in such a case, private procedure *helpedRemove* is used. Otherwise, if the visited node conflicts with the new node, a dependency edge among them is added (lines 21 and 22). At the end, the new node is inserted in the set of nodes and tested if ready (lines 25 and 26). During *helpedRemove* of a node, edges from other nodes waiting for this one (line 6) are removed (line 7). The logically removed node is then excluded from the graph (see lines 8 to 11).

### 6.2.1 Correctness

We start discussing COS semantics with concurrent operations. The fact that the graph topological structure is modified sequentially is central to the correctness of our algorithm. This stems from the following properties of the algorithm.

**Algorithm 7** Lock-Free DAG - Operations

```
1:  private function int: testReady(n : NodeRef)
2:    if {n_i ∈ n.depOn | n_i.st ≠ rmd} = ∅ ∧
                          compareAndSet(n.st, wtg, rdy) then
3:      return 1
4:    return 0
5:  private procedure helpedRemove(n', n : NodeRef)
6:    ∀n_i ∈ n'.depMe
7:      n_i.depOn ← n_i.depOn \ {n'}   {free dependents node from n}
8:    if n = N then                    {removing first element}
9:      atomicAssign(N, n'.nxt)                    {LPrmv}
10:   else                             {for any other, bypass n'}
11:     atomicAssign(n.nxt, n'.nxt)                {LPrmv}
12: public function int: lfInsert(c : Command)
13:   n_n ← createNode(c)              {create new node structure}
14:   if N = ⊥ then                    {empty graph}
15:     atomicAssign(N, n_n)           {insert the only node, finish ; LPins}
16:   else                             {let us find dependencies }
17:     n' ← N; n ← N
18:     while n' ≠ ⊥                   {consider each node, in order}
19:       if atomicRead(n'.st) = rmd then {n' logically removed}
20:         helpedRemove(n', n)        {remove it}
21:       else if conflict(n', n_n) then {n' is valid, they conflict?}
22:         n'.depMe ← n'.depMe ∪ {n_n}   {n_n depends on n'}
23:         n_n.depOn ← n_n.depOn ∪ {n'}  {reference count in n_n}
24:       n ← n'; n' ← atomicRead(n'.nxt)   {consider the next}
25:     atomicAssign(n.nxt, n_n)       {a visible node of the graph ; LPins}
26:   return testReady(n_n)            {informs if n_n is ready}
27: public procedure NodeRef : lfGet()
28:   n ← N                            {there is a ready node! start search}
29:   while n ≠ ⊥                      {consider each node, in arrival order}
30:     if compareAndSet(n.st, rdy, exe) then          {LPget}
31:       return n                     {found ready, mark it, deliver for execution}
32:     n ← atomicRead(n.nxt)
33: public function int: lfRemove(n : NodeRef)
34:   atomicAssign(n.st, rmd)          {logic removal ; LPlogicRmv}
35:   rdys ← 0                         {assumes n exists and has n.st = exe}
36:   ∀n_i ∈ n.depMe
37:     rdys ← rdys + testReady(n_i)   {check if n_i is ready, count}
38:   return rdys
```

**(i)** $lfInsert$ operations are invoked sequentially.
**(ii)** The $lfRemove$ operation marks a node as removed (logical remove) while actual removal from the graph takes place within $lfInsert$ during *helpedRemove*.
**(iii)** Concurrent operations read the graph topology and may only modify the node state $st$.
**(iii.a)** $lfGet$ operations traverse the node list to find a ready ($rdy$) node to execute, i.e., they read a node's $nxt$ field and may mark its $st$ate atomically as $exe$cuting.
**(iii.b)** An $lfRemove$ operation directly marks a given node's $st$ate as $rmd$ and uses $testReady$ to calculate nodes that became free. This is done by only reading the topological

information to calculate if nodes became ready, accordingly marking their *st*ate to *rdy*.

Therefore, topological modifications are sequential, easing the task of keeping the graph structure consistent.

***Interference freedom from* lfInsert *w.r.t.* lfGet *and* lfRemove.** With respect to the correctness of topological insertions and removals, besides concurrent operations being able only to *read* the topology, it remains to observe that any of these operations are able only to make the following atomic modifications to a node's *st*ate: from *wtg* to *rdy* (*testReady*), from *rdy* to *exe* ($lfGet$), and from *exe* to *rmd* ($lfRemove$). The only node state modification that could affect $lfInsert$ is when it becomes removed (*rmd*). Consider a node that is being visited (lines 18 to 24). Its state is either not *rmd* or *rmd*. In the first case, dependencies may be added to the node. Even if *immediately* after line 19 the node is concurrently set to *rmd*, node topological removal will happen sequentially, only after finishing the current insertion and during the next one, when *helpedRemove* is used. This is the case when $lfRemove$ and $lfInsert$ are concurrent and $lfRemove$ is linearized after $lfInsert$. Analogously, $lfInsert$ could be linearized after $lfRemove$ when the node is found with state *rmd*. It is simply not considered for dependencies with the new one, and is removed.

The above arguments care for the correctness of $lfInsert$ when $lfGet$ and $lfRemove$ are concurrent, showing that they do not interfere. Now we have to argue the other way, that $lfInsert$ does not interfere with $lfGet$ and $lfRemove$.

***Interference freedom from* lfGet *w.r.t.* lfInsert.** To argue for $lfGet$, first we claim that the traversal is safe during topological insertions and removals. This is based on:

**(i)** a total order of nodes, represented by the node *nxt* field;
**(ii)** node topological insertion is done at the end of the list, when the new node has all needed information, using an *atomic assignment* of the last node's *nxt* field (which is $\perp$) to the new node; and
**(iii)** node removal is done by by-passing the removed node, which may be any one in the list, *atomically assigning* the *nxt* field of the previous node to the next one.

Due to (i) and (ii) immediately above, whenever an operation $lfGet$ is traversing the list and a node is being concurrently inserted, either $lfGet$ will visit the node, meaning that $lfInsert$ was linearized before $lfGet$, or not, meaning the other way around. In any case, the traversal is safe due to the atomic reference assignment. The same applies to a removal. Due to (iii), either the traversal will not visit the node being concurrently removed, continuing the traversal, or it will. For the last case, garbage collection ensures that the by-passed node during *helpedRemove* will still be valid in memory since $lfGet$ holds a reference to it. Also, it still holds a valid *nxt* reference to the next node, which is used by $lfGet$ to advance in the traversal. Notice that this is inductively true if the next valid node is also concurrently removed, since it will be referenced by the first removed one and will still hold a reference to the next.

Concerning $lfGet$ semantics concurrently with either insertions or removals, our algorithms at the blocking layer ensure that when a $lfGet$ enters the graph, there is a *rdy* node to execute. With the above traversal guarantees, it will be returned by $lfGet$.

***Interference freedom from* lfRemove *w.r.t.* lfInsert.** Notice that $lfRemove$ has as argument an assumedly valid node which is in state *exe* since it was taken by a previous $lfGet$ operation. It remains to mark the node atomically as removed (*rmd*). Since the end of execution of a node will free possibly dependent nodes to execute, it remains to check if every dependent node is free, which is to check if all the nodes it depends on are removed. This is done with *testReady* which, given a node, will check if all nodes it depends on (*depOn*) are removed (*rmd*) and the state can be changed from *wtg* to *rdy*, see line 2.

Notice that while *testReady*, during $lfRemove$, is *reading* the node's dependency set *depOn*, *helpedRemove* during $flInsert$ may subtract references from this set. This is not a problem because *testReady* is checking if existing dependencies are from nodes not yet logically removed (line 2). In case such dependencies from logically removed nodes were already removed, their absence has the same effect.

***Interference freedom among* lfGet *and* lfRemove.** $lfGet$ and $lfRemove$ can take place concurrently since they leave the graph topology unchanged and operate on separate nodes, i.e. nodes respectively with state *wtg* and *exe*. Any order of concurrent $lfGet$ and $lfRemove$ is valid.

***Interference freedom among* lfGets.** The atomic compare and set of a node's state (*st*) ensures only one operation $lfGet$ will return a given node. Any node ready to execute is valid for $lfGet$. Any order of concurrent $lfGet$s is valid.

***Interference freedom among* lfRemoves.** Since $lfRemove$ assumes a valid node as argument, any order is valid. Notice that during concurrent $lfRemove$ we have concurrent *testReady*. Concurrent *testReady* will check if new nodes became ready to execute. Interference could take place if two nodes being removed had one common dependent node. In such case both *testReady* operations on the same node could result in a double signaling of ready node to the blocking layer. To avoid that, line 2 uses a compare and set operation to signal ready node in the first operation only.

Now we discuss the COS semantics for real-time ordered operations. We have to argue that, considering any two operations, if one operation returns before the other, then the effects of the first are observable before the other.

This is straightforward for operations $lfGet$ and $lfInsert$. Any operation after $lfGet$ will find the element returned

by $lfGet$ in state $exe$. As any operation after $lfInsert$ will traverse the list with the inserted element.

The remove operation logically removes a node, that afterwards is removed from the graph during an insertion. The effects of a remove, which may affect the semantics of other operations, are all caused during the logical remove ($lfRemove$). The logically removed node is not considered by $lfGet$ neither by $lfInsert$. Likewise, the switching of nodes from $wtg$ to $rdy$ due to a removal take place within the $lfRemove$. Therefore, any operation real-time ordered after $lfRemove$ will accordingly consider the graph without the removed element and with new ready ones as appropriate.

#### 6.2.2 Progress

The progress argument is the same as in the coarse-grained solution. Since commands can only depend on previous conflicting ones, according to a total order, dependencies will never build a cycle (i.e., the graph is acyclic). Therefore, there is always at least one command in the graph that is free to execute. When a free command executes, it will inductively free other commands assuring that there is at least a next one free to execute, or no command left.

## 7 Experiments

In order to compare the performance of the solutions proposed to implement the COS primitives, we implemented the three techniques studied in this paper and conducted several experiments. These techniques, as presented in the paper, are referred as *coarse-grained*, *fine-grained* and *lock-free*. Firstly, we assessed the performance of the data structures alone without integration in a SMR (§7.3) and then we compared their performance also when used for command scheduling in a parallel SMR (§7.4).

### 7.1 Environment

We implemented a scheduler that uses each of the graphs for command scheduling in Bft-SMaRt [4], which is a SMR library that can be configured to use protocols optimized to tolerate crash failures only or Byzantine failures. In all experiments, we configured Bft-SMaRt to tolerate crash failures. Bft-SMaRt was developed in Java and its atomic broadcast protocol executes a sequence of consensus instances, where each instance orders a batch of commands. To further improve the performance of Bft-SMaRt ordering protocol, we implemented interfaces to enable clients to send a batch of commands inside the same message. The experimental environment was configured with 7 machines connected in a 1Gbps switched network. The machines were configured with the Ubuntu Linux 18.04 operating system and a 64-bit Java virtual machine version 10.0.2. Bft-SMaRt was configured with 3 replicas hosted in separate machines (Dell PowerEdge R815 nodes equipped with four 16-core AMD Opteron 6366HE processors running at 1.8 GHz and 128 GB of RAM) to tolerate up to 1 replica crash. Up to 200 clients were distributed uniformly across another 4 machines (HP SE1102 nodes equipped with two quad-core Intel Xeon L5420 processors running at 2.5 GHz and 8 GB of RAM). The experiments with the data structures alone without integration in a SMR were executed in one of the Dell PowerEdge machines.

### 7.2 Application

We used a linked list application with operations to check whether an entry (i.e., an integer) is in the list (*contains*) and to include an entry in the list (*add*), representing a readers-and-writers service. Operation $contains(i)$ returns *true* if entry $i$ is in the list, otherwise it returns *false*; operation $add(i)$ includes $i$ in the list and returns *true* if $i$ is not in the list, otherwise it returns *false*. Notice that in the concurrency model for this application, *contains* commands do not conflict with each other but conflict with *add* commands, which conflict with all commands.

Hereafter, we refer to operations that check whether an entry is in the list and to operations that include an entry in the list as *read* and *write* operations, respectively. Each list was initialized with $1k$, $10k$ and $100k$ entries at each replica (ranging from 0 to *list size - 1*) representing operations with light, moderate, and heavy execution costs, respectively. The integer parameter used in an entry in a read and write operations was a randomly chosen position in the list.

In the experiments, we configured the maximum size of the dependency graph with 150 entries for all approaches. We measured the throughput of the system at the servers and the latency of each command at the clients. In the experiments with the data structures alone, we measured only the overall throughput obtained by the worker threads since it does not make sense to compute the latency in this case. A warm-up phase preceded each experiment.

### 7.3 Data structures performance

This section reports the performance results for the data structures alone (i.e., without integration in a SMR). To resemble the scheduler presented in Algorithm 1, one thread looped without waiting interval over a list of pre-created requests (to spare creation times), and invoked the *insert* operation. Moreover, each worker thread is as presented in Algorithm 1, looping to *get*, execute and *remove* a command.

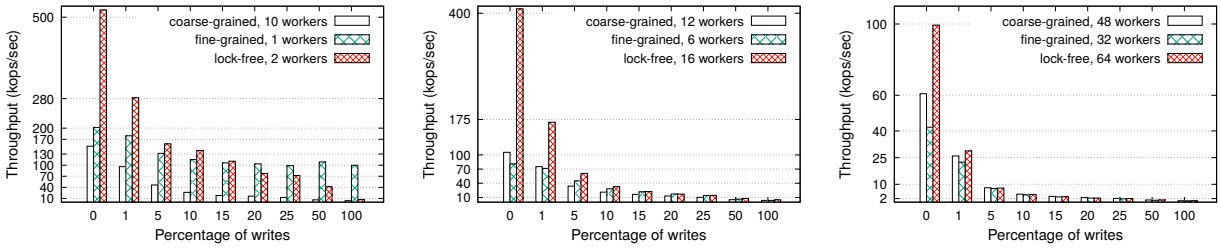
#### 7.3.1 Results for different degree of parallelism

Fig. 2 shows the throughput presented by each approach for different execution costs and number of worker threads, considering 0% of writes (i.e., only read operations). This workload allows maximum parallelism and is useful to define performance upper bounds for each technique.

In general, we can observe that the lock-free algorithm performs best in all scenarios and scales with the number of threads until it achieves peak throughput. In the experiments with light and moderate execution costs, the thread inserting


(a) Light (b) Moderate (c) Heavy

**Figure 2.** Throughput for different execution costs and number of workers (0% of writes).




(a) Light (b) Moderate (c) Heavy

**Figure 3.** Throughput for different percentage of writes and execution costs.

requests in the graph eventually becomes a bottleneck. We identified that after the peak throughput is reached, the graph mean population is close to zero, indicating that the insert thread is at its performance limit. For heavy operations, the time to execute operations is predominant in the system.

The other approaches presented much lower throughput, where performance is limited by the synchronization and contention in the dependency graph. Interestingly, the coarse-grained algorithm outperforms the fine-grained in most cases. This happens because, on the one hand, in the coarse-grained solution, removing a node with an independent operation needs little processing (only lines 34–35 of Alg. 2 are executed). On the other hand, to allow concurrent accesses to the graph, the fine-grained approach always removes a node by walking through the nodes of the graph (Alg. 4). Finally, notice that the performance gains in the lock-free solution are higher for the light and moderate workloads: as operations become more expensive, their execution cost dominate overall performance, and the time spent in synchronization becomes less important.

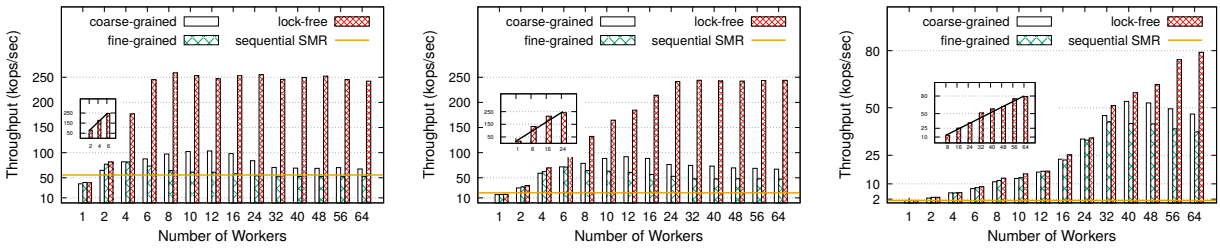
### 7.3.2 Results for different percentage of writes

Having established the peak throughput of each technique with 0% of writes (i.e., workload that allows maximum concurrency), we picked for each technique the best performing number of threads for light, moderate and heavy loads. We then conducted a second set of experiments to understand the impact of dependencies in each technique, leading to the results in Fig. 3. The percentage of writes represents the number of write operations with respect to all operations in a workload. For example, 15% of writes represents a workload with 15% of writes and 85% of reads. These operations were uniformly generated and inserted in the graphs.

The scenario with light requests shows that the lock-free algorithm outperforms the others in the range from 0% to 15% of writes while the fine-grained algorithm shows less impact as writes rise. This is because the best performing configuration of the fine-grained algorithm is with one worker thread, leading to a sequential execution. In such case, the concurrency enabled by the fine-grained technique outperforms the coarse-grained one. As dependencies tend to 100%, the fine-grained technique loses performance since it has to proportionally add and remove dependency edges in the graph. But under high percentage of writes, it outperforms the other techniques since the workload becomes more sequential and it is configured with just one worker thread.

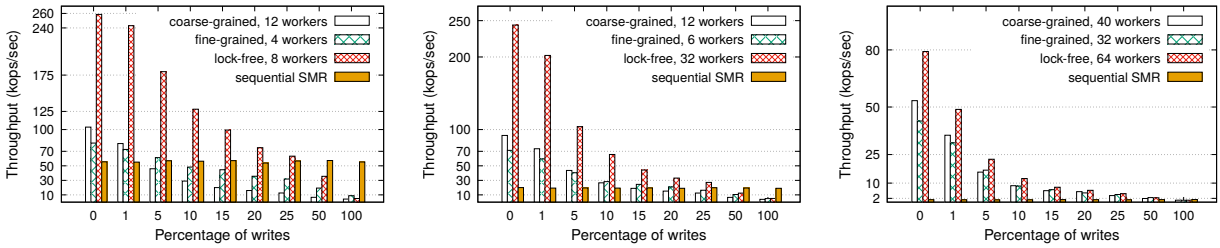
For moderate and heavy workloads, the scheduling cost is less impacting and the parallel request processing more important. In such cases, the lock-free algorithm clearly outperforms the others because it scales to a higher number of worker threads. This behavior is clearly visible in workloads with few writes, which allow a high degree of parallelism.

A final remark is about the considered lock granularity. Locking the complete graph (i.e., the coarse-grain approach) and individual graph nodes (i.e., the fine-grain approach) represent two ends of a "lock granularity spectrum". Alternatively, one could experiment with other granularities of locks (e.g., granular locks [11]), trading concurrency for overhead.




(a) Light (b) Moderate (c) Heavy

**Figure 4.** Throughput for different execution costs and number of workers (0% of writes), linear scale in the inset graphs.




(a) Light (b) Moderate (c) Heavy

**Figure 5.** Throughput for different percentage of writes and execution costs.

### 7.4 Parallel SMR Performance

This section compares the performance of the techniques when integrated in a parallel SMR. For completeness, we also present the results for a traditional sequential SMR.

#### 7.4.1 Results for different degree of parallelism

Fig. 4 shows the throughput presented by each approach for different execution costs and number of worker threads, considering a workload with 0% of writes. In general, the behavior was similar to the ones presented in the experiments with the data structures only. However, the performance values are smaller in these experiments since there is the overhead imposed by the SMR protocols and, moreover, the execution of these protocols also uses the machines computational power (see the BFT-SMaRt paper [4] for an explanation about the number of threads and the staged message processing used in the system).

Fig. 4 shows that the lock-free solution outperforms the other approaches for all configurations. Additionally, the lock-free technique scales linearly with the number of worker threads, as evidenced in the inset graphs, which show throughput values in a linear scale. Moreover, the parallel solutions outperform the sequential SMR for all configurations with more than one worker thread, which represents the case that leads to sequential execution for all techniques.

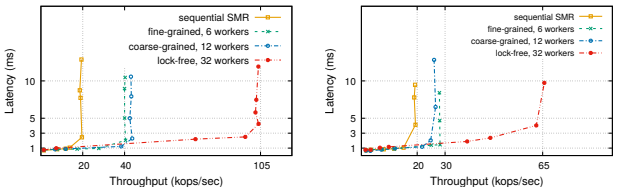
#### 7.4.2 Results for different percentage of writes

Fig. 5 shows the throughput presented by each technique for different execution costs and percentage of writes. Read and write operations were uniformly distributed among clients, respecting the defined percentage of writes. Again, we configured each solution with the number of worker threads that presented the best performance in the previous experiment.

These results show that the free-lock solution outperforms the other approaches for parallel execution in all cases. Fig. 5 also presents values for a sequential SMR, allowing us to observe the percentage of writes from where a sequential SMR outperforms the approaches that allow concurrent execution. For light and moderate loads, the lock-free solution outperforms a sequential SMR until up to 25% of writes. The time demanded to execute a request is more relevant for heavy loads, consequently in this case the solutions with parallelism outperform a sequential SMR for almost all cases. Although it is well known that sequential execution outperforms parallel techniques for high degrees of conflicting operations [1, 23], empirical evidence suggests that low conflict rates, between 0.3% and 2%, are the most realistic [5, 27].

For workloads with 5% and 10% of writes and moderate execution costs, Fig. 6 also shows latency versus throughput results. In the approaches for parallel SMR, reads and writes have similar latency because they have similar execution costs and the synchronization of writes impacts the performance of reads ordered after a write. Obviously, the same behavior occurs in the sequential SMR. Consequently, Fig. 6 presents the average latency considering both operations. It is possible to observe that all approaches presented similar latency until near system saturation and from this point latency increases abruptly. Since the same behavior occurs for

the other configurations and workloads, we opted to present only these cases as examples.




(a) 5% of writes (b) 10% of writes

**Figure 6.** Latency versus throughput for moderate cost.

## 8 Related work

This paper is at the intersection of two areas of research: concurrent graph structures and state machine replication.

### 8.1 Concurrent graph structures

In [28], a concurrent graph is proposed to calculate serializable executions. Nodes and edges represent transactions and their conflicts, respectively. Whenever a transaction is added, edges are included to represent conflicts. In case of cycles, vertices and edges are removed to keep the graph acyclic. The graph is implemented as a linked list. Nodes and edges can be created individually and concurrently. The node list is ordered according to a key. The synchronization strategy is *lazy* [15], or optimistic: In a first step, the list is searched without locks. Once the right position to operate on the list is found, locks are acquired on needed nodes. Once locks are obtained, a validation is performed to check if the conditions during search are still valid for those nodes. If not, the operation is repeated. Upon deletion, the node is first marked as logically deleted, then locking and actual removal take place. Nodes and edges are manipulated independently. The strategy allows wait-free operations to traverse the graph to check if a node is in the list as well as to detect cycles.

In [18], authors propose a concurrent graph without a specific application. An adjacency matrix is used for representation. It contains a fixed vertex set and allows concurrent operations to insert, remove or modify weights of edges. A dynamic traversal is proposed to obtain a consistent view, i.e., the weights of all edges visited have co-existed at some point in time despite concurrent modifications. Operations are wait-free [26], which is achieved using a helping mechanism [14]. Operations concurrent to updates help updates to carry out edge modifications.

A concurrent, unbounded and directed graph is proposed in [7]. Addition, removal and lookup on the sets of vertices and edges are supported on a lock-free basis, while a reachability query is obstruction-free. It also uses a helping strategy to achieve lock-freedom.

Both our lock-free graph algorithm and the ones described above build on basic principles to allow concurrent access to a shared data structure. However, while our algorithm implements a Conflict-Ordered Set (COS), the ones above implement operations on a single node or a single edge. It is unclear how to implement COS using these approaches.

### 8.2 State machine replication

It has been early observed that independent commands can be executed concurrently in SMR [29]. Previous works have shown that many workloads are dominated by independent commands, which justifies strategies for concurrent execution (e.g., [21, 23, 24]).

Existing proposals that introduce parallel execution in state machine replication differ in the strategy and architecture to detect and handle conflicts. We can classify related work in techniques that: (i) use application knowledge to deterministically process parallel commands [1, 2, 21, 23]; (ii) are oblivious to application knowledge [8, 12]; and (iii) employ optimistic strategies to parallelize commands [19, 24]. A scheduler that serializes the execution of dependent commands and dispatches independent commands to be processed in parallel by a pool of worker threads is an example of technique that exploits application knowledge (i.e., in the form of dependent and independent commands). This idea has been explored in transactional systems, where information about data items accessed by transactions can be inferred a priori (e.g., [20, 30]). Techniques that are oblivious to application knowledge resort to more complex runtime architectures to coordinate replicas to lead to compatible parallel executions. In the third case, replicas optimistically execute commands in parallel, as they arrive, and then check after execution if consistency is violated.

## 9 Conclusion

Parallel state machine replication techniques 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. In this paper, we have identified the fundamental problem that a scheduler-worker-based implementation of parallel state machine replication must face. We have also proposed an implementation that significantly improves the performance of existing approaches.

## Acknowledgments

We wish to thank the reviewers for the constructive comments and our shepherd Sonia Ben Mokhtar. This work is partially supported by CAPES - Print - PUCRS and the Swiss Nacional Science Foundation (SNSF). Ian A. Escobar is partially supported by CNPq through the PUCRS-BPA program.

## References


[1] E. Alchieri, F. Dotti, O. M. Mendizabal, and F. Pedone. 2017. Reconfiguring Parallel State Machine Replication. In *IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*. 104–113.

[2] Eduardo Alchieri, Fernando Dotti, and Fernando Pedone. 2018. Early Scheduling in Parallel State Machine Replication. In *ACM Symposium on Cloud Computing*.

[3] R. Bayer and M. Schkolnick. 1977. Concurrency of operations on B-trees. *Acta Informatica* 9, 1 (01 Mar 1977), 1–21. https://doi.org/10.1007/BF00263762

[4] Alysson Bessani, João Sousa, and Eduardo Alchieri. 2014. State Machine Replication for the Masses with BFT-SMaRt. In *DSN*.

[5] Mike Burrows. 2006. The Chubby Lock Service for Loosely-coupled Distributed Systems. In *OSDI*.

[6] T. D. Chandra and S. Toueg. 1996. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of ACM* 43, 2 (1996), 225–267.

[7] B. Chatterjee, S. Peri, M. Sa, and N. Singhal. 2018. A Simple and Practical Concurrent Non-blocking Unbounded Graph with Reachability Queries. *ArXiv e-prints* (Sept. 2018). arXiv:cs.DC/1809.00896

[8] Heming Cui, Rui Gu, Cheng Liu, Tianyu Chen, and Junfeng Yang. 2015. Paxos Made Transparent. In *25th Symposium on Operating Systems Principles*.

[9] Xavier Défago, André Schiper, and Péter Urbán. 2004. Total Order Broadcast and Multicast Algorithms: Taxonomy and Survey. *Comput. Surveys* 36, 4 (Dec. 2004), 372–421.

[10] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. 1985. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM* 32, 2 (April 1985), 374–382.

[11] Jim Gray and Andreas Reuter. 1992. *Transaction Processing: Concepts and Techniques* (1st ed.). Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[12] Zhenyu Guo, Chuntao Hong, Mao Yang, Dong Zhou, Lidong Zhou, and Li Zhuang. 2014. Rex: Replication at the Speed of Multi-core. In *Proceedings of the Ninth European Conference on Computer Systems*.

[13] Vassos Hadzilacos and Sam Toueg. 1993. Fault-tolerant Broadcasts and Related Problems. In *Distributed Systems*, Sape Mullender (Ed.). ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 97–145.

[14] M. Herlihy. 1990. A Methodology for Implementing Highly Concurrent Data Structures. *SIGPLAN Not.* 25, 3 (Feb. 1990), 197–206.

[15] M. Herlihy and N. Shavit. 2008. *The art of multiprocessor programming*. Morgan Kaufmann.

[16] Maurice Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Transactions on Programing Languages and Systems* 12, 3 (July 1990), 463–492.

[17] Patrick Hunt, Mahadev Konar, Flavio P Junqueira, and Benjamin Reed. 2010. ZooKeeper: wait-free coordination for internet-scale systems. In *ATC*, Vol. 8.

[18] Nikolaos D. Kallimanis and Eleni Kanellou. 2015. Wait-Free Concurrent Graph Objects with Dynamic Traversals. In *OPODIS*.

[19] Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. 2012. All about Eve: execute-verify replication for multi-core servers. In *OSDI*.

[20] Bettina Kemme, Fernando Pedone, Gustavo Alonso, Andre Schiper, and Matthias Wiesmann. 2003. Using Optimistic Atomic Broadcast in Transaction Processing Systems. *IEEE Trans. on Knowl. and Data Eng.* 15, 4 (July 2003), 1018–1032.

[21] Ramakrishna Kotla and Mike Dahlin. 2004. High throughput Byzantine fault tolerance. In *DSN*.

[22] L. Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (1978), 558–565.

[23] Parisa Jalili Marandi, Carlos Eduardo Benevides Bezerra, and Fernando Pedone. 2014. Rethinking State-Machine Replication for Parallelism. In *ICDCS*.

[24] P. J. Marandi and F. Pedone. 2014. Optimistic Parallel State-Machine Replication. In *SRDS*.

[25] O. M. Mendizabal, R. T. S. Moura, F. L. Dotti, and F. Pedone. 2017. Efficient and Deterministic Scheduling for Parallel State Machine Replication. In *IPDPS*.

[26] Mark Moir and Nir Shavit. 2004. Concurrent Data Structures. In *Handbook of Data Structures and Applications*.

[27] Iulian Moraru, David G Andersen, and Michael Kaminsky. 2013. There is more consensus in egalitarian parliaments. In *SOSP*.

[28] Sathya Peri, Muktikanta Sa, and Nandini Singhal. 2016. Maintaining Acyclicity of Concurrent Graphs. *CoRR* abs/1611.03947 (2016). arXiv:1611.03947 http://arxiv.org/abs/1611.03947

[29] F. B. Schneider. 1990. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *Comput. Surveys* 22, 4 (1990), 299–319.

[30] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *SIGMOD*.