State-Machine and Deferred-Update Replication: Analysis and Comparison

Paweł T. Wojciechowski, Tadeusz Kobus, and Maciej Kokociński

Abstract—In the paper, we analyze and experimentally compare two popular replication schemes relying on atomic broadcast: state machine replication (SMR) and deferred update replication (DUR). We estimate the lower bounds on the time of executing requests by the SMR and DUR systems running on multi-core servers. We also consider variants of systems that can process read-only requests with a lower overhead. In the analysis of DUR, we consider conflict patterns. We then formally show the scalability of SMR and DUR, which reflects the capacity of systems to effectively utilize an increasing number of processor cores. Next, we compare SMR and DUR experimentally under different levels of contention, using several benchmarks. We show throughput, abort rate (in DUR), and network congestion. The key results of our work are that neither system is superior in all cases, and that the theoretical and experimental results are heavily influenced by the dominance of either the CPU execution time or atomic broadcast time. We therefore propose to combine both replication schemes and gain the best of both worlds.

Index Terms—State machine replication, deferred update replication, distributed transactional memory.

1 Introduction

With the emergence of cloud computing, where services in the cloud can be accessed by a large number of clients in parallel, there was an explosion of interest in various approaches to replication. Replication can improve service availability and reliability by processing client requests in parallel and tolerating machine failures. A *replicated service* is deployed on several interconnected servers, each of which may fail independently and can only access its own local memory (or storage). The accesses are coordinated using a *replication scheme*, so that the system maintains a consistent state view despite failures of some servers or communication links. However, each replication scheme incurs the performance overhead due to the required synchronization.

In this paper we formally analyze and experimentally compare under various conditions the performance of two well-known replication schemes. They gave rise to numerous systems that were developed for full replication of services without resorting to any central coordinator. However, the systems vary widely, making it hard to compare the schemes.

State machine replication (SMR) [1], [2] is a classical approach for building fault-tolerant systems. It assumes that a process (service) being replicated can be modelled as a *state machine* which executes arbitrary but deterministic computation. Replicated processes are coordinated, so that exactly the same sequence of commands are executed by every process. In particular, each client request (which may consist of a sequence of read/write operations) is executed by every process. Since processes are deterministic, starting from the same initial state, each of them will produce the same state change. Therefore crash of some servers can be tolerated.

Deferred update replication (DUR) [3] was introduced in the context of replicated databases but the idea is more general. A client request can be sent to any non-faulty server (database replica) that executes the request as an *atomic transaction*—a sequence of read/write operations on the database. Once a finished transaction is successfully certified, all replicas are updated with the values written by the transaction. *Transaction certification* means checking if a transaction conflicts with local or remote concurrent transactions that have already committed. If any conflicts were detected, the conflicting transaction is aborted and reexecuted.

The two replication techniques are fundamentally different: SMR supports arbitrary (but deterministic) computation. DUR supports non-deterministic computation and requires rollbacks. SMR typically implements *linearizability* [4] while DUR typically implements *serializability* [5]. SMR ensures that command execution always succeeds (provided enough replicas exist). In DUR, an atomic transaction may abort and never complete, even after successive reexecutions.

We therefore propose a common framework of *transactional replication (TR)* and performance models of SMR and DUR that allow us to compare the two techniques according to a few metrics, including performance and scalability. Our study shows under what circumstances each technique performs at its best (considering low and high network capacity). We also present benchmark performance results obtained with two prototype systems: *JPaxos* [6] for SMR, and *Paxos STM*, which we developed to experiment with DUR.

• *The authors are with the Institute of Computing Science, Poznań University of Technology, 60-965 Poznań, Poland.*
E-mail: {Pawel.T.Wojciechowski, Tadeusz.Kobus, Maciej.Kokocinski}@cs.put.edu.pl

Manuscript received August 3, 2015; revised April 10, 2016; accepted July 1, 2016. Date of publication ?? August 2016; date of current version ?? August 2016. Recommended for acceptance by ??.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. ????.

We only regard the SMR and DUR systems that use a *total-order (atomic) broadcast* protocol (or an *abcast* protocol, in short). SMR requires the protocol to send a client request to all replicas for replica coordination, while DUR requires the protocol to send transaction read-sets and any state updates to all replicas as part of agreement coordination. As shown in [7], [8], agreement coordination relying on abcast has several advantages: it prevents deadlocks and allows better scalability than when using a two-phase commitment protocol (see also [9], [10]). JPaxos and Paxos STM share the same implementation of a fault-tolerant abcast protocol, which uses the Paxos algorithm [11].

1.1 Motivations and contributions

The motivations to conduct our research were twofold. Firstly, there was no prior work on the rigorous analysis and comparison of SMR and DUR. Secondly, reasoning about the advantages, limitations, and possible optimization paths of the replication schemes is difficult without a performance model that abstracts from any uninteresting details. Although the *modus operandi* of SMR and DUR may appear simple, concurrency and network capacity make the model quite subtle.

The main contributions of our work are the following:

- We defined a performance model of SMR and DUR, and also their typical *optimized variants*, where read-only requests are processed in an unconstrained way, in parallel with update requests;
- We estimated the lower time bound on the execution of requests by servers with multi-core processors, parameterized by workload type, network capacity (modelled by capacity of abcast), and the number of servers/CPU cores and conflicts (in DUR);
- We used the model to analytically compare the two replication schemes with their optimized variants, and showed when SMR can be faster than sequential processing of requests. We also analyzed the worst possible conflict patterns in DUR;
- We formally analyzed the scalability of the SMR and DUR systems on multi-core processors by inferring the speedup and efficiency of the parallel systems with the estimated lower bounds;
- We experimentally compared the throughput and scalability of JPaxos and Paxos STM using two microbenchmarks (Hashmap and Bank), and also compared their execution with a non-replicated service executing all requests sequentially and with predictions made by our performance model.

This paper is a largely revised and extended version of [12]. To our best knowledge, we are the first to formally analyze parallelism and scalability of SMR and DUR and to compare their performance under varying contention in a uniform communication environment.

1.2 Paper structure

The paper has the following structure. First, we discuss related work in §2. Then, we present the analytical model in §3 and §4. We discuss our evaluation experiments and compare SMR and DUR in §5. Then, we propose to combine them in §6. Finally, we conclude in §7.

2 Related Work

Replication is one of the most researched topics by the distributed systems community. Different models and replication techniques have emerged in the due course of years (see [3] for a survey). In our model, we consider full replication with strong consistency, i.e., each data is replicated on every live server. Below we briefly describe some of the work most closely related to ours.

The idea of state machine replication was introduced in [1], [2], [13] and has evolved considerably since then. The SMR systems typically use fault-tolerant *distributed agreement* protocols (e.g., Paxos) that guarantee progress if each protocol message is received by a quorum of processes (see also the quorum-based replication in [14]).

Deferred update replication [3] employs a distributed multi-master replication protocol, where multiple server processes are peers that can execute transactions concurrently and propagate the transactions' updates eagerly or lazily to all replicas. The blocking protocol can use e.g., *two-phase commit (2PC)* [15]. In our model, however, we consider DUR relying on atomic broadcast (see e.g., [7], [16], [17] among others), which avoids blocking, thus increasing parallelism and performance (see e.g., [7], [8], [17] and also [9], [10]).

Various optimizations of the DUR scheme are possible. E.g., in Postgres-R [18], read-sets of update transactions are not broadcast but an extra communication phase is required to broadcast the decision regarding committing or restarting a transaction. In our model and implementation, we used *multiversioning* [19]—a more general optimization technique that enabled us to optimize read-only requests. Multiversioning allows for multiple versions of transactional objects. However, each transaction has access to only one version of an object. Object versions are immutable, thus they can be accessed concurrently without any synchronization.

There exist many other replication schemes that build on the basic SMR and DUR schemes that we modelled but differ among themselves in a number of ways, e.g., they use speculative executions or explore other models of data distribution and failure. Below we give example references to the most recent work.

Romano, Palmieri, Quaglia, Carvalho, and Rodrigues [20] (see also [21]) explored speculative replication protocols for transactional systems. The key idea of their approach is to run an *optimistic atomic broadcast (OAB)* algorithm that is used to provide an early, possibly erroneous guess on transactions' serialization order and to determine the actual order.

Marandi, Primi, and Pedone [22] optimized the SMR scheme by using speculative execution to reduce the response time and state partitioning to increase the throughput of SMR. In the follow-up paper [23], the authors proposed *parallel state-machine replication (P-SMR)*, which optimizes SMR by exploiting service semantics to determine when commands can execute concurrently and when serial execution is needed.

In [24], Arun, Hirve, Palmieri, Peluso, and Ravindran observed that in DUR even in case when remote transactions rarely conflict with each other, the conflicts among local transactions (on the same replica) can significantly decrease

performance. They explored speculation to optimize this scenario and prevent some local transactions from aborting each other.

Sciascia, Pedone, and Junqueira [25] proposed *scalable deferred update (S-DUR)* aimed at increasing scalability of DUR through optimizing the execution of update transactions. The key idea of their approach is to divide the state into logical partitions, replicate each one among a group of servers, and orchestrate the execution and termination of transactions across partitions using a 2PC-like protocol. Pacheco *et al.* [26] built on this idea to scale DUR on multicore processors. In [27], Sciascia and Pedone researched the application of DUR to geo-replicated storage systems, and also discussed two optimizations of DUR for geo-replication which explore delaying and reordering of transactions.

In [28], the authors examined tradeoffs in distributed database systems that use the primary copy approach for data replication. Under the pessimistic *two-phase locking (2PL)*, replication degraded the system performance. Under the time-stamp-based optimistic protocol and semi-optimistic protocol (a combination of the optimistic one with 2PL), replication can improve response time. Moreover, with replication, the last protocol usually yields the best performance. We study different protocols and models but draw a similar conclusion: combining two replication protocols can bring benefits.

Jiménez-Peris, Patiño-Martínez, Kemme, and Alonso [29] analyzed a scale-out factor for the *read-one write-all available (ROWAA)* replication scheme, which indicates how much of the nominal capacity of the system remains after replication has been taken into consideration. The scale-out depends on the number of replicas, the percentage of update transactions, and the cost of a remote transaction (work a site has to do on behalf of other sites, e.g. installing the updates) to the cost of a local one. These results are complementary to our analysis. In [30], they compared the ROWAA and quorum-based data replication schemes, and concluded that the former approach outperforms the latter in most cases.

Several authors proposed analytical models targeting replicated and nonreplicated transactional systems. Ciciani *et al.* [28] examined the performance trade-offs of data replication in distributed database systems, in terms of the communication overhead and delay, response time constraints, CPU power, and fraction of read-only transactions. Nicola and Jarke [31] surveyed and classified a variety of analytical performance models for distributed and replicated database systems. They also developed an analytical model of a replicated database, which focuses on the interplay between replication and communication. However, the model is limited to the primary copy replication approach. More recently, Didona *et al.* [32] proposed a system for prediction of performance of replicated in-memory transactional data grids, that relies on the joint use of analytical modeling and machine learning. The above work is in contrast to our work, which aims to estimate the key trade-offs underlying two popular replication schemes.

It is also worth to mention work on modeling the performance of atomic broadcast. Borran *et al.* [33] presented a timing analysis and quantitative comparison of selected consensus algorithms. However, they only focused on round-based consensus algorithms, so the model is not directly applicable to other consensus algorithms, like Paxos. Couceiro *et al.* [34] investigated the use of machine learning methods to predict latency of abcast. Santos and Schiper [35] analyzed the performance benefits of using pipelining and batching in Paxos. They studied analytically what are the combinations of a batch size and the number of parallel instances that maximize the system throughput.

Paxos STM, which we developed to evaluate the replication schemes, provides a fault-tolerant distributed (replicated) *transactional memory (TM)*, but its design is unique compared to other such systems. DiSTM [36] can either use a mutual exclusion protocol or a lease-based protocol to serialize concurrent transactions, both depend on a central coordinator which creates a bottleneck. In [37], DiSTM is extended with object replication using 3PC. Like our system, D2STM [38] uses an optimistic transaction certification based on abcast and multiversioning, and shared objects are replicated on all nodes. In the follow-up work [39], the leases were used to limit abort rate under high contention. However, all these systems use an *ad hoc* replication protocol built on top of a non-distributed TM while our system has been built from the ground up to experiment with DUR.

3 System Model and Properties

A *replicated process* $P = \{P_1, ..., P_N\}$ consists of N processes P_i ($i = 1..N$) running on independent servers (replicas) connected via a network. Each server has access to its own volatile memory and stable storage; the combined content of the two constitutes a *local state* S_i. $S = \{S_1, ..., S_N\}$ is a *replicated state*, where S_i is a local state of P_i. A transaction executed by process P_i can only access objects that belong to local state S_i. We assume *full replication*—all objects are replicated on every server. Each server can receive requests from *clients*. The clients are independent and do not communicate with each other directly. The only possible interaction is through the replicated service.

We assume a distributed asynchronous system: There is no central coordinator and the processes communicate solely by exchanging messages using bidirectional fair-loss links [40]. Processes may fail and messages may be lost and no upper bound on message transmission is known. The failure pattern of messages is independent from the one of processes. No assumption is made on the relative computation speeds of processes. The processes can use a suitable failure detector, implemented with extensions to the described model.

Our experimental implementations of SMR and DUR support crash recovery, i.e., a server that crashed may later restart and recover its local state (either from stable storage or, preferably, from other replicas). However, we analyze the fault-free operation, where processes do not fail. Therefore, we assume a simpler crash-stop model. In this model, a process is *faulty* if it crashes at some time during the execution. It is said to be *correct* if it never crashes and executes an infinite number of steps.

A client request executed by a server (more precisely: by some process located on a server) can be regarded as a *transaction* that can execute any legal program containing operations r(o)v and w(o)v which, respectively, read and

write a value v from/to some object o, where the object is part of the replicated state. *Read-only (RO) transactions (requests)* may only consist of read operations. *Read-write (RW) or update transactions (requests)* must contain at least one write. Transactions are *atomic*, i.e., the intermediate states of transaction execution are not visible to any other transactions. We use r, x, y to denote requests; T_x denotes a transaction spawned by request x. A request spawning a RO transaction is called a *query*.

We say that state S_i is *updated* by process P_i as the result of executing a transaction T_x, if S_i is updated with the new state of all shared objects modified by T_x and other transactions can see the modifications. A replicated state S is updated if states S_i of all non-crashed processes have been updated. Each transaction T_x executed by a non-crashed process will eventually *commit*, with a replicated state S being updated accordingly, or explicitly *abort*, leaving state S unchanged.

We say that transaction T_x *precedes* transaction T_y (denoted $T_x \prec T_y$) iff T_x has completed execution before T_y begins (on the same or other server). If neither $T_x \prec T_y$ nor $T_y \prec T_x$, then T_x and T_y are *concurrent*. A transaction T_x (request x) *conflicts* with some concurrent but already committed transaction T_y (request y), if T_x can read any object modified by T_y. To prevent inconsistent reads, *conflicting transaction* T_x is rolled back and reexecuted.

We define *transactional replication* by a set of properties that describe the handling of requests by a replicated process (**R1-R7**) and the interaction with clients (**C1-C4**). We require two properties for ordering of state updates (**R6**, **R7**) and one property for ordering messages between the clients and the replicated process (**C4**). We use the symbol $\rightarrow$ to denote a *happened-before relation* [1]. We regard serializability as a safety property. In this paper, we assume that all SMR and DUR protocols guarantee these properties.

Properties of a replicated process P:

R1: *Validity:* If a process P_j modifies object o with v and state S_j is updated with the modification, then some process P_i ($i = j$ or $i \neq j$) has executed an operation w(o)v as part of some transaction that commits.

R2: *Termination:* If a transaction T_x commits, the local state of every non-crashed process is updated with the new state of all objects modified by T_x.

R3: *Integrity:* No process updates its local state twice as the result of executing a transaction T_x.

R4: *Agreement:* No two non-crashed processes update their local state differently as the result of executing a transaction T_x.

R5: *Atomicity:* All updates of the local states of all non-crashed processes, which stem from the execution of a transaction T_x, are performed atomically (not partially).

R6: *Local order:* No process P_i updates state S_i as the result of request r_2 unless P_i has already updated S_i as the result of any update request r_1, such that $r_1 \rightarrow r_2$.

R7: *Global order:* Let r_1 and r_2 be any two requests. Let P_i and P_j be any two processes that update state as the result of r_2. If P_i updates state on r_1 before r_2 then P_j updates state on r_1 before r_2.

Properties of client-P interaction:

C1: *Validity:* If a correct client sends a request r to a correct process P_i then replicated process P executes T_r and eventually returns the response to r to the client.

C2: *No creation:* If a request r is handled by some process P_i, then r was previously sent by some client.

C3: *No duplication:* No response is delivered more than once.

C4: *Causal order:* Let r_1 and r_2 be any two requests such that $r_1 \rightarrow r_2$. If res_1 and res_2 are responses to r_1 and r_2, respectively, which have been delivered to the client, then res_1 is delivered before res_2.

In the SMR approach, no transactions are considered. The SMR protocol is used to synchronize *commands* of a replicated state machine. But if we regard a command as a single-op transaction, then the above properties hold since all commands are processed serially by every process, and the protocol ensures that command execution always succeeds (provided enough replicas exist). This notion can be extended to an arbitrary (but deterministic) computation, with important restrictions. In DUR, transactions can be aborted (and possibly restarted). In SMR, rollback is not possible and transactions can only commit. In DUR, concurrent transactions may conflict since they are executed optimistically. In SMR, transactions never conflict. In DUR, state is updated on commit while in SMR objects are modified in place, so state is updated immediately.

4 Performance Model

4.1 Definitions

We can identify four phases in SMR and DUR protocols:

1) Sending a client request,
2) Replica coordination (in SMR only), $t_{rc} = t_r$,
3) Request CPU processing, $e_o = e + t_o$,
4) Agreement coordination (in DUR only), $t_{ac} = t_u$,
5) Sending a system response (or answer).

The request CPU processing time has two components: e is the time of executing the transaction code only, and t_o is the overhead time of a DUR scheme, excluding the time of network communication required for abcast; in SMR, $t_o = 0$ by definition of t_o. For simplicity, we assume that the time e is the same for all requests and all processes. Then, we have a total time $e_o = e + t_o$. We use t_r and t_u to denote the mean times of an atomic broadcast in SMR and DUR, respectively. The values t_r and t_u depend on the size of transmitted data and can differ significantly.

Then, the total time T^{rs} of processing n given requests by a replicated service rs equals $\mathcal{P}(n, e_o, t_{\{rc,ac\}})$, where function $\mathcal{P}$ depends on the parallelism enabled by replication scheme and the underlying execution environment. $n = n_q + n_{rw}$, where n_q and n_{rw} denote correspondingly, the number of read-only requests (queries) and read-write (update) requests.

In the model, we assume c processor cores per server, and abstract away any hardware restrictions on parallel executions. For simplicity, we allow at most c concurrent threads on each server at any time, with no thread interleaving. Thus, in our analysis, we can take as granted that

transactions are indivisible. We also assume that the threads do not wait for replies to be delivered to the clients to start new transactions. All servers are assumed to be interconnected via a perfect, reliable network. In the following sections, we will deduce lower bound T_{lowb} on time T^{rs}, which corresponds to processing n given requests by SMR or DUR systems *without any delay*.

The abcast protocol is optimized using batching and pipelining, which bring performance benefits (see e.g., [35]). *Batching* means broadcasting a batch of messages (if available) by only one protocol instance. *Pipelining* allows the protocol leader to initiate several instances of the protocol in parallel (as in [11]). In our model, we use β_1 and β_2 to denote respectively, the number of messages broadcast per protocol instance (*batch size*), and the maximal number of concurrent instances of abcast at a time. Then, $\beta = \beta_1\beta_2$. We assume that the bandwidth of a computer network is large enough for β.

4.2 State machine replication: SMR and LSMR

We model a service (required to be deterministic) as a *deterministic state machine* processing client requests, and use the SMR scheme to replicate the state machine on N c-core servers. In such a system:

1) each client can send each request to any non-faulty server (replica) P_i ($i = 1..N$), which then atomically broadcasts the request in a group of all replicas for *replica coordination*[1], where $t_{rc} = t_r$,
2) all client requests are executed sequentially and in the same order by every replica, with no additional overhead (so $t_o = 0$),
3) while a request is being transmitted to other replicas by a replica P_i, replica P_i can execute other requests in parallel.

The above rules hold for the basic SMR scheme which does not recognize the type of requests. By incorporating the readers/writers locks optimization, we get *SMR with Locks* (*LSMR*), where RO requests can access a consistent snapshot of object versions with no need for inter-node synchronization. To describe LSMR, we restrict rules 1 and 2 to RW requests only, and add new rules:

4) request types (RO or RW) are known *a priori*,
5) each RO request can be sent by a client to any non-faulty server (replica) which then executes it locally, in parallel with any other RO requests (no replica coordination is required),
6) RO and RW requests are processed by concurrent threads of replica P_i within a critical section guarded by the *shared/exclusive* (or *readers/writers*) *locks* [41]. Thus, we allow multiple threads to read shared local state S_i concurrently, but a thread modifying S_i must do so when no other thread is accessing S_i. In effect, a RW transaction is executed in isolation and RO transactions—in parallel.

The optimized SMR is nontrivial, as follows. Consider, e.g., two single-op requests executed by a replicated state machine in the following order: the first request writes object o and the second one reads o. SMR ensures that the read will see o's update. In the optimized SMR, however, RO requests are not globally ordered, so the read may not see the update. To guarantee the causal order defined by rule **C4**, additional machinery is required, which however does not impact the lower bound estimation. Essentially, clock values are piggybacked on requests and responses, so the order can be established by delaying some actions (see e.g., [42] for details).

Note that LSMR cannot ensure linearizability, since it is not possible to construct a sequential history that is correct according to the sequential definition of replicated objects. Consider an object o that is updated first with v on replica P_1 and then with z on replica P_2. Next, o is correctly read on P_2, giving z. Next, o is read on P_1. Since RO and RW transactions are not synchronized, the 2nd update on P_1 may occur later than the 2nd read, so v can be returned instead of z, which is at odds with the sequential definition of the object.

Below we compute the lower bounds on the total time of processing n requests, assuming, for simplicity, that $\frac{n}{\beta} = \lceil \frac{n}{\beta} \rceil$ in SMR, and $\frac{n_{rw}}{\beta} = \lceil \frac{n_{rw}}{\beta} \rceil$ in LSMR.

4.2.1 Lower bound for SMR

In the best case, the system exploits as much concurrency as possible, so that the replica coordination phase and the execution of requests proceed in parallel. Then, the lower bound on the time of processing n given requests is twofold, depending on which of the parallel parts will last longer (expressed by a function $max(a, b)$ which returns a if $a \geq b$, or b if $a \leq b$):

$$T^{\text{SMR}}_{lowb} = max(\frac{n}{\beta}t_r, ne) + \delta^{\text{SMR}} = max(\frac{t_r}{\beta}, e)n + \delta^{\text{SMR}} \quad (1)$$

$$\text{where} \quad \delta^{\text{SMR}} = \begin{cases} \beta e & \text{if } max(\frac{t_r}{\beta}, e) = \frac{t_r}{\beta} \\ t_r & \text{if } max(\frac{t_r}{\beta}, e) = e\ . \end{cases} \quad (2)$$

If $\frac{t_r}{\beta} \geq e$, so $\delta^{\text{SMR}} = \beta e$, we say that the *abcast time is dominant* (e.g., the network is slow or the requests are short). If $e \geq \frac{t_r}{\beta}$, so $\delta^{\text{SMR}} = t_r$, we say that the *execution time is dominant* (e.g., the network is fast or the requests are long). We illustrate both cases for $N = 3$, $n = 2$, and unoptimized abcast ($\beta = 1$) in Fig. 1.

4.2.2 Lower bound for LSMR

In the optimized SMR, a RO request is processed only by one server (replica) and is not broadcast to other servers. As before, execution of requests and broadcasting of RW requests are independent, so they can occur in parallel. On multi-core processors ($c > 1$), a RO request can be processed in parallel with other RO requests but serially with respect to RW requests.

a) If each server has only one CPU core ($c = 1$), the lower bound on the time of processing n requests is:

$$T^{\text{LSMR}}_{lowb} = t^{\text{SMR}}_{rw} + \Delta, \quad \text{where} \quad (3)$$

$$t^{\text{SMR}}_{rw} = max(\frac{n_{rw}}{\beta}t_r, n_{rw}e) + \delta^{\text{SMR}} = max(\frac{t_r}{\beta}, e)n_{rw} + \delta^{\text{SMR}}\ .$$

1. In systems using leader-based protocols, such as Paxos, the server sends the request to a current leader which then broadcasts it.

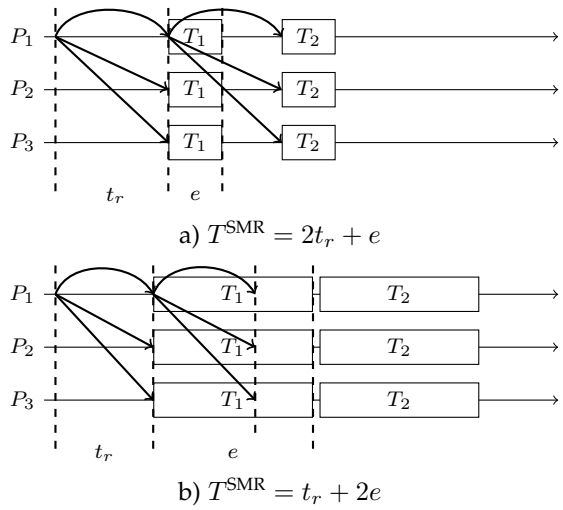

Fig. 1. SMR with abcast time dominance (a) and execution time dominance (b), where $N = 3$, $\beta = 1$, and $n = 2$.

We estimate Δ as follows. First, we assume that RO requests are executed by any free server that currently does not execute any RW request. Those RO requests which are processed in parallel with message broadcasting take $t_A = t_{rw}^{\text{SMR}} - n_{rw}e$ time units and can be neglected since they do not increase the total time of processing RW requests. Then, Δ is the time of processing by N single-core servers any remaining RO requests after all RW requests have been executed, or $\Delta = 0$ if none such a request exists:

$$\Delta = \begin{cases} \Delta' & \text{if } \Delta' > 0 \\ 0 & \text{if } \Delta' \leq 0 \end{cases} \tag{4}$$

$$\text{where } \Delta' = \left\lceil \frac{n_q}{N} \right\rceil e - t_A = \left\lceil \frac{n_q}{N} \right\rceil e - (t_{rw}^{\text{SMR}} - n_{rw}e)$$
$$= \begin{cases} (\lceil \frac{n_q}{N} \rceil + n_{rw} - \beta)e - \frac{n_{rw}}{\beta}t_r & \text{if } \frac{t_r}{\beta} \geq e \\ \lceil \frac{n_q}{N} \rceil e - t_r & \text{if } e \geq \frac{t_r}{\beta} \end{cases}.$$

b) If each server has c CPU cores ($c \geq 1$), the lower bound on the time of processing n requests is:

$$T_{lowb}^{\text{LSMR}} = t_{rw}^{\text{SMR}} + \Pi \tag{5}$$

where Π is estimated as follows. Let us assume that RO requests that before were executed on single core servers for Δ time units are now executed on $c-1$ extra cores that each server has at its disposal. This takes $t_B = \frac{\Delta}{c-1}$ time units. A certain number of RO requests are executed in parallel with abcast for t_A time units. Then, Π is the time of processing any other RO requests by all available c cores of each server, or $\Pi = 0$ if none exists:

$$\Pi = \begin{cases} \lceil \Pi' \rceil & \text{if } \Pi' > 0 \\ 0 & \text{if } \Pi' \leq 0 \end{cases} \tag{6}$$

$$\text{where } \Pi' = \frac{(t_B - t_A)(c-1)}{c} = \frac{\Delta - (t_{rw}^{\text{SMR}} - n_{rw}e)(c-1)}{c}$$
$$= \begin{cases} \frac{\Delta - (\frac{t_r}{\beta}n_{rw} + \beta e - n_{rw}e)(c-1)}{c} & \text{if } \frac{t_r}{\beta} \geq e \\ \frac{\Delta - t_r(c-1)}{c} & \text{if } e \geq \frac{t_r}{\beta} \end{cases}.$$

Note that extra cores do not make any difference for RW requests, as they are processed sequentially by each server. Note also that if $c = 1$, then as expected $\Pi = \Delta$.

Let us compare the performance of SMR and LSMR, assuming that both systems process requests optimally:

Lemma 1. *The difference in elapsed time of processing n requests without any delay by LSMR compared to SMR is:*

$$T_{diff}{}_{\text{LSMR}}^{\text{SMR}} = T_{lowb}^{SMR} - T_{lowb}^{LSMR}$$
$$= \begin{cases} \frac{n_q}{\beta}t_r - \Pi & \text{if } \frac{t_r}{\beta} \geq e \\ n_q e - \Pi & \text{if } e \geq \frac{t_r}{\beta} \end{cases}. \tag{7}$$

Note that by definition of Π, if $n_q = 0$ then $\Pi = 0$. Thus, if $n_q = 0$ then $T_{diff}{}_{\text{LSMR}}^{\text{SMR}}$ is also 0, as expected.

The time taken by an algorithm to execute on a single-core processor is called the *sequential execution time*, denoted T^{SEQ}. The execution time T_{Nc} of the corresponding parallel algorithm run on N identical c-core processors is called the *parallel execution time*. The task of processing n client requests on a single-core processor can be accomplished sequentially in $T^{\text{SEQ}} = ne$ time units. The best parallel algorithm matching our specification of LSMR takes at least $T_{Nc} = T_{lowb}^{\text{LSMR}}$ time units.

Let us compare a highly available and reliable replicated service built using the best LSMR algorithm with a non-replicated counterpart that lacks these properties. The time difference between processing n requests on N c-core servers by the replicated service and processing them sequentially by one CPU core is:

$$T_{diff}{}_{\text{LSMR}}^{\text{SEQ}} = T^{\text{SEQ}} - T_{lowb}^{\text{LSMR}} = ne - t_{rw}^{\text{SMR}} - \Pi$$
$$= ne - max(\frac{t_r}{\beta}, e)n_{rw} - \delta^{\text{SMR}} - \Pi\,. \tag{8}$$

Theorem 1. *In the best case, a service replicated using LSMR and executed on N multi-core servers ($c \geq 1$) is faster than processing requests sequentially if*

$$(n - \beta)e > \frac{n_{rw}}{\beta}t_r + \Pi \tag{9}$$

when the abcast time is dominant, and if

$$n_q e > t_r + \Pi \tag{10}$$

when the execution time is dominant.

Proof: It is straightforward by rewriting of (8). □

4.2.3 SMR and LSMR scalability

Together, a parallel system architecture and the parallel algorithm running on it constitute a *parallel system*. The *speedup (S)* obtained from a parallel system is defined as the ratio of the sequential execution time to the parallel execution time. The *efficiency (E)* of a parallel system is defined as the ratio of the speedup obtained to the total number of processor cores used. We define the *problem size* as the number of operations the best sequential algorithm executes in order to solve the problem on a single-core processor. For instance, the size of the best algorithm executing n requests sequentially is n.

For a given problem instance, the efficiency drops as the number of processors increases. A parallel system is *scalable*

n_q	n_{rw}	$Nc \rightarrow$ $n \downarrow$	2	4	8	16	32
3	1	4	**0.8**	0.57	0.36	0.21	0.11
22	2	24	0.92	**0.8**	0.63	0.44	0.28
81	3	84	0.96	0.90	**0.8**	0.65	0.47
236	4	240	0.98	0.95	0.89	**0.8**	0.66
615	5	620	0.99	0.98	0.95	0.89	**0.8**

Fig. 2. LSMR's efficiency *vs* no. of c-core processors.

if efficiency can be kept constant as the number of processors is increased, provided that the problem size is also increased [43], [44]. Then, we get

Theorem 2. *LSMR scales and SMR does not scale.*

Proof: The speedup and efficiency of a system replicated using LSMR in the best possible case are given by:

$$S^{\text{LSMR}} = T^{\text{SEQ}}/T^{\text{LSMR}}_{lowb}$$
$$E^{\text{LSMR}} = \frac{T^{\text{SEQ}}}{T^{\text{LSMR}}_{lowb}Nc} = \frac{ne}{(t^{\text{SMR}}_{rw} + \Pi)Nc} \,. \tag{11}$$

Consider $\Pi > 0$. Then we have

$$E^{\text{LSMR}}_{\Pi>0} \approx \frac{n}{(\lceil \frac{n_q}{N} \rceil + n_{rw}c)N} \approx \frac{n}{n_q + n_{rw}Nc} \,. \tag{12}$$

For a given problem instance, the efficiency of the LSMR system drops with an increasing number of processors. In order to ensure that the LSMR efficiency does not decrease as the number of processors increase, the number of RO requests should increase (see example instances in Fig. 2). Thus, if the number of RO requests is large enough, LSMR scales.

If $\Pi = 0$, then

$$E^{\text{LSMR}}_{\Pi=0} = \begin{cases} \frac{ne}{(\frac{n_{rw}}{\beta}t_r + \beta e)Nc} & \text{if } \frac{t_r}{\beta} \geq e \\ \frac{ne}{(t_r + n_{rw}e)Nc} & \text{if } \frac{t_r}{\beta} \leq e \,. \end{cases} \tag{13}$$

In both cases, the LSMR system also scales if the number of RO requests (in the numerator) will be sufficient to compensate for an increasing number of cores (in the denominator), so that efficiency is constant.

If the SMR system is unoptimized, there is no need to distinguish between RO and RW requests (i.e., $n_q = 0$, $n = n_{rw}$, $\Pi = 0$) and we get

$$E^{\text{SMR}} = \frac{T^{\text{SEQ}}}{T^{\text{SMR}}_{lowb}Nc} = \frac{ne}{T^{\text{SMR}}_{lowb}Nc} = E^{\text{LSMR}}_{\Pi=0} \text{ where } n_{rw} = n \,. \tag{14}$$

It is easy to see that the efficiency E^{SMR} cannot be maintained at a constant value when increasing the number of processors/cores since if we increase the problem size n in the numerator, then the denominator increases even more. Thus, we have proven that SMR does not scale. □

4.3 Deferred update replication: DUR and MvDUR

We model a (possibly nondeterministic) service as a *state machine* processing client requests, which is replicated on N c-core servers, and use the DUR scheme to maintain consistency of the replicated state. In such a system:

1) each client can send each request to any non-faulty server (replica), which then processes the request by executing a *local atomic transaction*,
2) transactions operate on their own (local) copies of shared objects. Shared objects are replicated on every server. On commit of a RW transaction, all replicas must agree upon the globally consistent state and update their objects accordingly. For this, the transaction's *read-set* (a set of memory locations or objects read by the transaction) and the *updates* are atomically broadcast to all servers (replicas) for *agreement coordination*, where $t_{ac} = t_u$. However, no agreement coordination is required for RO transactions, since they do not update state,
3) concurrent transactions are executed optimistically with no locking. To avoid inconsistencies, before a transaction executes a read operation and if it wrote to any objects then also on transaction commit, the transaction must be *certified*—i.e., checked if it does not conflict with any other concurrent transactions that have committed. If certification fails the transaction is aborted and reexecuted. Otherwise, the transaction finally commits and all replicas update their state accordingly (this operation and the final certification are done atomically).

By certifying transactions on every read, conflicts are detected as soon as possible, so a conflicting transaction can be aborted before its completion. For simplicity, we ignore this in the model and assume a uniform CPU execution time: $e_o = e + t_o$ for RW and $e'_o = e + t'_o$ for RO transactions, where t_o/t'_o are the mean overhead times per transaction of certification and housekeeping operations (e.g., creating/removing object copies, collecting read-sets and updates) that are specific for RW/RO transactions. t_o also includes the mean time of state update of all RW transactions (taking 0 for aborted ones).

By incorporating multiversioning, we obtain *DUR with Multiversioning* (*MvDUR*) that requires additional rules:

4) on a transaction T_x's commit, an immutable version of each object that was modified by T_x is created, to be accessed atomically by any future transactions. Each transaction T_y can only access one version of a given object—the last one before T_y commenced. Thus, each read operation returns consistent state, so it does not require certification. A transaction is certified on commit, if its updates are not null,
5) RO transactions never conflict and always commit, so do not need certification; if they are known *a priori*, the overhead is null, thus $t'_o = 0$, so $e'_o = e$.

The DUR and MvDUR algorithms are quite subtle (see e.g., [42]), so our performance model only approximates their behavior and hides details. E.g., the updates of several transactions delivered by a single abcast are applied to local state of every replica sequentially. However, since updating can proceed concurrently with the abcast and with the execution of other transactions that can commence before the state update is finished, we do not clutter the model with the sequential part, and assume that e_o is the mean CPU time of executing a RW transaction (excluding abcast

time), which also includes any overhead on a local site and on remote sites. Also, concurrent threads must occasionally synchronize access to some shared variables, but it is a very short time relatively, so we can neglect it in our analysis.

Let K be the total number of conflicts due to processing n requests (transactions). Then, K is also the number of transaction reexecutions caused by the conflicts. K depends on the number of concurrent transactions trying to modify and read the same objects. Note that write-only transactions cannot conflict according to the definition in §3[2]. K cannot be statically predicted in case of optimistic concurrency control. However, we can estimate the upper bound, as follows. If n RW transactions are executed concurrently, the number of conflicts cannot be greater than $(n-1)+(n-2)+\ldots+1=\frac{(n-1)n}{2}$.

Below we derive the lower bound on the total time of processing n requests by DUR. We first assume the unoptimized DUR, no conflicts ($K=0$), and no read-only transactions ($n=n_{rw}$). Then, we include conflicts. Finally, we extend the model with read-only transactions and analyze the optimized DUR and conflict patterns.

4.3.1 Lower bound for DUR

In the best case, transactions are executed in parallel by all Nc processor cores and concurrently with agreement coordination. Thus, we obtain the lower bound on the time of processing n given requests as:

$$T_{lowb}^{\text{DUR}} = max(t_a^{\text{DUR}}, t_e^{\text{DUR}}) \tag{15}$$

where t_a^{DUR} is attributed (at large) to abcast time, while t_e^{DUR} is attributed (at large) to CPU time, as given below. If $t_a^{\text{DUR}} \geq t_e^{\text{DUR}}$ then we say that the *abcast time is dominant*. If $t_a^{\text{DUR}} < t_e^{\text{DUR}}$ then the *execution time is dominant*.

In the simplest case, when there are no conflicts and no RO requests ($K=n_q=0$) and the abcast protocol is not optimized, so it can broadcast only one message at a time ($\beta=1$), the lower bound on processing at least one RW request ($n_{rw}>0$) is as follows:

$$T_{lowb}^{\text{DUR}_{\beta=1}^{K=n_q=0}} = max(e_o + n_{rw}t_u, \left\lceil \frac{n_{rw}}{Nc} \right\rceil e_o + t_u)\ . \tag{16}$$

In Fig. 3a-c., we show example executions for $N=2$, $c=1$, $\beta=1$, and $n_{rw}=3$, where abcast dominance is illustrated in a-b and execution dominance in c.

In practical systems the abcast protocol is optimized: β_1 messages are broadcast in one instance and at most β_2 instances of the protocol are executed in parallel. In our model, we assume that $0<\beta_1\leq Nc$ and[3] β_2 is some natural number, s.t. $\beta=\beta_1\beta_2\geq Nc$. Then, we have:

$$\begin{aligned} T_{lowb}^{\text{DUR}^{K=n_q=0}} &= max(t_a^{\text{DUR}}, t_e^{\text{DUR}}), \quad \text{where} \\ t_a^{\text{DUR}} &= e_o + \left\lceil \frac{n_{rw}}{\beta} \right\rceil t_u \\ t_e^{\text{DUR}} &= \left\lceil \frac{n_{rw}}{Nc} \right\rceil e_o + t_u\ . \end{aligned} \tag{17}$$

2. In object-based TM, e.g. Paxos STM, write-only transactions appear as read-write transactions since they usually modify only a subset of object fields but the whole object is read and replaced on commit.

3. Note that $\beta_1 > Nc$ is not desirable since abcast would be delayed, causing objects to be updated less frequently, so increasing likelihood of conflicts and thus downgrading the optimization.

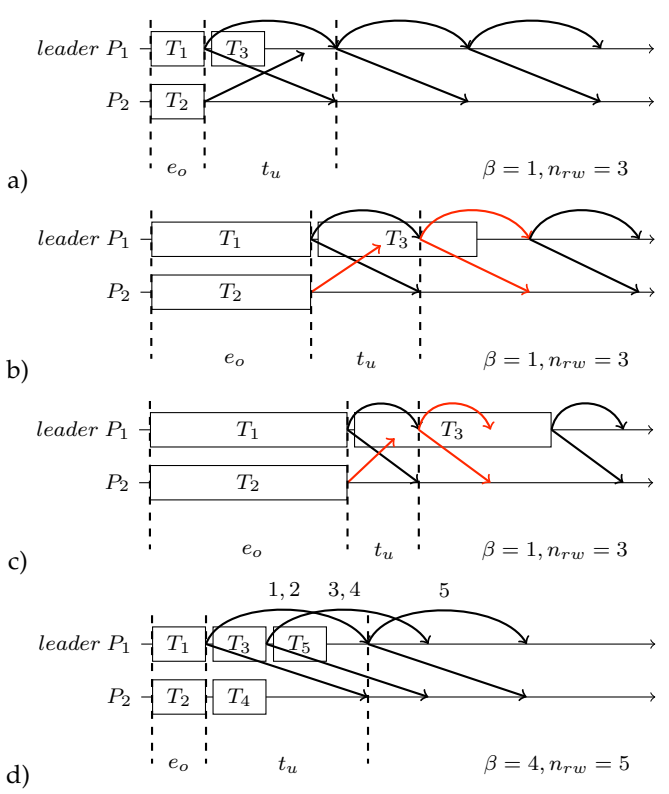

Fig. 3. DUR with abcast time dominance (a-b, d) and execution time dominance (c), where $N=2$ and $c=1$.

We illustrate case when $t_a^{\text{DUR}} \geq t_e^{\text{DUR}}$ (abcast dominance) in Fig. 3d, where we show an example execution for $N=2$, $c=1$, $\beta=4$, and $n_{rw}=5$. Note that if a network had greater capacity ($\beta\geq 5$), the execution time is dominant. E.g., if $\beta=5$ then $t_a^{\text{DUR}}=e_o+t_u<t_e^{\text{DUR}}=3e_o+t_u$.

If K transactions conflict, then each of them is reexecuted. Thus, from (17), we have:

$$T_{lowb}^{\text{DUR}^{K\geq 0, n_q=0}} = max(e_o + \left\lceil \frac{n_{rw}+K}{\beta} \right\rceil t_u, \left\lceil \frac{n_{rw}+K}{Nc} \right\rceil e_o + t_u)\ . \tag{18}$$

Let us now consider RO transactions. In case of the unoptimized DUR, RO transactions may conflict, so require certification, but do not update state, so do not require agreement coordination. We use e'_o to denote the mean CPU time of executing a RO request, where $e'_o < e_o$, and we have $t_u=0$ for RO requests. If the number of conflicts for RW and RO transactions is respectively, K_{rw} and K_q (where $K_{rw}+K_q=K$), then we can approximate the lower bound by extending (18) as follows:

$$\begin{aligned} T_{lowb}^{\text{DUR}^{K\geq 0, n_q\geq 0}} &= max(t_a^{\text{DUR}}, t_e^{\text{DUR}}), \quad \text{where} \\ t_a^{\text{DUR}} &= e_o + \left\lceil \frac{n_{rw}+K_{rw}}{\beta} \right\rceil t_u \\ t_e^{\text{DUR}} &\approx \left\lceil \frac{n_{rw}+K_{rw}}{Nc} \right\rceil e_o + \tau_1 \\ \tau_1 &= max(t_u, \left\lceil \frac{n_q+K_q}{Nc} \right\rceil e'_o)\ . \end{aligned} \tag{19}$$

Note that if $\lceil (n_{rw}+K_{rw})/Nc \rceil = (n_{rw}+K_{rw})/Nc$, then equation (19) is not approximate but exact.

4.3.2 Lower bound for MvDUR

In MvDUR, read-only transactions never conflict ($K_q = 0$) and do not require certification, so we approximate their execution time e' as $e' \approx e$. By modifying (19) we then obtain the lower bound for MvDUR:

$$\begin{aligned} T_{lowb}^{\text{MvDUR}} &= max(t_a^{\text{MvDUR}}, t_e^{\text{MvDUR}}), \quad \text{where} \\ t_a^{\text{MvDUR}} &= e_o + \left\lceil \frac{n_{rw} + K_{rw}}{\beta} \right\rceil t_u \\ t_e^{\text{MvDUR}} &\approx \left\lceil \frac{n_{rw} + K_{rw}}{Nc} \right\rceil e_o + \tau_2 \\ \tau_2 &= max(t_u, \left\lceil \frac{n_q}{Nc} \right\rceil e) \ . \end{aligned} \tag{20}$$

If $t_u \geq \lceil \frac{n_q}{Nc} \rceil e$ (so $\tau_2 = t_u$) then we say that the *abcast dominates RO requests*. If $t_u < \lceil \frac{n_q}{Nc} \rceil e$ (so $\tau_2 = \lceil \frac{n_q}{Nc} \rceil e$) then the *RO requests dominate abcast*.

It is important to emphasize that the main advantage of MvDUR when compared to the unoptimized DUR is not much due to the fact that $e < e'_o$ but because $K_q = 0$. In MvDUR-based systems, there are less conflicts (thus less transaction rexecutions), which greatly decreases the total time of processing n concurrent requests.

Let us compare the performance of DUR and MvDUR:

Lemma 2. *If the abcast time is dominant, $t_a^{DUR} = t_a^{MvDUR}$, so there is no difference between DUR and MvDUR. Otherwise, the difference between the elapsed time of processing n given requests without any delay by DUR compared to MvDUR is:*

$$T_{diff}{}_{MvDUR}^{DUR} = T_{lowb}^{DUR} - T_{lowb}^{MvDUR} = t_e^{DUR} - t_e^{MvDUR} = \tau_1 - \tau_2 \ . \tag{21}$$

Proof: Straightforward from (20) and (19). □

Thus, if the abcast dominates RO requests, $T_{diff}{}_{\text{MvDUR}}^{\text{DUR}}$ is equal 0 since $\tau_1 = \tau_2 = t_u$, which is as expected since the performance gain due to optimizing RO requests is counteracted by the high cost of broadcasting updates of the last update request. If RO requests dominate abcast, $T_{diff}{}_{\text{MvDUR}}^{\text{DUR}} = \lceil \frac{n_q + K_q}{Nc} \rceil e'_o - \lceil \frac{n_q}{Nc} \rceil e \approx \frac{K_q}{Nc} e$ if $e'_o \approx e$.

In practice, the time e_o of processing an RW request in MvDUR is shorter than in DUR, as shared objects can be implemented more efficiently (with no locks). Therefore, $T_{diff}{}_{\text{MvDUR}}^{\text{DUR}} > \tau_1 - \tau_2$ if the execution time is dominant, and $T_{diff}{}_{\text{MvDUR}}^{\text{DUR}} > 0$ if the abcast time is dominant.

4.3.3 Analysis of transaction conflicts

Now let us analyze the behavior of MvDUR for specific conflict patterns, assuming the worst possible case. In the worst case, all concurrent transactions conflict with each other and there are no RO transactions. Thus, we can use (18) to compute the lower bound on processing n given requests, where $n_{rw} = n$. We can also derive the lower bound for a particular conflict pattern.

We consider two conflict patterns, assuming at most c concurrent transactions on each server.

a) Transactions never wait for conflicts to be resolved:

In this case, only the first transaction commits and the remaining $Nc - 1$ transactions abort and repeat execution in parallel with a new fresh transaction that is processed by a free core. This process repeats until the last new fresh transaction appears. Then, again one transaction commits and all but one are reexecuted, and so on until the last transaction is executed.

Thus, the time of processing n given requests assuming that the execution time is dominant:

$$\begin{aligned} t_e^{\text{DUR}}(a) &= e_o + (n - Nc)e_o + (Nc - 1)e_o + t_u \\ &= ne_o + t_u \ . \end{aligned} \tag{22}$$

The number of conflicts is

$$\begin{aligned} K_a &= (n - Nc)(Nc - 1) + \frac{(Nc - 1)Nc}{2} \\ &= n(Nc - 1) + \frac{Nc - (Nc)^2}{2} \ . \end{aligned} \tag{23}$$

b) Transactions wait until conflicts are resolved:

In this case, we execute the first Nc transactions in parallel. Since they all conflict, we wait with processing the next batch of Nc transactions until all conflicts are resolved and the conflicting transactions are reexecuted. We repeat this process until all transactions are executed. Since in the last iteration the number of fresh new transactions can be smaller than Nc, we use μ to describe their time of execution.

Thus, the time of processing n given requests assuming that the execution time is dominant:

$$\begin{aligned} t_e^{\text{DUR}}(b) &= \left(\left\lfloor \frac{n}{Nc} \right\rfloor + (Nc - 1) \left\lfloor \frac{n}{Nc} \right\rfloor + \mu \right) e_o + t_u, \\ &\quad \text{where } \mu = n - \left\lfloor \frac{n}{Nc} \right\rfloor Nc \\ &= ne_o + t_u \ . \end{aligned} \tag{24}$$

The number of conflicts is

$$\begin{aligned} K_b &= \frac{(Nc - 1)Nc}{2} \left\lfloor \frac{n}{Nc} \right\rfloor + \frac{(\mu - 1)\mu}{2} \\ &= \frac{(Nc)^2}{2} \left\lfloor \frac{n}{Nc} \right\rfloor + \frac{\mu^2 - n}{2} \approx \frac{n(Nc - 1)}{2} \ . \end{aligned} \tag{25}$$

Thus, if the execution time is dominant then the lower bound is the same for both conflict patterns. However, the number of conflicts differ. If $n \geq Nc$ then $K_b < K_a$, which means that in case b less processor power is used.

If the abcast time is dominant, the time of processing n given requests for conflicts patterns a and b is, by (20):

$$t_a^{\text{DUR}} = e_o + \left\lceil \frac{n_{rw} + K}{\beta} \right\rceil t_u \ . \tag{26}$$

where $K = K_a$ or $K = K_b$, respectively. Since typically $n \geq Nc$, so $K_b < K_a$, thus it takes more time to complete the processing in case of conflict pattern a. Thus, one way to optimize abcast dominant DUR systems is to *delay* transaction processing in order to reduce the number of conflicts (transaction reexecutions).

4.3.4 *MvDUR* vs. *sequential execution*

Compared to sequential request processing, an MvDUR system has the advantage of replication which makes it possible to tolerate machine failures and distribute workload evenly since each transaction can be processed on any replica. However, neither system outperforms the other in terms of performance in every case.

Let us assume that the execution time is dominant, so $T_{lowb}^{\text{MvDUR}} = t_e^{\text{MvDUR}}$. Then an MvDUR system is likely to outperform sequential execution since Nc transactions

are processed in parallel. The time of processing n given requests sequentially is $T^{\text{SEQ}} = ne$. To compute t_e^{MvDUR}, the number of transactions is divided by Nc but it is also enlarged by the number of conflicts (20). If $Nc = 1$, then no conflicts can occur and $t_e^{\text{MvDUR}} = n_{rw}e_o + \tau_2$, which gives $n_{rw}e_o + t_u$ if abcast dominates RO requests, and $n_{rw}e_o + n_q e \approx ne$ otherwise (assuming $e \approx e_o$). On the other hand, if we assume that $Nc > 1$ and *all* transactions conflict, then $t_e^{\text{MvDUR}} = t_e^{\text{MvDUR}}(a|b) = n_{rw}e_o + t_u$ (since $n_q = 0$). Thus, we can stipulate that on average the modelled MvDUR system is never much slower than its sequential counterpart if $e_o \approx e$, and can be faster if the number of processors/cores is large enough.

If the abcast time is dominant, so $T_{lowb}^{\text{MvDUR}} = t_a^{\text{MvDUR}}$, then an MvDUR system can be faster than its sequential counterpart only if the number of processors/cores is large enough *and* the abcast time t_u is short enough.

4.3.5 MvDUR scalability

Note that the number of transaction conflicts may grow with the number of RW requests and processors/cores (since more transactions are processed in parallel). We say that the number of conflicts *explodes* for a parallel system if it grows at least linearly with the problem size (n). E.g., K_a and K_b explode.

Below we consider scalability of a replicated system in the best possible case, separately for the execution and abcast time dominance. Then, we get

Theorem 3. *When the execution time is dominant, MvDUR scales if the number of conflicts does not explode.*

Proof: The best possible parallel algorithm matching the specification of MvDUR takes at least $T_{Nc} = T_{lowb}^{\text{MvDUR}}$ time units. Thus, by (20), if the execution time is dominant, the speedup and efficiency of an MvDUR system in the best possible case are, respectively:

$$S^{\text{MvDUR}} = T^{\text{SEQ}}/t_e^{\text{MvDUR}}$$
$$E^{\text{MvDUR}} = \frac{T^{\text{SEQ}}}{t_e^{\text{MvDUR}}Nc} \approx \begin{cases} \frac{ne}{\sigma + t_u Nc} & \text{if } t_u \geq \lceil \frac{n_q}{Nc} \rceil e \\ \frac{ne}{\sigma + n_q e} & \text{if } t_u \leq \lceil \frac{n_q}{Nc} \rceil e \end{cases} \quad (27)$$
$$\text{where } \sigma = (n_{rw} + K_{rw})e_o \ .$$

In case when $e \approx e_o$, efficiency can be approximated to

$$E^{\text{MvDUR}} \approx \begin{cases} \frac{n}{n_{rw} + K_{rw} + \frac{t_u Nc}{e}} & \text{if } t_u \geq \lceil \frac{n_q}{Nc} \rceil e \\ \frac{n}{n + K_{rw}} & \text{if } t_u \leq \lceil \frac{n_q}{Nc} \rceil e \ . \end{cases} \quad (28)$$

Thus, if abcast dominates RO requests (i.e., $t_u \geq \lceil \frac{n_q}{Nc} \rceil e$), then the MvDUR-based system scales if the number of requests n is large enough to compensate for an increasing number of cores and the number of conflicts does not explode. If RO transactions predominate ($t_u < \lceil \frac{n_q}{Nc} \rceil e$), then the system scales perfectly if the number of conflicts does not explode. □

For example, if $K_{rw} = K_b$, then the number of conflicts explodes and $E^{\text{MvDUR}}_{t_u \leq \lceil \frac{n_q}{Nc} \rceil e} \approx \frac{2n}{2n_q + n_{rw}(1+Nc)}$, i.e., the system does not scale since the efficiency cannot be maintained at a constant value by simultaneously increasing the number of processors (or cores) and the size of the problem. E.g., if $n_q = 0$, then the efficiency is $\frac{2}{1+Nc}$.

Theorem 4. *When the abcast time is dominant, MvDUR scales worse than when the execution time is dominant, and it does not scale if all requests are updating.*

Proof: In case of abcast time dominance, the speedup and efficiency of the MvDUR system in the best possible case are, respectively:

$$S^{\text{MvDUR}} = T^{\text{SEQ}}/t_a^{\text{MvDUR}}$$
$$E^{\text{MvDUR}} = \frac{T^{\text{SEQ}}}{t_a^{\text{MvDUR}}Nc} = \frac{ne}{(e_o + \lceil \frac{n_{rw}+K_{rw}}{\beta} \rceil t_u)Nc} \quad (29)$$

Note that in the denominator Nc is multiplied by the number of RW requests and conflicts which are variables that are likely to grow in proportion to the problem size. Thus, the system scales worse than when the execution time was dominant, since in the latter case in the denominator Nc was either multiplied by a constant value t_u or the efficiency did not depend on Nc at all, so it was easier to maintain efficiency at a constant value when increasing the number of processors/cores. However, if the number of RW requests is constant and the number of conflicts does not explode, then if the abcast time is dominant, the MvDUR system can scale for RO requests. But if $n_q = 0$, then none increase of problem size $n = n_{rw}$ is able to keep efficiency constant once the number of processor cores increases (note that K_{rw} can only grow and other parameters are constant values). □

5 Experimental Evaluation

In this section, we present the results of experimental evaluation of SMR and MvDUR under different workload types and varying contention levels, obtained using popular microbenchmarks: *Hashtable* and *Bank*. For each benchmark, we developed a non-replicated service (*SeqHashtable* and *SeqBank*) executing requests sequentially on one machine, and a replicated, fault-tolerant counterpart, where the program code and data structures (hashtable and bank accounts) were fully replicated on N nodes, each one equipped with a c-core processor. The replicated service was built using JPaxos, which utilizes the state machine replication approach, and Paxos STM, which implements the MvDUR algorithm [42].

As in the original state machine replication approach, JPaxos does not recognize requests types, so it implements the SMR scheme. Paxos STM can execute requests in parallel on multicores. Moreover, multiversioning ensures that RO transactions never conflict (so never abort), thus it implements the MvDUR scheme. Both systems use the implementation of abcast based on Paxos [11], with support of message batching and pipelining.

Both systems allow replicas to crash and later recover and seamlessly rejoin. Notably, nonvolatile storage is scarcely used during regular (nonfaulty) system operation. During recovery, a recovering replica can obtain the current state from other live replicas (if at least a majority of replicas is operational all the time). In the paper, we compared the performance of the two systems during regular (nonfaulty) execution. But in all our experiments, we ran both systems with the recovery protocol enabled, so they were fully fault-tolerant.

The *Hashtable* benchmark features a hashtable of size h, storing pairs of key and value, and accessed using the *get*, *put*, and *remove* operations. A run of this benchmark consists of a load of requests which are issued to the hashtable. As in the model, we consider two types of requests (transactions): A *read-only (RO)* request executes a series of *get* operations on a randomly chosen set of keys. A *read-write (RW)* request executes a series of *get* operations, followed by a series of update operations (either *put* or *remove*). The Hashtable is prepopulated with $\frac{h}{2}$ random integer values from a defined range, thus giving the saturation of 50%. The saturation level is preserved all the time: If a randomly chosen key points at an empty element, a new value is inserted using *put*; otherwise, the element is removed using *remove*.

We used three Hashtable configurations: Default, Prolonged, and High-Contention, which represent various *workload types*, modeled by varying the number and length of operations in RO and RW transactions. In all cases, each RO request scans a vast amount of data using 100 *get* operations. In contrary, RW requests have fewer operations (10 in Default and Prolonged and 50 in High-Contention), where 20% are update operations. In Prolonged Hashtable, each RO and RW transaction is prolonged 1 ms, which simulates a computation-heavy workload for execution time dominance.

The *Bank* benchmark features a replicated array of 250k bank accounts. A run of this benchmark consists of a load of RW and RO requests accessing the accounts. A RW request transfers money between two accounts, by executing two *get* and two *put* operations on the replicated array. A RO request computes a balance, by reading all accounts and summing up the funds.

For each benchmark, we examined three *test scenarios*, obtained by the following mix of RW and RO requests in the test load: 10%, 50%, and 90% of RW requests. RO requests were known *a priori*. In Paxos STM, different test scenarios allow us to simulate variable *contention* in the access to data shared by concurrent transactions.

We ran tests in a cluster of 20 nodes connected via 1Gb Ethernet network. Each node had 28-core Intel E5-2697 v3 2.60GHz processor, 64GB RAM, and used Scientific Linux CERN 6.7 with Java HotSpot 1.8.0.

The abcast protocol used by JPaxos and Paxos STM was configured to have at most two concurrent instances of consensus and the batch capacity 64KB. We experimentally established an optimal number of threads in Paxos STM to be 160 for Hashtable and 280 for Bank (these values were used in all our tests). In all of our tests, the number of threads is high and far exceeds the number of physical cores, to compensate for threads that are blocking on I/O (network) operations. This way we can fully exercise the hardware, and show the peak performance of JPaxos and Paxos STM, which corresponds to our model.

To reduce the overhead caused by client-server communication, the clients ran on replicas. The number of clients in JPaxos and Paxos STM was constant per replica, and equal the number of threads in Paxos STM.

5.1 Benchmark results

For all benchmarks, we present *throughput*—the number of handled requests (committed transactions) per second. In services built using Paxos STM, concurrent transactions may conflict and abort, so we also present the *abort rate*—the percentage of transactions aborted due to conflicts out of a total number of transaction executions (i.e., $\frac{K}{n+K}100\%$, where n is the number of requests and K is the number of conflicts). The abort rate gives a useful insight into the level of contention. We also measured data transfer in Mb/s to witness network congestion, as it is a limiting factor in many tests.

In Fig. 4, we give the measured results and the optimal throughput n/T_{lowb} of SMR and MvDUR for $N = 3..20$ replicas, where T_{lowb} was calculated using (1) and (20); n, n_{rw}, n_q and c are constant input data; t_r, t_u, e, e_o, and *message size* are obtained separately for each benchmark and test scenario through measurements of JPaxos and Paxos STM; β_1 = 64KB/*message size*, where 64KB is the capacity of abcast batch, and $\beta_2 = 2$. These parameters are constant in each test scenario. Only K_{rw} (the actual number of conflicts) changes with N. The abcast times t_r and t_u were measured when network was *not* congested. Below we summarize the results. See the supplemental material for more discussion.

The experimental results in Fig. 4 and 5 corroborate the model in §4. Not surprisingly, the sequential, non-replicated implementations of our benchmarks outperform JPaxos. More interestingly, Paxos STM and sequential services gave variable results. E.g., Paxos STM was the clear winner for Prolonged Hashtable (see Fig. 4b and 5b) and for Bank with 10% and 50% of RW requests (see Fig. 4d and 5d). This is credited to Paxos STM's ability of executing transactions in parallel, thus being able to fully utilize the multi-core hardware. In other cases, the sequential, non-replicated services demonstrated higher throughput. However, they are vulnerable to system failures. In contrast, replicated services can tolerate failures of machines and communication links, thus ensuring service reliability and availability.

JPaxos executes all requests sequentially, thus if the execution time is dominant, it performs poorly compared to Paxos STM which can process transactions in parallel and can scale by Theorem 3 (see Fig. 4b). Paxos STM behaves well especially in test scenarios involving many RO transactions, but suffers under high contention (evidenced by abort rate). Then, JPaxos is clearly better since it delivers predictable and stable performance (see Fig. 4c for 50% and 90% RW). However, the overhead caused by abcast, and by the network that often gets saturated, severely reduce the performance of both systems. By Theorem 4, abcast dominance can overshadow the gain of parallelism in Paxos STM and, in consequence, reduce scalability. If at least RO requests dominate abcast, then efficiency could be kept constant by decreasing the number of conflicts, e.g. by contention management.

The optimizations of abcast give Paxos STM considerable performance boost. This is especially visible in Bank (see Fig. 4d) where replicas can get a lot of transactions ready to commit at the same time, so the abcast protocol with batching can broadcast them all at once.

As expected, the calculated throughput of SMR is constant for all numbers of replicas, since by Theorem 2 the SMR scheme does not scale for any workload type. However, when SMR is abcast time dominant (Fig. 4a,c), JPaxos does not perform uniformly, and the peak throughput is

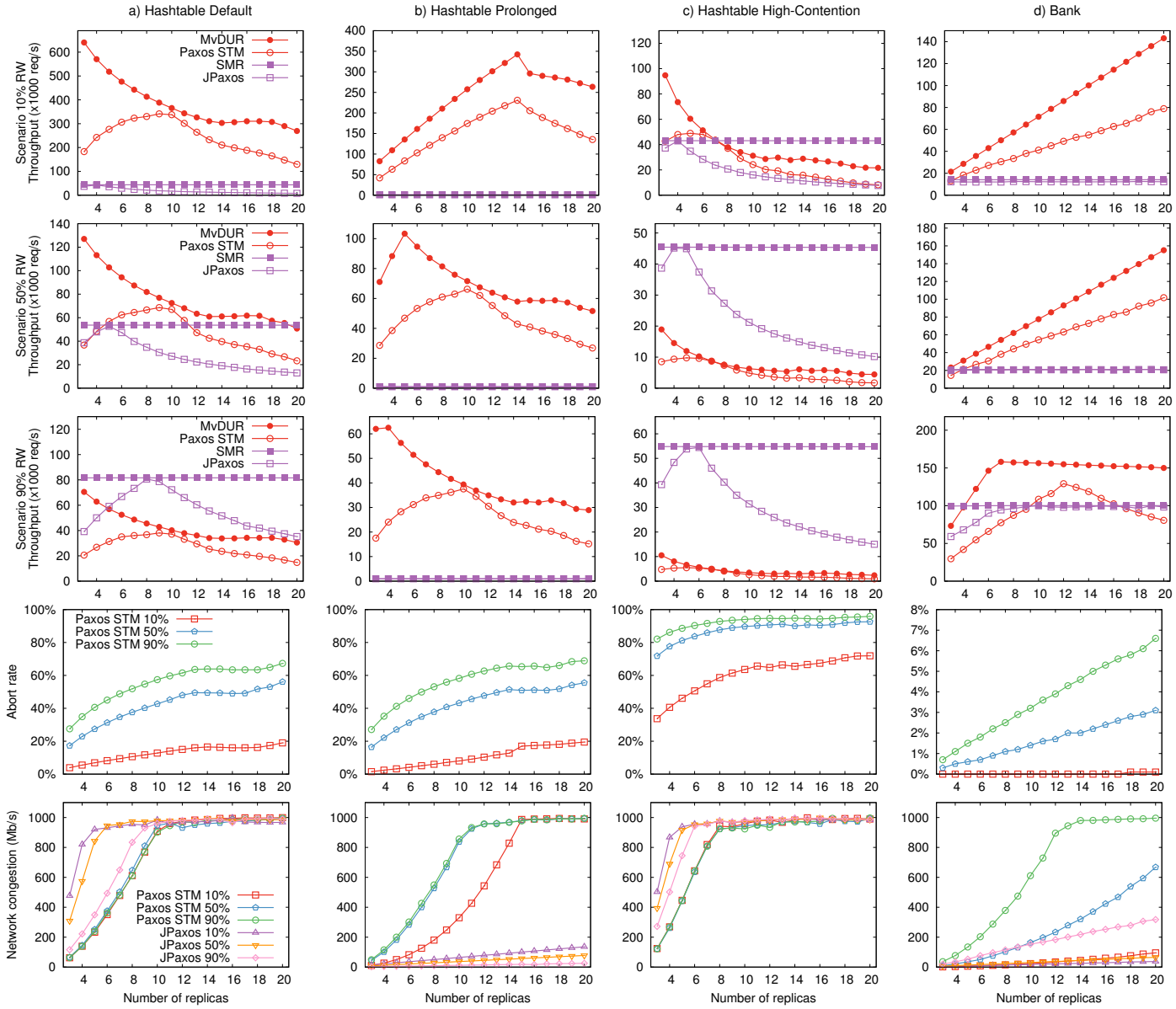

Fig. 4. Model predictions and benchmark results for 10%, 50%, and 90% of RW requests (or transactions).

Test scenario →	10% RW	50% RW	90% RW
a) SeqHashtable Default	657785	1002342	2270151
b) SeqHashtable Prolonged	994	996	998
c) SeqHashtable High-Cont.	468966	528255	650705
d) SeqBank	40477	73174	208853

Fig. 5. Throughput of sequential services (req/s).

when the network is nearly saturated (see the network congestion plot). Before that, JPaxos's throughput grows with the number of replicas. This is because with replicas are collocated clients that can now produce more requests, but still their quantity and capacity are not enough to effectively utilize the abcast protocol, i.e., the number of requests per abcast batch is lower than declared β_1 while in the model it is always equal β_1. (The network saturation occurs later in Bank where message sizes are small.) Once the network becomes saturated, the throughput is deteriorating as the cluster size increases, since threads (CPUs) are exhausted by abcasts and execute less requests. When SMR is execution time dominant (Fig. 4b,d), the throughput of SMR and JPaxos are uniform, and slightly higher for SMR, as the overhead caused by the inter-thread communication and message (un)marshalling was not modelled.

The calculated throughput of MvDUR and the measured one of Paxos STM, vary for workload types. In the abcast time dominant workloads (Fig. 4a,c, and the 90% RW in Fig. 4b,d above network saturation), it decreases with an increasing number of replicas, since the number of conflicts grows. Like in JPaxos, Paxos STM underutilizes the network when a number of replicas (clients) is small, and underperforms once the network is saturated. In the execution time dominant workloads (Fig. 4b until the network is saturated, and Fig. 4d, except 90% RW above network saturation), the results for MvDUR and Paxos STM look similar, but the measured throughput is lower due the overhead of inter-

thread communication and message (un)marshalling.

5.2 Analytical model *vs.* experimental evaluation

For all benchmarks, we measured the actual throughput of JPaxos and Paxos STM, and calculated the throughput of SMR and MvDUR using equations for lower bounds, where the values of parameters were measured in separate tests using JPaxos and Paxos STM (see the Supplemental material for details on how these values were obtained). The actual and calculated throughputs differ, which is not surprising. Our analytical model aims to estimate the upper bounds on the performance achievable by using well-known transactional replication schemes, assuming an idealized execution environment, while the measured throughput reflects the actual conditions when executing benchmarks. E.g., the network was slow compared to the computational power of our grid, and often got saturated, so the processors were underutilized when running benchmarks. On the other hand, to predict the throughput using the model, we measured all parameters when the network was not congested, and neglected some overhead (e.g. due to message (un)marshalling and thread synchronization). Moreover, in our calculation the abcast protocol was always fully utilized, so the peak performance was achieved also for a small number of nodes.

6 Combining SMR and DUR

The analytical and experimental results clearly indicate that neither replication scheme is superior. One should choose SMR or DUR depending on the expected workload. In the follow-up work [45], we proposed the *Hybrid Transactional Replication (HTR)* scheme, which combines SMR and DUR for better performance and scalability. Depending on the current workload, a HTR request can be processed either as a *state machine (SM) transaction*, or a *deferred update (DU) transaction*. SM transactions are handled as in SMR: all requests are executed sequentially and in the same order by all processes. DU transactions are executed as in DUR—optimistically and in isolation. The final certification of DU transactions and installing the updates must be serialized with the execution of SM transactions, to avoid inconsistent reads. But multiple DU transactions can execute in parallel with a single SM transaction. RO requests are always executed as DU transactions. The algorithm is in [45].

Having two modes of transaction execution has several advantages. Firstly, the system performance is improved for various workloads. CPU intensive workloads can benefit from the concurrent execution of DU transactions. On the contrary, transactions that generate large readsets, writesets or updates are better off in SM mode. This is because a SM transaction usually only requires to broadcast a reference to the code to be executed, which is far less costly than broadcasting the updates resulting from the execution of a DU transaction. Moreover, SM transactions are guaranteed to commit. Therefore SM transactions are suitable for requests that generate high contention. Secondly, HTR offers richer semantics than SMR or DUR alone, as it introduces rollback capabilities to SMR and equips DUR with the support for irrevocable operations (which cannot be rolled back).

In HTR, the programmer (or system) is free to choose the SM/DU execution mode for each transaction's run. The decision depends on the characteristics of the transaction (e.g., read-only, CPU intensive, accessed objects) and the current system load (e.g., abort rate, network saturation). The programmer specifies the desired decision rules within an *oracle*, queried before each transaction's run. Since the oracle has access to a vast number of parameters describing the system's performance, it allows for adapting to the changing workload.

7 Conclusions

In our study of SMR and DUR, we have proven several results. The key corollary is that neither replication scheme is superior in all cases. This is due to the differences between SMR and DUR in sensitivity to various workloads. Execution dominated workloads are handled much better when using DUR since this approach can execute multiple requests concurrently, contrary to classical SMR. In particular, DUR achieves higher throughput than SMR for read-write requests with a majority of read operations that do not cause conflicts (which is a typical workload of web services). However, performance gains from parallel request execution are overshadowed by high costs of atomic broadcast, which is especially visible in the abcast-dominated workloads.

As formally proven, DUR exercises the ability to scale on multicores if certain conditions are met, so one would expect that it outperforms SMR. But the overhead of the transactional machinery makes SMR a better choice in some cases. One can also observe poor efficiency of replicated services compared to their non-replicated variants, due to the high cost of abcast. This cost can be compensated in DUR by the ability to process requests in parallel. If there are few conflicts, in execution-dominated workloads, a service replicated using DUR performs much better than when not replicated. We showed that a suitable conflict pattern can reduce conflicts.

Acknowledgments

The authors would like to thank the reviewers for their useful comments. We also thank the Poznań Supercomputing and Networking Center (PSNC) for providing computing resources. This work was funded from National Science Centre funds granted by decisions No. DEC-2012/06/M/ST6/00463 and DEC-2011/01/N/ST6/06762.

References

[1] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM (CACM)*, vol. 21, no. 7, pp. 558–565, Jul. 1978.
[2] F. B. Schneider, "Replication management using the state-machine approach," in *Distributed systems (2nd Ed.)*. ACM Press/Addison-Wesley, 1993, pp. 169–197.
[3] B. Charron-Bost, F. Pedone, and A. Schiper, Eds., *Replication - Theory and Practice*, ser. LNCS. Springer, 2010, vol. 5959.
[4] 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, 1990.
[5] C. H. Papadimitriou, "The serializability of concurrent database updates," *Journal of the ACM (JACM)*, vol. 26, no. 4, pp. 631–653, Oct. 1979.
[6] J. Kończak, N. Santos, T. Żurkowski, P. T. Wojciechowski, and A. Schiper, "JPaxos: State machine replication based on the Paxos protocol," EPFL, I&C, Tech. Rep. 167765, Jul. 2011.

[7] D. Agrawal, G. Alonso, A. E. Abbadi, and I. Stanoi, "Exploiting atomic broadcast in replicated databases (extended abstract)," in *Proc. of Euro-Par '97*, Aug. 1997.

[8] B. Kemme, F. Pedone, G. Alonso, and A. Schiper, "Processing transactions over optimistic atomic broadcast protocols," in *Proc. ICDCS '99*, Jun. 1999.

[9] A. Schiper and M. Raynal, "From group communication to transactions in distributed systems," *Communications of the ACM (CACM)*, vol. 39, no. 4, Apr. 1996.

[10] J. Gray, P. Helland, P. O'Neil, and D. Shasha, "The dangers of replication and a solution," in *Proc. of SIGMOD '96*, Jun. 1996.

[11] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems (TOCS)*, vol. 16, no. 2, May 1998.

[12] P. T. Wojciechowski, T. Kobus, and M. Kokociński, "Model-driven comparison of state-machine-based and deferred-update replication schemes," in *Proc. of SRDS '12*, Oct. 2012.

[13] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: a tutorial," *ACM Computing Surveys (CSUR)*, vol. 22, no. 4, pp. 299–319, Dec. 1990.

[14] D. K. Gifford, "Weighted voting for replicated data," in *Proc. of SOSP '79*, Dec. 1979, pp. 150–162.

[15] P. A., Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency control and recovery in database systems*. Addison-Wesley, 1987.

[16] F. Pedone, R. Guerraoui, and A. Schiper, "Exploiting atomic broadcast in replicated databases," in *Proc. Euro-Par '98*, Sep. 1998.

[17] ——, "The database state machine approach," *Distributed and Parallel Databases*, vol. 14, no. 1, Jul. 2003.

[18] B. Kemme and G. Alonso, "Don't be lazy, be consistent: Postgres-R, a new way to implement database replication," in *Proc. of VLDB 2000*, Sep. 2000.

[19] P. A. Bernstein and N. Goodman, "Multiversion concurrency control—theory and algorithms," *ACM Transactions on Database Systems (TODS)*, vol. 8, no. 4, pp. 465–483, Dec. 1983.

[20] P. Romano, R. Palmieri, F. Quaglia, N. Carvalho, and L. Rodrigues, "On speculative replication of transactional systems," *Journal of Computer and System Sciences*, vol. 80, no. 1, pp. 257–276, Feb. 2014.

[21] R. Palmieri, F. Quaglia, and P. Romano, "OSARE: Opportunistic speculation in actively REplicated transactional systems," in *Proc. of SRDS '11*, Oct 2011.

[22] P. J. Marandi, M. Primi, and F. Pedone, "High performance state-machine replication," in *Proc. of DSN '11*, Jun. 2011.

[23] P. J. Marandi, C. E. Bezerra, and F. Pedone, "Rethinking state-machine replication for parallelism," in *Proc. ICDCS '14*, Jun. 2014.

[24] B. Arun, S. Hirve, R. Palmieri, S. Peluso, and B. Ravindran, "Speculative client execution in deferred update replication," in *Proc. of MW4NG '14: the 9th Middleware for Next Generation Internet Computing Workshop*, Dec. 2014.

[25] D. Sciascia, F. Pedone, and F. Junqueira, "Scalable deferred update replication," in *Proc. of DSN '12*, Jun. 2012.

[26] L. Pacheco, D. Sciascia, and F. Pedone, "Parallel deferred update replication," in *Proc. of NCA '14*, Aug. 2014.

[27] D. Sciascia and F. Pedone, "Geo-replicated storage with scalable deferred update replication," in *Proc. of DSN '13*, Jun. 2013.

[28] B. Ciciani, D. M. Dias, and P. S. Yu, "Analysis of replication in distributed database systems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 2, no. 2, pp. 247–261, Jun. 1990.

[29] R. Jiménez-Peris, M. Patiño-Martínez, B. Kemme, and G. Alonso, "Improving the scalability of fault-tolerant database clusters," in *Proc. of ICDCS '02*, Jul. 2002.

[30] R. Jiménez-Peris, M. Patiño-Martínez, G. Alonso, and B. Kemme, "Are quorums an alternative for data replication?" *ACM Transactions on Database Systems*, vol. 28, no. 3, pp. 257–294, Sep. 2003.

[31] M. Nicola and M. Jarke, "Performance modeling of distributed and replicated databases," *IEEE Transactions on Knowledge and Data Engineering*, vol. 12, no. 4, pp. 645–672, Jul. 2000.

[32] D. Didona, P. Romano, S. Peluso, and F. Quaglia, "Transactional Auto Scaler: Elastic scaling of replicated in-memory transactional data grids," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 9, no. 2, pp. 11:1–11:32, Jul. 2014.

[33] F. Borran, M. Hutle, N. Santos, and A. Schiper, "Quantitative analysis of consensus algorithms," *IEEE Transactions on Dependable and Secure Computing*, vol. 9, no. 2, pp. 236–249, March-Apr. 2012.

[34] M. Couceiro, P. Romano, and L. Rodrigues, "A machine learning approach to performance prediction of total order broadcast protocols," in *Proc. of SASO '10*, Sept.-Oct. 2010.

[35] N. Santos and A. Schiper, "Optimizing Paxos with batching and pipelining," *Theoretical Computer Science (TCS)*, vol. 496, pp. 170–183, 2013.

[36] C. Kotselidis, M. Ansari, K. Jarvis, M. Luján, C. C. Kirkham, and I. Watson, "DiSTM: A software transactional memory framework for clusters," in *Proc. of ICPP '08*, Sep. 2008.

[37] C. Kotselidis, M. Lujan, M. Ansari, K. Malakasis, B. Kahn, C. Kirkham, and I. Watson, "Clustering JVMs with software transactional memory support," in *Proc. of IPDPS '10*, Apr. 2010.

[38] M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues, "D2STM: Dependable distributed software transactional memory," in *Proc. of PRDC '09*, Nov. 2009.

[39] N. Carvalho, P. Romano, and L. Rodrigues, "Asynchronous lease-based replication of software transactional memory," in *Proc. of ACM/IFIP/USENIX Middleware '10*, ser. LNCS, vol. 6452, 2010.

[40] M. K. Aguilera, C. Wei, and S. Toueg, "Failure detection and consensus in the crash-recovery model," in *Proc. of DISC '98*, Sep. 1998.

[41] A. Birrell, "An introduction to programming with C# threads," Microsoft Research, Tech. Rep., May 2005.

[42] T. Kobus, M. Kokociński, and P. T. Wojciechowski, "Introduction to transactional replication," in *Transactional Memory: Foundations, Algorithms, Tools, and Applications. COST Action Euro-TM IC1001*, ser. LNCS. Springer, 2015, vol. 8913, pp. 309–340.

[43] A. Y. Grama, A. Gupta, and V. Kumar, "Isoefficiency: Measuring the scalability of parallel algorithms and architectures," *Journal of IEEE Parallel and Distributed Technology: Systems & Applications*, vol. 1, no. 3, pp. 12–21, Aug. 1993.

[44] V. P. Kumar and A. Gupta, "Analyzing scalability of parallel algorithms and architectures," *Journal of Parallel and Distributed Computing*, vol. 22, no. 3, pp. 379–391, Sep. 1994.

[45] T. Kobus, M. Kokociński, and P. T. Wojciechowski, "Hybrid replication: State-machine-based and deferred-update replication schemes combined," in *Proc. of ICDCS '13*, Jul. 2013.

Paweł T. Wojciechowski received his PhD degree in Computer Science from the University of Cambridge in 2000. He holds a Habilitation degree from Poznań University of Technology, Poland, where he is currently an Assistant Professor in the Institute of Computing Science. He was a postdoctoral researcher in the School of Computer and Communication Sciences at École Polytechnique Fédérale de Lausanne (EPFL), Switzerland, from 2001 to 2005. He has led many research projects and coauthored dozens of papers. His research interests span topics in concurrency, distributed computing, and programming languages.

Tadeusz Kobus is currently pursuing a PhD degree and working as a Research Assistant in the Institute of Computing Science, Poznań University of Technology, Poland, where he also received BS and MS degrees in Computer Science, in 2009 and 2010 respectively. In the summer of 2014, he was an intern at IBM T. J. Watson Research Center. His research interests include fault tolerant distributed algorithms, transactional memory, and group communication systems.

Maciej Kokociński is currently pursuing a PhD degree and working as a Research Assistant in the Institute of Computing Science, Poznań University of Technology, Poland, where he also received BS and MS degrees in Computer Science, in 2009 and 2010 respectively. He was a summer intern at Microsoft in Redmond. His research interests include theory of distributed systems and transactional memory.

State-Machine and Deferred-Update Replication: Analysis and Comparison (Supplemental Material)

Paweł T. Wojciechowski, Tadeusz Kobus, and Maciej Kokociński

1 Introduction

This is an appendix to the article: "State-machine and deferred-update replication: analysis and comparison" by P. T. Wojciechowski, T. Kobus, and M. Kokociński. It contains a list of symbols in Table 1, the proofs of lemmas and theorems in Section 2, some additional results for LSMR in Section 3, description of state machine replication with multiversioning in Section 4, and a thorough description of the experimental evaluation results in Section 5.

2 Proofs

2.1 State machine replication: SMR and LSMR

Below we compute the lower bounds on the total time of processing n requests, assuming, for simplicity, that $\frac{n}{\beta} = \lceil \frac{n}{\beta} \rceil$ in SMR, and $\frac{n_{rw}}{\beta} = \lceil \frac{n_{rw}}{\beta} \rceil$ in LSMR.

2.1.1 Lower bound for SMR

By definition

$$T^{\text{SMR}}_{lowb} = max(\frac{n}{\beta}t_r, ne) + \delta^{\text{SMR}} = max(\frac{t_r}{\beta}, e)n + \delta^{\text{SMR}} \quad (1)$$

$$\text{where} \quad \delta^{\text{SMR}} = \begin{cases} \beta e & \text{if } max(\frac{t_r}{\beta}, e) = \frac{t_r}{\beta} \\ t_r & \text{if } max(\frac{t_r}{\beta}, e) = e \ . \end{cases} \quad (2)$$

If $\frac{t_r}{\beta} \geq e$, so $\delta^{\text{SMR}} = \beta e$, we say that the *abcast time is dominant* (e.g., the network is slow or the requests are short). If $e \geq \frac{t_r}{\beta}$, so $\delta^{\text{SMR}} = t_r$, we say that the *execution time is dominant* (e.g., the network is fast or the requests are long).

• *The authors are with the Institute of Computing Science, Poznań University of Technology, 60-965 Poznań, Poland.*
E-mail: {Pawel.T.Wojciechowski, Tadeusz.Kobus, Maciej.Kokocinski}@cs.put.edu.pl

Manuscript received August 3, 2015; revised April 10, 2016; accepted July 1, 2016. Date of publication ?? August 2016; date of current version ?? August 2016. Recommended for acceptance by ??.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. ????.

TABLE 1
A list of symbols

P_i	ith process (server)
P	replicated process (server)
N	number of processes (servers)
S_i	(local) state of P_i
S	replicated state
o	object
n_q	number of read-only (RO) requests (queries)
n_{rw}	number of read-write (RW) update requests
n	total number of requests ($n = n_q + n_{rw}$)
r, x, y	requests
T_x	transaction spawned by request x
$\prec$	transaction precedence relation
$\rightarrow$	happens-before relation
t_{rc}	time of replica coordination (in SMR)
t_{ac}	time of agreement coordination (in DUR)
e	time of executing the transaction code only
t_o	DUR overhead time per RW transaction, excl. abcast time
t'_o	DUR overhead time per RO transaction, excl. abcast time
e_o	$e_o = e + t_o$ (in DUR)
e'_o	$e'_o = e + t'_o$ (in DUR)
t_r	time of request abcast (in SMR)
t_u	time of update abcast (in DUR)
T^{rs}	total time of processing requests
T^{rs}_{lowb}	lower bound on time T^{rs}
c	number of CPU cores
β_1	number of messages broadcast per abcast instance
β_1	number of concurrent abcast instances at a time
β	$\beta = \beta_1\beta_2$
T^{SEQ}	sequential execution time
T_{Nc}	parallel execution time (on N c-core CPUs)
S^{rs}	speedup of system rs
E^{rs}	efficiency of system rs
K_{rw}	number of RW conflicts (in DUR)
K_q	number of RO conflicts (in DUR)
K	total number of conflicts (in DUR) ($K = K_q + K_{rw}$)
k_{rw}	number of conflicts per RW transaction ($k_{rw} = \frac{K_{rw}}{n_{rw}}$)

2.1.2 Lower bound for LSMR

a) If each server has only one CPU core ($c = 1$), the lower bound on the time of processing n requests is:

$$T^{\text{LSMR}}_{lowb} = t^{\text{SMR}}_{rw} + \Delta, \ \text{ where} \quad (3)$$

$$t^{\text{SMR}}_{rw} = max(\frac{n_{rw}}{\beta}t_r, n_{rw}e) + \delta^{\text{SMR}} = max(\frac{t_r}{\beta}, e)n_{rw} + \delta^{\text{SMR}} \ . \quad (4)$$

We estimate Δ as follows. First, we assume that RO requests are executed by any free server that currently does

not execute any RW request. Those RO requests which are processed in parallel with message broadcasting take $t_A = t_{rw}^{\text{SMR}} - n_{rw}e$ time units and can be neglected since they do not increase the total time of processing RW requests. Then, Δ is the time of processing by N single-core servers any remaining RO requests after all RW requests have been executed, or $\Delta = 0$ if none such a request exists:

$$\Delta = \begin{cases} \Delta' & \text{if } \Delta' > 0 \\ 0 & \text{if } \Delta' \leq 0 \end{cases} \tag{5}$$

$$\text{where } \Delta' = \left\lceil \frac{n_q}{N} \right\rceil e - t_A = \left\lceil \frac{n_q}{N} \right\rceil e - (t_{rw}^{\text{SMR}} - n_{rw}e)$$
$$= \begin{cases} (\lceil \frac{n_q}{N} \rceil + n_{rw} - \beta)e - \frac{n_{rw}}{\beta} t_r & \text{if } \frac{t_r}{\beta} \geq e \\ \lceil \frac{n_q}{N} \rceil e - t_r & \text{if } e \geq \frac{t_r}{\beta} \end{cases} .$$

Proof: If $\frac{t_r}{\beta} \geq e$ then, by (2) and (4), $\delta^{\text{SMR}} = \beta e$ and $t_{rw}^{\text{SMR}} = \frac{t_r}{\beta} n_{rw} + \beta e$. Then

$$\Delta' = \left\lceil \frac{n_q}{N} \right\rceil e - t_A = \left\lceil \frac{n_q}{N} \right\rceil e - (t_{rw}^{\text{SMR}} - n_{rw}e)$$
$$= \left\lceil \frac{n_q}{N} \right\rceil e + n_{rw}e - \left(\frac{t_r}{\beta} n_{rw} + \beta e \right)$$
$$= \left(\left\lceil \frac{n_q}{N} \right\rceil + n_{rw} - \beta \right) e - \frac{n_{rw}}{\beta} t_r .$$

If $e \geq \frac{t_r}{\beta}$ then, by (2) and (4), $\delta^{\text{SMR}} = t_r$ and $t_{rw}^{\text{SMR}} = en_{rw} + t_r$. Then

$$\Delta' = \left\lceil \frac{n_q}{N} \right\rceil e - t_A = \left\lceil \frac{n_q}{N} \right\rceil e - (t_{rw}^{\text{SMR}} - n_{rw}e)$$
$$= \left\lceil \frac{n_q}{N} \right\rceil e + n_{rw}e - (en_{rw} + t_r) = \left\lceil \frac{n_q}{N} \right\rceil e - t_r .$$

□

b) If each server has c CPU cores ($c \geq 1$), the lower bound on the time of processing n requests is:

$$T_{lowb}^{\text{LSMR}} = t_{rw}^{\text{SMR}} + \Pi \tag{6}$$

where Π is estimated as follows. Let us assume that RO requests that before were executed on single core servers for Δ time units are now executed on $c-1$ extra cores that each server has at its disposal. This takes $t_B = \frac{\Delta}{c-1}$ time units. A certain number of RO requests are executed in parallel with abcast for t_A time units. Then, Π is the time of processing any other RO requests by all available c cores of each server, or $\Pi = 0$ if none exists:

$$\Pi = \begin{cases} \lceil \Pi' \rceil & \text{if } \Pi' > 0 \\ 0 & \text{if } \Pi' \leq 0 \end{cases} \tag{7}$$

$$\text{where } \Pi' = \frac{(t_B - t_A)(c-1)}{c} = \frac{\Delta - (t_{rw}^{\text{SMR}} - n_{rw}e)(c-1)}{c}$$
$$= \begin{cases} \frac{\Delta - (\frac{t_r}{\beta} n_{rw} + \beta e - n_{rw}e)(c-1)}{c} & \text{if } \frac{t_r}{\beta} \geq e \\ \frac{\Delta - t_r(c-1)}{c} & \text{if } e \geq \frac{t_r}{\beta} \end{cases} .$$

Proof: $t_A = t_{rw}^{\text{SMR}} - n_{rw}e$ and $t_B = \frac{\Delta}{c-1}$ by definition. Hence

$$\Pi' = \frac{(t_B - t_A)(c-1)}{c}$$
$$= \frac{[\frac{\Delta}{c-1} - (t_{rw}^{\text{SMR}} - n_{rw}e)](c-1)}{c}$$
$$= \frac{\Delta - (t_{rw}^{\text{SMR}} - n_{rw}e)(c-1)}{c} .$$

Now let us consider two cases. If $\frac{t_r}{\beta} \geq e$ then, by (2) and (4), $\delta^{\text{SMR}} = \beta e$ and $t_{rw}^{\text{SMR}} = \frac{t_r}{\beta} n_{rw} + \beta e$. Then

$$\Pi' = \frac{\Delta - (t_{rw}^{\text{SMR}} - n_{rw}e)(c-1)}{c}$$
$$= \frac{\Delta - (\frac{t_r}{\beta} n_{rw} + \beta e - n_{rw}e)(c-1)}{c} .$$

If $e \geq \frac{t_r}{\beta}$ then, by (2) and (4), $\delta^{\text{SMR}} = t_r$ and $t_{rw}^{\text{SMR}} = en_{rw} + t_r$. Then

$$\Pi' = \frac{\Delta - (t_{rw}^{\text{SMR}} - n_{rw}e)(c-1)}{c}$$
$$= \frac{\Delta - (en_{rw} + t_r - n_{rw}e)(c-1)}{c}$$
$$= \frac{\Delta - t_r(c-1)}{c} .$$

□

Below we show another result for Δ. Let us consider the case when $\Pi' > 0$.

$$\Pi' > 0 \implies \frac{\Delta - (t_{rw}^{\text{SMR}} - n_{rw}e)(c-1)}{c} > 0$$
$$\implies \Delta > (t_{rw}^{\text{SMR}} - n_{rw}e)(c-1)$$
$$\implies \frac{\Delta}{c-1} > t_{rw}^{\text{SMR}} - n_{rw}e .$$

Then, by definition of Π if $\frac{\Delta}{c-1} > t_{rw}^{\text{SMR}} - n_{rw}e$ then $\Pi = \lceil \Pi' \rceil$ and if $\frac{\Delta}{c-1} \leq t_{rw}^{\text{SMR}} - n_{rw}e$ then $\Pi = 0$.

Note that extra cores do not make any difference for RW requests, as they are processed sequentially by each server. Note also that if $c = 1$, then as expected $\Pi = \Delta$.

Let us compare the performance of SMR and LSMR, assuming that both systems process requests optimally:

Lemma 1. *The difference in elapsed time of processing n requests without any delay by LSMR compared to SMR is:*

$$T_{diff}{}_{LSMR}^{SMR} = T_{lowb}^{SMR} - T_{lowb}^{LSMR}$$
$$= \begin{cases} \frac{n_q}{\beta} t_r - \Pi & \text{if } \frac{t_r}{\beta} \geq e \\ n_q e - \Pi & \text{if } e \geq \frac{t_r}{\beta} \end{cases} . \tag{8}$$

Proof: By (6) we obtain

$$T_{diff}{}_{\text{LSMR}}^{\text{SMR}} = T_{lowb}^{\text{SMR}} - T_{lowb}^{\text{LSMR}}$$
$$= T_{lowb}^{\text{SMR}} - (t_{rw}^{\text{SMR}} + \Pi) .$$

If $\frac{t_r}{\beta} \geq e$ then, by (2) and (4), $\delta^{\text{SMR}} = \beta e$ and $t_{rw}^{\text{SMR}} = \frac{t_r}{\beta} n_{rw} + \beta e$. Then, by (1), we get

$$T_{diff}{}_{\text{LSMR}}^{\text{SMR}} = \frac{t_r}{\beta} n + \beta e - (\frac{t_r}{\beta} n_{rw} + \beta e + \Pi)$$
$$= \frac{t_r}{\beta}(n - n_{rw}) - \Pi = \frac{t_r}{\beta} n_q - \Pi = \frac{n_q}{\beta} t_r - \Pi .$$

If $e \geq \frac{t_r}{\beta}$ then, by (2) and (4), $\delta^{\text{SMR}} = t_r$ and $t_{rw}^{\text{SMR}} = en_{rw} + t_r$. Then, by (1), we get

$$T_{diff}{}_{\text{LSMR}}^{\text{SMR}} = en + t_r - (en_{rw} + t_r + \Pi)$$
$$= e(n - n_{rw}) - \Pi = n_q e - \Pi .$$

□

Theorem 1. *In the best case, a service replicated using LSMR and executed on N multi-core servers ($c \geq 1$) is faster than processing requests sequentially if*

$$(n-\beta)e > \frac{n_{rw}}{\beta}t_r + \Pi \tag{9}$$

when the abcast time is dominant, and if

$$n_q e > t_r + \Pi \tag{10}$$

when the execution time is dominant.

Proof: If $\frac{t_r}{\beta} \geq e$ then, by (2) and (4), $\delta^{\text{SMR}} = \beta e$ and $t_{rw}^{\text{SMR}} = \frac{t_r}{\beta}n_{rw} + \beta e$. Then, by (6), we get

$$\begin{aligned} & T^{\text{SEQ}} > T_{lowb}^{\text{LSMR}} \\ \Longrightarrow\ & ne > t_{rw}^{\text{SMR}} + \Pi \\ \Longrightarrow\ & ne > \frac{t_r}{\beta}n_{rw} + \beta e + \Pi \\ \Longrightarrow\ & (n-\beta)e > \frac{n_{rw}}{\beta}t_r + \Pi\ . \end{aligned}$$

If $e \geq \frac{t_r}{\beta}$ then, by (2) and (4), $\delta^{\text{SMR}} = t_r$ and $t_{rw}^{\text{SMR}} = en_{rw} + t_r$. Then, by (6), we get

$$\begin{aligned} & T^{\text{SEQ}} > T_{lowb}^{\text{LSMR}} \\ \Longrightarrow\ & ne > t_{rw}^{\text{SMR}} + \Pi \\ \Longrightarrow\ & ne > en_{rw} + t_r + \Pi \\ \Longrightarrow\ & n_q e > t_r + \Pi\ . \end{aligned}$$

□

2.1.3 SMR and LSMR scalability

Theorem 2. *LSMR scales and SMR does not scale.*

Proof: The speedup and efficiency of a system replicated using LSMR in the best possible case are given by:

$$\begin{aligned} S^{\text{LSMR}} &= T^{\text{SEQ}}/T_{lowb}^{\text{LSMR}} \\ E^{\text{LSMR}} &= \frac{T^{\text{SEQ}}}{T_{lowb}^{\text{LSMR}}Nc} = \frac{ne}{(t_{rw}^{\text{SMR}} + \Pi)Nc}\ . \end{aligned} \tag{11}$$

Consider $\Pi > 0$. Then we have

$$\begin{aligned} E_{\Pi>0}^{\text{LSMR}} &= \frac{T^{\text{SEQ}}}{T_{lowb}^{\text{LSMR}}Nc} = \frac{ne}{(t_{rw}^{\text{SMR}} + \Pi)Nc} \\ &= \frac{ne}{\left(t_{rw}^{\text{SMR}} + \left\lceil \frac{\Delta - (t_{rw}^{\text{SMR}} - n_{rw}e)(c-1)}{c} \right\rceil\right)Nc} \\ &\approx \frac{ne}{(t_{rw}^{\text{SMR}}c + \Delta - (t_{rw}^{\text{SMR}} - n_{rw}e)(c-1))N} \\ &\approx \frac{ne}{(t_{rw}^{\text{SMR}}c + \Delta - (t_{rw}^{\text{SMR}}c - t_{rw}^{\text{SMR}} - n_{rw}e(c-1)))N} \\ &\approx \frac{ne}{(t_{rw}^{\text{SMR}}c + \Delta - t_{rw}^{\text{SMR}}c + t_{rw}^{\text{SMR}} + n_{rw}e(c-1))N} \\ &\approx \frac{ne}{(\Delta + t_{rw}^{\text{SMR}} + n_{rw}ec - n_{rw}e)N} \\ &\approx \frac{ne}{(\lceil \frac{n_q}{N} \rceil e - t_{rw}^{\text{SMR}} + n_{rw}e + t_{rw}^{\text{SMR}} + n_{rw}ec - n_{rw}e)N} \\ &\approx \frac{ne}{(\lceil \frac{n_q}{N} \rceil e + n_{rw}ec)N} \\ &\approx \frac{n}{(\lceil \frac{n_q}{N} \rceil + n_{rw}c)N} \approx \frac{n}{n_q + n_{rw}Nc}\ . \end{aligned} \tag{12}$$

For a given problem instance, the efficiency of the LSMR system drops with an increasing number of processors. In order to ensure that the LSMR efficiency does not decrease as the number of processors increase, the number of RO requests should increase. Thus, if the number of RO requests is large enough, LSMR scales.

If $\Pi = 0$, then

$$\begin{aligned} E_{\Pi=0}^{\text{LSMR}} &= \frac{T^{\text{SEQ}}}{T_{lowb}^{\text{LSMR}}Nc} = \frac{ne}{(t_{rw}^{\text{SMR}} + \Pi)Nc} = \frac{ne}{t_{rw}^{\text{SMR}}Nc} \\ &= \frac{ne}{(max(\frac{t_r}{\beta}, e)n_{rw} + \delta^{\text{SMR}})Nc}\ . \end{aligned} \tag{13}$$

If $\frac{t_r}{\beta} \geq e$ then, by (2), $\delta^{\text{SMR}} = \beta e$, so

$$E_{\Pi=0}^{\text{LSMR}} = \frac{ne}{(\frac{t_r}{\beta}n_{rw} + \beta e)Nc} = \frac{ne}{(\beta e + \frac{n_{rw}}{\beta}t_r)Nc}\ .$$

If $e \geq \frac{t_r}{\beta}$ then, by (2), $\delta^{\text{SMR}} = t_r$, so

$$E_{\Pi=0}^{\text{LSMR}} = \frac{ne}{(en_{rw} + t_r)Nc} = \frac{ne}{(t_r + n_{rw}e)Nc}\ .$$

Hence, we obtain

$$E_{\Pi=0}^{\text{LSMR}} = \begin{cases} \frac{ne}{(\frac{n_{rw}}{\beta}t_r + \beta e)Nc} & \text{if } \frac{t_r}{\beta} \geq e \\ \frac{ne}{(t_r + n_{rw}e)Nc} & \text{if } \frac{t_r}{\beta} \leq e\ . \end{cases} \tag{14}$$

In both cases, the LSMR system also scales if the number of RO requests (in the numerator) will be sufficient to compensate for an increasing number of cores (in the denominator), so that efficiency is constant.

If the SMR system is unoptimized, there is no need to distinguish between RO and RW requests (i.e., $n_q = 0$, $n = n_{rw}$, $\Pi = 0$) and we get

$$E^{\text{SMR}} = \frac{T^{\text{SEQ}}}{T_{lowb}^{\text{SMR}}Nc} = \frac{ne}{T_{lowb}^{\text{SMR}}Nc} = E_{\Pi=0}^{\text{LSMR}} \text{ where } n_{rw} = n\ . \tag{15}$$

It is easy to see that the efficiency E^{SMR} cannot be maintained at a constant value when increasing the number of processors/cores since if we increase the problem size n in the numerator, then the denominator increases even more. Thus, we have proven that SMR does not scale. □

2.2 Deferred update replication: DUR and MvDUR

2.2.1 Lower bound for MvDUR

By definition

$$\begin{aligned} T_{lowb}^{\text{MvDUR}} &= max(t_a^{\text{MvDUR}}, t_e^{\text{MvDUR}}), \quad \text{where} \\ t_a^{\text{MvDUR}} &= e_o + \left\lceil \frac{n_{rw} + K_{rw}}{\beta} \right\rceil t_u \\ t_e^{\text{MvDUR}} &\approx \left\lceil \frac{n_{rw} + K_{rw}}{Nc} \right\rceil e_o + \tau_2 \\ \tau_2 &= max(t_u, \left\lceil \frac{n_q}{Nc} \right\rceil e)\ . \end{aligned} \tag{16}$$

If $t_a^{\text{DUR}} \geq t_e^{\text{DUR}}$ then we say that the *abcast time is dominant*. If $t_a^{\text{DUR}} < t_e^{\text{DUR}}$ then the *execution time is dominant*. If $t_u \geq \lceil \frac{n_q}{Nc} \rceil e$ (so $\tau_2 = t_u$) then we say that the *abcast dominates RO requests*. If $t_u < \lceil \frac{n_q}{Nc} \rceil e$ (so $\tau_2 = \lceil \frac{n_q}{Nc} \rceil e$) then the *RO requests dominate abcast*.

2.2.2 Analysis of transaction conflicts

We consider two conflict patterns, assuming at most c concurrent transactions on each server.

a) Transactions never wait for conflicts to be resolved:

In this case, only the first transaction commits and the remaining $Nc-1$ transactions abort and repeat execution in parallel with a new fresh transaction that is processed by a free core. This process repeats until the last new fresh transaction appears. Then, again one transaction commits and all but one are reexecuted, and so on until the last transaction is executed.

Thus, the time of processing n given requests assuming that the execution time is dominant:

$$\begin{aligned} t_e^{\text{DUR}}(a) &= e_o + (n-Nc)e_o + (Nc-1)e_o + t_u \\ &= e_o + ne_o - Nce_o + Nce_o - e_o + t_u \\ &= ne_o + t_u \ . \end{aligned} \tag{17}$$

The number of conflicts is

$$\begin{aligned} K_a &= (n-Nc)(Nc-1) + \frac{(Nc-1)Nc}{2} \\ &= nNc - n - (Nc)^2 + Nc + \frac{(Nc-1)Nc}{2} \\ &= \frac{2nNc - 2n - 2(Nc)^2 + 2Nc + (Nc)^2 - Nc}{2} \\ &= \frac{2nNc - 2n - (Nc)^2 + Nc}{2} \\ &= n(Nc-1) + \frac{Nc-(Nc)^2}{2} \ . \end{aligned} \tag{18}$$

b) Transactions wait until conflicts are resolved:

In this case, we execute the first Nc transactions in parallel. Since they all conflict, we wait with processing the next batch of Nc transactions until all conflicts are resolved and the conflicting transactions are reexecuted. We repeat this process until all transactions are executed. Since in the last iteration the number of fresh new transactions can be smaller than Nc, we use μ to describe their time of execution.

Thus, the time of processing n given requests assuming that the execution time is dominant:

$$\begin{aligned} t_e^{\text{DUR}}(b) &= \left(\left\lfloor\frac{n}{Nc}\right\rfloor + (Nc-1)\left\lfloor\frac{n}{Nc}\right\rfloor + \mu\right)e_o + t_u, \\ &\qquad \text{where } \mu = n - \left\lfloor\frac{n}{Nc}\right\rfloor Nc \\ &= \left(\left\lfloor\frac{n}{Nc}\right\rfloor + Nc\left\lfloor\frac{n}{Nc}\right\rfloor - \left\lfloor\frac{n}{Nc}\right\rfloor + \mu\right)e_o + t_u \\ &= \left(Nc\left\lfloor\frac{n}{Nc}\right\rfloor + n - \left\lfloor\frac{n}{Nc}\right\rfloor Nc\right)e_o + t_u \\ &= ne_o + t_u \ . \end{aligned} \tag{19}$$

The number of conflicts is

$$\begin{aligned} K_b &= \frac{(Nc-1)Nc}{2}\left\lfloor\frac{n}{Nc}\right\rfloor + \frac{(\mu-1)\mu}{2} \\ &= \frac{(Nc)^2 - Nc}{2}\left\lfloor\frac{n}{Nc}\right\rfloor + \frac{\mu^2-\mu}{2} \\ &= \frac{(Nc)^2}{2}\left\lfloor\frac{n}{Nc}\right\rfloor - \frac{Nc}{2}\left\lfloor\frac{n}{Nc}\right\rfloor + \frac{\mu^2}{2} - \frac{n}{2} + \frac{Nc}{2}\left\lfloor\frac{n}{Nc}\right\rfloor \\ &= \frac{(Nc)^2}{2}\left\lfloor\frac{n}{Nc}\right\rfloor + \frac{\mu^2-n}{2} = \frac{1}{2}\Big((Nc)^2\left\lfloor\frac{n}{Nc}\right\rfloor + \\ &\quad + n^2 - 2nNc\left\lfloor\frac{n}{Nc}\right\rfloor + \left(Nc\left\lfloor\frac{n}{Nc}\right\rfloor\right)^2 - n\Big) \\ &\approx \frac{1}{2}\Big(nNc + n^2 - 2n^2 + n^2 - n\Big) \\ &\approx \frac{1}{2}\Big(nNc - n\Big) \approx \frac{n(Nc-1)}{2} \ . \end{aligned} \tag{20}$$

2.2.3 MvDUR scalability

Below we consider scalability of a replicated system in the best possible case, separately for the execution and abcast time dominance. Then, we get

Theorem 3. *When the execution time is dominant, MvDUR scales if the number of conflicts does not explode.*

Proof: The best possible parallel algorithm matching the specification of MvDUR takes at least $T_{Nc} = T_{lowb}^{\text{MvDUR}}$ time units. Thus, by (16), if the execution time is dominant, the speedup and efficiency of an MvDUR system in the best possible case are, respectively:

$$\begin{aligned} S^{\text{MvDUR}} &= T^{\text{SEQ}}/t_e^{\text{MvDUR}} \\ E^{\text{MvDUR}} &= \frac{T^{\text{SEQ}}}{t_e^{\text{MvDUR}}Nc} \\ &= \begin{cases} \frac{ne}{(\lceil\frac{n_{rw}+K_{rw}}{Nc}\rceil e_o + t_u)Nc} & \text{if } t_u \geq \lceil\frac{n_q}{Nc}\rceil e \\ \frac{ne}{(\lceil\frac{n_{rw}+K_{rw}}{Nc}\rceil e_o + \lceil\frac{n_q}{Nc}\rceil e)Nc} & \text{if } t_u \leq \lceil\frac{n_q}{Nc}\rceil e \end{cases} \\ &\approx \begin{cases} \frac{ne}{(n_{rw}+K_{rw})e_o + t_u Nc} & \text{if } t_u \geq \lceil\frac{n_q}{Nc}\rceil e \\ \frac{ne}{(n_{rw}+K_{rw})e_o + n_q e} & \text{if } t_u \leq \lceil\frac{n_q}{Nc}\rceil e \end{cases} \\ &\approx \begin{cases} \frac{ne}{\sigma + t_u Nc} & \text{if } t_u \geq \lceil\frac{n_q}{Nc}\rceil e \\ \frac{ne}{\sigma + n_q e} & \text{if } t_u \leq \lceil\frac{n_q}{Nc}\rceil e \end{cases} \\ &\qquad \text{where } \ \sigma = (n_{rw}+K_{rw})e_o \ . \end{aligned} \tag{21}$$

In case when $e \approx e_o$, efficiency can be approximated to

$$\begin{aligned} E^{\text{MvDUR}} &\approx \begin{cases} \frac{n}{n_{rw}+K_{rw}+\frac{t_u Nc}{e}} & \text{if } t_u \geq \lceil\frac{n_q}{Nc}\rceil e \\ \frac{n}{n_{rw}+K_{rw}+n_q} & \text{if } t_u \leq \lceil\frac{n_q}{Nc}\rceil e \end{cases} \\ &= \begin{cases} \frac{n}{n_{rw}+K_{rw}+\frac{t_u Nc}{e}} & \text{if } t_u \geq \lceil\frac{n_q}{Nc}\rceil e \\ \frac{n}{n+K_{rw}} & \text{if } t_u \leq \lceil\frac{n_q}{Nc}\rceil e \ . \end{cases} \end{aligned} \tag{22}$$

Thus, if abcast dominates RO requests (i.e., $t_u \geq \lceil\frac{n_q}{Nc}\rceil e$), then the MvDUR-based system scales if the number of requests n is large enough to compensate for an increasing number of cores and the number of conflicts does not explode. If RO transactions predominate ($t_u < \lceil\frac{n_q}{Nc}\rceil e$), then the system scales perfectly if the number of conflicts does not explode. □

Theorem 4. *When the abcast time is dominant, MvDUR scales worse than when the execution time is dominant, and it does not scale if all requests are updating.*

Proof: See the article. □

3 Additional Results for (L)SMR

From Lemma 1, we can infer the time difference between SMR and LSMR for the simplest system, where $c = 1$ (so $\Pi = \Delta$) and $N = 1$:

Lemma 3. *If $N = c = 1$ then*

$$T_{diff}{}^{SMR}_{LSMR} = \begin{cases} (\frac{t_r}{\beta} - e)n + \beta e & \text{if } \Delta' > 0 \\ \frac{n_q}{\beta} t_r & \text{if } \Delta' \leq 0 \end{cases} \quad (23)$$

when the abcast time (network communication) is dominant, and

$$T_{diff}{}^{SMR}_{LSMR} = \begin{cases} t_r & \text{if } \Delta' > 0 \\ n_q e & \text{if } \Delta' \leq 0 \end{cases} \quad (24)$$

when the execution time (CPU processing) is dominant.

Proof:
If $\frac{t_r}{\beta} \geq e$ then by (5):

If $\Delta' > 0$

$$\begin{aligned} T_{diff}{}^{\text{SMR}}_{\text{LSMR}} &= \frac{n_q}{\beta} t_r - \Pi = \frac{n_q}{\beta} t_r - \Delta \\ &= \frac{n_q}{\beta} t_r - (n_q + n_{rw} - \beta)e + \frac{n_{rw}}{\beta} t_r \\ &= \frac{n}{\beta} t_r - (n - \beta)e = \frac{n}{\beta} t_r - ne + \beta e \\ &= \left(\frac{t_r}{\beta} - e\right) n + \beta e \, . \end{aligned}$$

If $\Delta' \leq 0$, then $\Pi = \Delta = 0$, so

$$T_{diff}{}^{\text{SMR}}_{\text{LSMR}} = \frac{n_q}{\beta} t_r \, .$$

If $e \geq \frac{t_r}{\beta}$ then by (5):

If $\Delta' > 0$

$$T_{diff}{}^{\text{SMR}}_{\text{LSMR}} = n_q e - \Pi = n_q e - n_q e + t_r = t_r \, .$$

If $\Delta' \leq 0$, then $\Pi = \Delta = 0$, so

$$T_{diff}{}^{\text{SMR}}_{\text{LSMR}} = n_q e \, .$$

□

From Theorem 1, it is easy to show as below:

Lemma 4. *In a system with single core CPUs ($c = 1$), in the best case, a service replicated using LSMR and executed on N servers is not slower than processing requests sequentially if $\Delta > 0$. If $\Delta = 0$ then LSMR is not slower if $(n - \beta)e \geq \frac{n_{rw}}{\beta} t_r$ when the abcast time is dominant, or if $n_q e \geq t_r$ when the execution time is dominant.*

Proof: Consider $\Delta > 0$. When the abcast time is dominant (i.e., $\frac{t_r}{\beta} \geq e$) then $\delta^{\text{SMR}} = \beta e$ and by (5):

$$\Delta = \Delta' = \left(\left\lceil \frac{n_q}{N} \right\rceil + n_{rw} - \beta \right) e - \frac{n_{rw}}{\beta} t_r$$

and from Theorem 1 we get that LSMR is not slower than its sequential counterpart if

$$(n - \beta)e \geq \frac{n_{rw}}{\beta} t_r + \Pi \, .$$

Since we assumed that $c = 1$, there is $\Pi = \Delta$ and we get

$$\begin{aligned} & (n - \beta)e \geq \frac{n_{rw}}{\beta} t_r + \left(\left\lceil \frac{n_q}{N} \right\rceil + n_{rw} - \beta \right) e - \frac{n_{rw}}{\beta} t_r \\ \implies & ne - \beta e \geq \left\lceil \frac{n_q}{N} \right\rceil e + n_{rw} e - \beta e \\ \implies & ne \geq \left\lceil \frac{n_q}{N} \right\rceil e + n_{rw} e \\ \implies & (n - n_{rw})e \geq \left\lceil \frac{n_q}{N} \right\rceil e \\ \implies & n_q e \geq \left\lceil \frac{n_q}{N} \right\rceil e \\ \implies & n_q \geq \left\lceil \frac{n_q}{N} \right\rceil \, . \end{aligned}$$

But $n_q \geq \lceil \frac{n_q}{N} \rceil$ is always true, which means that LSMR is always not slower than its sequential counterpart when the abcast time is dominant.

When the execution time is dominant (i.e., $e \geq \frac{t_r}{\beta}$) then $\delta^{\text{SMR}} = t_r$ and by (5):

$$\Delta = \Delta' = \left\lceil \frac{n_q}{N} \right\rceil e - t_r$$

and from Theorem 1 we get that LSMR is not slower than its sequential counterpart if

$$n_q e \geq t_r + \Pi \, .$$

Since we assumed that $c = 1$, there is $\Pi = \Delta$ and we get

$$\begin{aligned} & n_q e \geq t_r + \left\lceil \frac{n_q}{N} \right\rceil e - t_r \\ \implies & n_q e \geq \left\lceil \frac{n_q}{N} \right\rceil e \\ \implies & n_q \geq \left\lceil \frac{n_q}{N} \right\rceil \, . \end{aligned}$$

But $n_q \geq \lceil \frac{n_q}{N} \rceil$ is always true, which means that LSMR is always not slower than its sequential counterpart.

Consider $\Delta = 0$ (so $\Pi = 0$). When the abcast time is dominant then from Theorem 1 we immediately get that LSMR is not slower than its sequential counterpart if

$$(n - \beta)e \geq \frac{n_{rw}}{\beta} t_r \, .$$

When the execution time is dominant then from Theorem 1 we immediately get that LSMR is not slower than its sequential counterpart if

$$n_q e \geq t_r \, .$$

□

We can reduce LSMR to SMR by failing to distinguish RO and RW requests ($n_q = 0$) and suppressing multicores ($c = 1$). Then, we get as expected:

Theorem 5. *A service replicated using SMR and executed on N servers is always slower than its sequential counterpart.*

Proof: In SMR, we do not distinguish between RO and RW requests, so $n = n_{rw}$ (or $n_q = 0$) and $\Pi = \Delta = 0$. Hence, by Lemma 4, when the abcast time is dominant, SMR is not slower if $(n - \beta)e \geq \frac{n}{\beta} t_r$. From definition of abcast

time dominance $\frac{n}{\beta}t_r \geq ne$. Hence $(n-\beta)e \geq ne$, so $0 \geq \beta e$, but then we get contradiction since $\beta > 0$ and $e > 0$. When the execution time is dominant, SMR is not slower if $n_q \times e \geq t_r$, so we get $0 \geq t_r$ since $n_q = 0$, but this is false since $t_r > 0$. Hence SMR is slower than its sequential counterpart. □

4 State Machine Replication with Multiversioning (MvSMR)

In this section, we propose *State Machine Replication with Multiversioning (MvSMR)*—a variant of the state machine replication scheme that uses multiversioning instead of readers/writers locks to optimize read-only requests. We are not aware of any concrete implementation of such a system, so our proposal is arbitrary. To describe MvSMR, we use the assumptions and rules 1-3 describing SMR (see Section "State machine replication: SMR and LSMR" of the article), but restrict rules 1 and 2 to RW requests only, and add new rules:

4) request types (RO or RW) are known *a priori*,
5) RW requests are executed sequentially and in the same order by all servers. A RO request is processed in parallel with other RO/RW requests by any (but one) server, thus no coordination is required,
6) once a RW transaction T_x completes, an immutable version of each object that was modified by T_x is created, to be accessed exclusively by any future RO transactions. Each RO transaction T_y can only access one version of a given object—the last one before T_y commenced.

Note that if RW transactions were allowed to execute on each server in parallel (optimistically), they might conflict. Conflicting transactions would have to be aborted and reexecuted. Moreover, distributed agreement coordination would be required to ensure consistency among servers. Thus, effectively the MvSMR replication scheme would become very much similar to MvDUR. Since the lack of conflicts is one of the main advantages of SMR compared to DUR, we regard this further optimization *in potentia* as counteracting.

The serial execution of RW transactions in MvSMR involves some overhead due to creation of object versions. But contrary to MvDUR, this overhead is smaller since there is no overhead due to transaction certification. Also, maintaining each object only in two copies at a time is sufficient in MvSMR since RW transactions are executed sequentially. Therefore, we neglect this overhead and assume that the time of processing RW and RO requests is the same and equal e. (Similarly, we neglected the overhead of locks in LSMR.)

The potential advantage of SMR with multiversioning when compared with SMR with readers/writers locks is that the former scheme enables more parallelism since RO transactions can be executed in parallel with a RW transaction. However, SMR equipped with multiversioning, like DUR, requires a layer of memory management (to coordinate access to heap/object versions). On the contrary, any service can be easily replicated and deployed on a custom system using basic SMR scheme (with or without readers/writes locks) simply by intercepting and broadcasting requests sent to the service. In LSMR, it is necessary to instrument shared objects with locks, which is, however, much easier to ensure in a custom system than to provide memory management. This simplicity is a great advantage of SMR and LSMR that do not use multiversioning.

The theoretical analysis of the performance of MvSMR and the results obtained are exactly the same as for LSMR in Section "State machine replication: SMR and LSMR" of the article, but instead of Π' one should use Π'_{mv}, which is derived as follows:

$$\begin{aligned}\Pi'_{mv} &= \frac{[\Delta N/(N(c-1)) - t_{rw}^{\text{SMR}}]N(c-1)}{Nc} \\ &= \frac{\Delta - t_{rw}^{\text{SMR}}(c-1)}{c}\end{aligned}$$

and then we have:

$$\begin{aligned}\Pi_{mv} &= \begin{cases} \lceil \Pi'_{mv} \rceil & \text{if } \Pi'_{mv} > 0 \\ 0 & \text{if } \Pi'_{mv} \leq 0 \end{cases} \\ &= \begin{cases} \left\lceil \frac{\Delta - t_{rw}^{\text{SMR}}(c-1)}{c} \right\rceil & \text{if } \frac{\Delta}{c-1} > t_{rw}^{\text{SMR}} \\ 0 & \text{if } \frac{\Delta}{c-1} \leq t_{rw}^{\text{SMR}} \end{cases}.\end{aligned} \tag{25}$$

As expected, the time Π'_{mv} for MvSMR is shorter than Π' for LSMR, and the difference is attributed to $n_{rw}e$ which does not need to be subtracted from t_{rw}^{SMR} (as with readers/writers locks in LSMR), since RO transactions in MvSMR with multiversioning can be now executed in parallel with RW transactions.

5 Experimental Evaluation

In this section, we give a complete description of the results of our experimental evaluation of JPaxos (SMR) and Paxos STM (MvDUR). We also explain how we obtained the experimental data that we used to estimate the ideal throughput, based on the lower bounds from the model. For convenience, we included in this section the figure and some text fragments that appeared in the article; see the article for the summary and conclusions.

We present the results of experimental evaluation of SMR and MvDUR under different workload types and varying contention levels, obtained using popular microbenchmarks: *Hashtable* and *Bank*. For each benchmark, we developed a non-replicated service (*SeqHashtable* and *SeqBank*) executing requests sequentially on one machine, and a replicated, fault-tolerant counterpart, where the program code and data structures (hashtable and bank accounts) were fully replicated on N nodes, each one equipped with a c-core processor. The replicated service was built using JPaxos, which utilizes the state machine replication approach, and Paxos STM, which implements the MvDUR algorithm [1].

As in the original state machine replication approach, JPaxos does not recognize requests types, so it implements the SMR scheme. Paxos STM can execute requests in parallel on multicores. Moreover, multiversioning ensures that RO transactions never conflict (so never abort), thus it implements the MvDUR scheme. Both systems use the implementation of abcast based on Paxos [2], with support of message batching and pipelining.

Hashtable:	RO requests	RW requests
Default	100 get	8 get + 2 put/remove
Prolonged	100 get+1 ms	8 get + 2 put/remove+1ms
High-Contention	100 get	40 get + 10 put/remove

Fig. 1. No. of operations per Hashtable's configuration.

Both systems allow replicas to crash and later recover and seamlessly rejoin. Notably, nonvolatile storage is scarcely used during regular (nonfaulty) system operation. During recovery, a recovering replica can obtain the current state from other live replicas (if at least a majority of replicas is operational all the time). In the paper, we compared the performance of the two systems during regular (nonfaulty) execution. But in all our experiments, we ran both systems with the recovery protocol enabled, so they were fully fault-tolerant.

5.1 Benchmarks

The *Hashtable* benchmark features a hashtable of size h, storing pairs of key and value, and accessed using the *get*, *put*, and *remove* operations. A run of this benchmark consists of a load of requests which are issued to the hashtable. As in the model, we consider two types of requests (transactions): A *read-only (RO)* request executes a series of *get* operations on a randomly chosen set of keys. A *read-write (RW)* request executes a series of *get* operations, followed by a series of update operations (either *put* or *remove*). The Hashtable is prepopulated with $\frac{h}{2}$ random integer values from a defined range, thus giving the saturation of 50%. The saturation level is preserved all the time: If a randomly chosen key points at an empty element, a new value is inserted using *put*; otherwise, the element is removed using *remove*.

We used three Hashtable configurations: Default, Prolonged, and High-Contention, which represent various *workload types*, modeled by varying the number and length of operations in RO and RW transactions (see Fig. 1). In all cases, each RO request scans a vast amount of data using 100 *get* operations. In contrary, RW requests have fewer operations (10 in Default and Prolonged and 50 in High-Contention), where 20% are update operations. In Prolonged Hashtable, each RO and RW transaction is prolonged 1 ms, which simulates a computation-heavy workload for execution time dominance.

The *Bank* benchmark features a replicated array of 250k bank accounts. A run of this benchmark consists of a load of RW and RO requests accessing the accounts. A RW request transfers money between two accounts, by executing two *get* and two *put* operations on the replicated array. A RO request computes a balance, by reading all accounts and summing up the funds.

For each benchmark, we examined three *test scenarios*, obtained by the following mix of RW and RO requests in the test load: 10%, 50%, and 90% of RW requests. RO requests were known *a priori*. In Paxos STM, different test scenarios allow us to simulate variable *contention* in the access to data shared by concurrent transactions.

5.2 Experimental environment

We ran tests in a cluster of 20 nodes connected via 1Gb Ethernet network. Each node had 28-core Intel E5-2697 v3 2.60GHz processor, 64GB RAM, and used Scientific Linux CERN 6.7 with Java HotSpot 1.8.0.

The abcast protocol used by JPaxos and Paxos STM was configured to have at most two concurrent instances of consensus and the batch capacity 64KB. We experimentally established an optimal number of threads in Paxos STM to be 160 for Hashtable and 280 for Bank (these values were used in all our tests). In all of our tests, the number of threads is high and far exceeds the number of physical cores, to compensate for threads that are blocking on I/O (network) operations. This way we can fully exercise the hardware, and show the peak performance of JPaxos and Paxos STM, which corresponds to our model.

To reduce the overhead caused by client-server communication, the clients ran on replicas. The number of clients in JPaxos and Paxos STM was constant per replica, and equal the number of threads in Paxos STM.

5.3 Performance measures and parameters

For all benchmarks, we present *throughput*—the number of handled requests (committed transactions) per second. In Paxos STM-based service, concurrent transactions may conflict and abort, so we also present the *abort rate*, denoted R—the percentage of transactions aborted due to conflicts out of a total number of transaction executions:

$$R = \frac{K}{n+K}100\% \tag{26}$$

where n is the number of requests and K is the number of conflicts. The abort rate gives a useful insight into the level of contention. We also measured data transfer in Mb/s to witness network congestion, as it is a limiting factor in many tests.

In Fig. 4, we give the measured results and the optimal throughput n/T_{lowb} of SMR and MvDUR for $N = 3..20$ replicas, where T_{lowb} was calculated using (1) and (16); n, n_{rw}, n_q and c are constant input data; t_r, t_u, e, e_o, and *message size* are obtained separately for each benchmark and test scenario through measurements of JPaxos and Paxos STM; β_1 = 64KB/*message size*, where 64KB is the capacity of abcast batch, and $\beta_2 = 2$. These parameters are constant in each test scenario. Only K_{rw} (the actual number of conflicts) changes with N. The abcast times t_r and t_u were measured when network was *not* congested. Below we explain in more detail how the values of the parameters were established.

We obtained the experimental and theoretical results for the number of replicas N ranged from 3 to 20, since our Paxos-based abcast protocol requires $N \geq 3$ to make progress. The total number of requests n was chosen to be large enough to compute the meaningful (optimal) throughput, and was constant through all tests. In order to establish values of other parameters, we conducted additional experiments, as follows.

We configured the Paxos-based abcast protocol used by JPaxos and Paxos STM to ran in all tests two parallel instances of the protocol, with the abcast batch capacity equal 64KB, as this was an optimal configuration. Then, we get β_1=64KB/*message size* and $\beta_2 = 2$. Separately for each benchmark and each test scenario, we measured the average size of messages carrying requests in SMR, and read-sets and updates in MvDUR. This way we obtained 24 message

sizes (4 benchmarks $\times$ 3 test scenarios $\times$ 2 systems) which we then used to establish 24 different values of $\beta = \beta_1\beta_2$. The message sizes varied from circa 60B (Bank in JPaxos) to almost 800B (High-Contention Hashtable in Paxos STM).

The values of t_r and t_u in SMR and MvDUR respectively, were measured for the abcast protocol that is part of the JPaxos and Paxos STM implementation. For this, we used the protocol to broadcast a continuous stream of messages filled with a random content, and counted the messages delivered during T seconds (say x). Then, the elapsed time of a single abcast is $t_{\{r,u\}} = \frac{T}{x/\beta_1}\beta_2 = \frac{T}{x}\beta$. We ran this experiment for all message sizes that we measured when executing our benchmarks. In effect, we obtained 12 values of t_r (or t_u), which correspond to the best possible times of abcast in our benchmarks, but established when the network was not saturated, and the processors were not burdened by any additional tasks, as we executed the abcast protocol alone.

However, t_r and t_u do not include the time of the tasks that are required to prepare a message for a broadcast or use (message marshalling/unmarshalling), and for inter-thread communication, as a part of the regular system operation. This was intentional, since, according to the performance model, t_r and t_u describe only the abcast activity which can be parallelized with other computation. As such, these values are the optimal times of broadcasting a message atomically using the underlying network infrastructure.

The mean request execution time e in SMR and e_o/e (of RW/RO requests) in MvDUR were measured separately for each test scenario by running the actual benchmarks under JPaxos and Paxos STM, respectively. We used the `ThreadMXBean` interface for managing threads in JVM to obtain CPU times. In JPaxos, e was measured as the CPU time of processing the code of a request, from the receipt of a request till a response to the request was generated. In Paxos STM, the request execution time is the CPU time required by a thread to process a transaction from the transaction start till the transaction commit is executed, but before the abcast is launched, so this time does not include the time of the final certification and installing the updates since these two operations are executed after abcast message delivery. The measured times incorporate the overhead mentioned in the definition of t_o (i.e., transaction certification, creating/removing object copies, and collecting read-sets and updates). But both in SMR and MvDUR, the overhead of inter-thread communication and message (un)marshalling were not measured, as they do not logically belong to a request.

To calculate the MvDUR throughput, the number of conflicts K_{rw} per n requests was counted in each run of the Paxos STM test. K_{rw} can also be calculated from the abort rate using the following formula:

$$K_{rw} = \frac{nR}{100\% - R} \tag{27}$$

which was derived from (26) taking $K = K_{rw}$, where n is the total number of requests (or committed transactions) and R is the abort rate (given in %).

The average number of conflicts per update transaction (or the average number of times each update transaction is reexecuted), denoted k_{rw}, can be derived from (27), as follows:

$$k_{rw} = \frac{K_{rw}}{n_{rw}} = \frac{nR}{(100\% - R)n_{rw}} = \frac{R}{(100\% - R)p} \tag{28}$$

where $p = \frac{n_{rw}}{n}$ equals 0.1, 0.5, and 0.9, respectively, for the 10%, 50%, and 90% RW test scenarios. In Fig. 2, we show how k_{rw} changes in Paxos STM-based benchmarks with an increasing number of nodes, considering all three test scenarios (we only show example nodes for illustration). The results showed some variation, due to factors beyond us, e.g. the variable speed of Java garbage collector. Interestingly, given any Hashtable benchmark and a cluster size, k_{rw} is similar across test scenarios. Of course, the abort rate is larger for 10% RW than for 90% RW, but since the number of RW requests also increases, the average number of conflicts per update transaction is similar in our benchmark configurations.

Once we obtained the values of all parameters, as described above, we used formulas (1) and (16) to calculate the lower time bounds T_{lowb} on processing n requests. Depending on the benchmark used, a given test scenario, and the number of replicas, the concrete values of parameters revealed either abcast time dominance or execution time dominance for SMR and MvDUR, as summarized in Table 3. The optimal throughput n/T_{lowb} for SMR and MvDUR is given in Fig. 4.

Cluster size →	3	6	9	11	14	17	20
a) H. Default							
10% RW	0.41	0.89	1.33	1.63	1.98	1.90	2.35
50% RW	0.42	0.91	1.34	1.65	1.95	1.93	2.56
90% RW	0.42	0.91	1.34	1.65	1.97	1.92	2.29
b) H. Prolonged							
10% RW	0.15	0.43	0.76	1.00	1.45	2.15	2.42
50% RW	0.39	0.91	1.38	1.68	2.12	2.07	2.49
90% RW	0.41	0.95	1.41	1.72	2.13	2.05	2.46
c) H. High-Cont.							
10% RW	5.08	10.24	15.91	19.15	18.99	22.05	25.59
50% RW	5.09	10.27	16.02	18.41	18.00	19.74	25.03
90% RW	5.10	10.34	15.98	19.09	20.26	19.85	25.99
d) Bank							
10% RW	0.00	0.00	0.00	0.00	0.00	0.00	0.01
50% RW	0.01	0.01	0.02	0.03	0.04	0.05	0.06
90% RW	0.01	0.02	0.03	0.04	0.05	0.07	0.08

Fig. 2. k_{rw}—the number of conflicts per update transaction in the Paxos STM-based benchmarks.

Test scenario →		10% RW	50% RW	90% RW
a) H. Default	SMR		3-20a	
	MvDUR		3-20a	
b) H. Prolonged	SMR		3-20e	
	MvDUR	3-14e,15-20a	3-5e,6-20a	3e,4-20a
c) H. High-Cont.	SMR		3-20a	
	MvDUR		3-20a	
d) Bank	SMR		3-20e	
	MvDUR		3-20e	3-6e,7-20a

Fig. 3. The abcast time (a) *vs.* execution time (e) dominance for a number of replicas N=3..20 (3-20).

5.4 Benchmark results

5.4.1 Default Hashtable

Default Hashtable replicated using JPaxos executes all requests sequentially on each node. Thus, it cannot scale

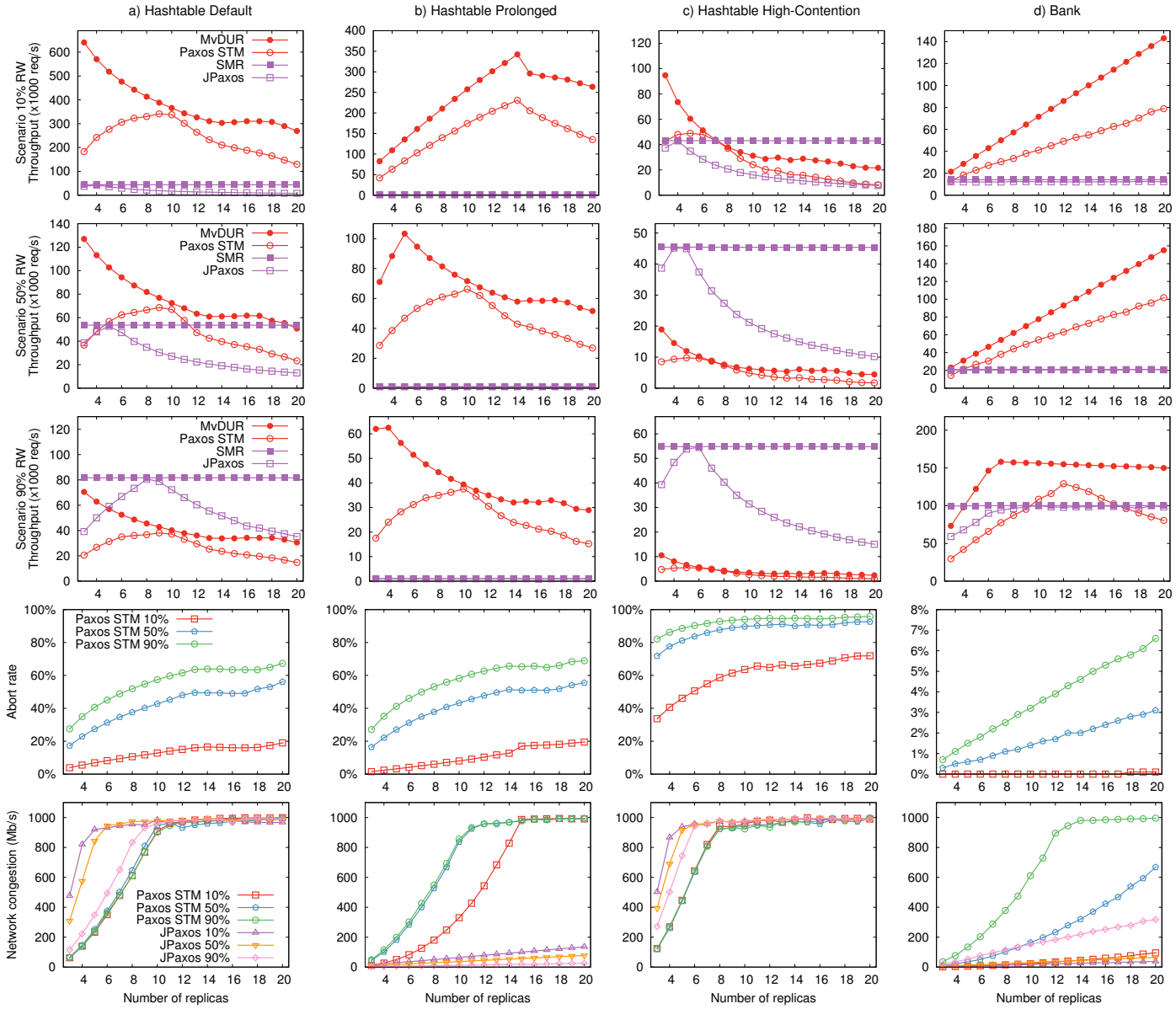

Fig. 4. Model predictions and benchmark results for 10%, 50%, and 90% of RW requests (or transactions).

Test scenario →	10% RW	50% RW	90% RW
a) SeqHashtable Default	657785	1002342	2270151
b) SeqHashtable Prolonged	994	996	998
c) SeqHashtable High-Cont.	468966	528255	650705
d) SeqBank	40477	73174	208853

Fig. 5. Throughput of sequential services (req/s).

with an increasing number of nodes. No transactions are ever reexecuted, since they never conflict. However, the performance of JPaxos is not uniform across the whole range of cluster sizes (see Fig. 4a). In all three test scenarios, JPaxos achieves the peak throughput when the network is nearly saturated (see the network congestion results in Fig. 4a), and later the throughput decreases with an increasing number of nodes, because the network is saturated, so the time of abcasting to a higher number of replicas also grows. On the other hand, before the network gets saturated, the increase in throughput in the low range of cluster sizes can only be attributed to gradually better utilization of the abcast protocol. When the number of nodes is small, the number of clients is also small, as they are collocated with the replicas, and, in effect, there are not enough requests produced by the clients to fully utilize the abcast protocol and harness its potential. Thus, effectively the number of requests per abcast batch was lower than the declared value of β_1, while in the model every instance of abcast carries β_1 requests, where β_1 is chosen to be a constant value such that the abcast protocol is utilized optimally. The results in Fig. 4a show that the execution time in Default Hashtable under JPaxos is *not* dominant, since otherwise the throughput would be constant. Thus, this benchmark is abcast dominated, and processors often wait for requests to arrive. JPaxos achieves the highest performance in the 90% RW test scenario that features the highest percentage of short RW requests, hence we infer that the abcast protocol was best utilized in this

test compared to other test scenarios, i.e., a larger number of requests was delivered per protocol instance.

As expected, the calculated (optimal) throughput of SMR for this benchmark is constant across the range of cluster sizes, because SMR does not scale by Theorem 2. We have used the optimal value of β which is constant per test scenario and across the range of cluster sizes. The optimal throughput is higher for a larger percentage of RW requests, since then the average message size is smaller, thus β_1 (so β) is larger, so the calculated time T_{lowb} of processing requests by SMR is shorter; see (1) and (2) for abcast dominance.

Conversely to JPaxos, the Default Hashtable benchmark replicated under Paxos STM gives better performance for a higher percentage of RO requests. This is not surprising since RO requests do not require agreement coordination, hence the costly abcast operation. The plots obtained through experimental evaluation of Paxos STM have a similar shape as for JPaxos, i.e., the throughput increases until the network is saturated, and later steadily drops as abort rate raises (see abort rate in Fig. 4a). The explanation of this behavior is the same as it was in case of Default Hashtable replicated with JPaxos. The abort rate is low for the 10% RW test scenario, moderate for the 50% RW test scenario, and high (ranging from 27% up to 67%) when 90% of transactions are updating. When Paxos STM had the highest throughput, then for all test scenarios on average every RW transaction was conflicting and had to be reexecuted at least once before it could commit, as $k_{rw} \approx 1.3$ (see Fig. 2). For three replicas, on average every second RW transaction was executed ($k_{rw} \approx 0.4$), while for 20 replicas on average every RW transaction was reexecuted at least twice ($k_{rw} \approx 2.3$). A high number of conflicts means wasted resources and lower overall throughput, which was more than two times worse compared to JPaxos in the same scenario (in the 10% and 50% RW scenarios, Paxos STM always performed better than JPaxos).

The calculated (optimal) throughput of MvDUR shows that Default Hashtable is abcast time dominant for all test scenarios and cluster sizes (see Fig. 3). By comparing the results predicted by the model with the results obtained in the experimental evaluation of Paxos STM, one can see that Paxos STM underutilizes the abcast protocol for a small number of replicas until the network becomes saturated. As it was explained before, the clients which are collocated with the replicas are not able to produce enough requests, so the number of update transactions (only those require agreement coordination using abcast) is not large enough to fully exercise the capability of the abcast protocol. Conversely, the calculated throughput of MvDUR does not expose this behavior since β is constant for each test scenario, and the abcast batch is always full of requests. Thus, for a small number of nodes, MvDUR enjoys the highest throughput since the number of transaction conflicts is minimal. But since the number of conflicts K_{rw} (counted by running the Default Hashtable benchmark with Paxos STM) grows with the cluster size, then also the time T_{lowb}^{MvDUR} increases; see (16) for abcast dominance. Therefore, the optimal throughput of MvDUR diminishes. However, it is higher than the measured throughput of Paxos STM also for a larger number of nodes, since in the model the network is never saturated, thus the abcast protocol is always used optimally (the abcast time t_u is optimal and constant for the whole range of cluster sizes). On the other hand, in the experimental evaluation of Paxos STM, when the network became saturated, the abcast time grew with an increasing number of nodes.

In all evaluation tests, the non-replicated Default SeqHashtable surpasses the performance of JPaxos- and Paxos STM-based replicated counterparts (see Fig. 5a). But this immense throughput is achieved at the cost of no fault tolerance and scaling capability. The throughput values among test scenarios vary since the time of executing RO and RW requests is different. In the best 10% RW scenario, Paxos STM-based Default Hashtable reaches at most about half of the performance of its non-replicated counterpart.

5.4.2 Prolonged Hashtable

In contrast to Default Hashtable, where the abcast time was dominant, the Prolonged Hashtable benchmark aims at mimicking a computation-heavy workload with the execution time dominance. For this, we have used exactly the same set of operations in a transaction as in Default Hashtable, but the execution of each request is prolonged by 1 ms.

Prolonged Hashtable replicated using JPaxos has a stunningly uniform throughput of about 965 req/s, regardless of the number of nodes (see Fig. 4b). This indicates that the lengthy execution time of each request entirely covers up the cost of replica coordination. Since all requests are processed sequentially, the parallel architecture does not bring any gain in throughput. Moreover, the lengths of RO and RW transactions are similar, thus despite the execution time dominance, there is no difference in performance between the 10%, 50%, and 90% RW scenarios. Our performance model correctly predicts the execution time dominance for Prolonged Hashtable with SMR. The calculated throughput is constant and equal 994 req/s. Note that JPaxos does not quite reach the optimal throughput of SMR, due to the overhead caused by inter-thread communication and message (un)marshalling which were not modelled.

In contrast, Prolonged Hashtable replicated using Paxos STM shows excellent scaling. In the 10% RW scenario, the throughput increases with the number of nodes almost linearly, up to the cluster with 14 replicas, when the network becomes saturated. In other scenarios, Paxos STM scales pretty well up to 10 nodes. Then, the network gets saturated and the performance drops. The overall throughput is significantly higher than it is for JPaxos-based Prolonged Hashtable, even for just three nodes. This is credited to Paxos STM's ability of executing transactions in parallel, thus taking the advantage of the multi-core hardware. Note that the abort rate is nearly the same as in the Default Hashtable benchmark. The model predictions for MvDUR are consistent with the results that we obtained experimentally for Paxos STM—the workload is execution time dominant for a small number of replicas, and changes into abcast time dominant for a higher number of replicas. In the 50% and 90% RW scenarios, the calculated results for MvDUR show that the workload is already abcast time dominant from a small number of nodes (see Fig. 3). As it was in case of JPaxos, Paxos STM does not reach the calculated (optimal) throughput of MvDUR, due to the overhead of inter-thread

communication and message (un)marshalling that were not modelled.

The non-replicated Prolonged SeqHashtable has the throughput in the range of 994–998 req/s, depending on the scenario (see Fig. 5b), which is very similar to the one of JPaxos-based replicated counterpart, where all requests are also processed sequentially. This is as expected since the Prolonged Hashtable benchmark replicated using JPaxos has the execution time dominant workload, thus broadcasting a message atomically takes a relatively small amount of time compared to the duration of request execution. In contrast to Default SeqHashtable, the non-replicated Prolonged SeqHashtable does not even come close to the performance of Paxos STM-based replicated counterpart that was superior in all test scenarios and delivered a much higher throughput.

5.4.3 High-Contention Hashtable

The High-Contention Hashtable benchmark aims at testing a replicated system in a high contention environment. Thus, compared to Default Hashtable's configuration, the number of read and update operations in RW requests grew 5 times: to 40 *get* and 10 *put/remove* operations. Note that the execution time of a RW transaction is longer in High-Contention Hashtable than in Default Hashtable. However, it is still much shorter than the time of abcast, which is dominant in this benchmark both for JPaxos and Paxos STM.

The High-Contention and Default Hashtables, which were replicated using JPaxos, have a very similar performance (see Fig. 4c). The slight difference in the overall throughput stems from the larger size of RW requests in the High-Contention Hashtable benchmark. Therefore, the maximum throughput which is achieved by JPaxos-based High-Contention Hashtable is just a few percent lower compared to the maximum throughput of JPaxos-based Default Hashtable across all scenarios.

The High-Contention Hashtable benchmark, which is implemented using Paxos STM, suffers from a very high contention level (a large number of concurrent transactions try to access the same data). The increased level of contention (compared to contention in Default and Prolonged Hashtables) causes a larger number of transactions to be aborted due to conflicts and reexecuted, which diminishes the overall throughput. The abort rate under Paxos STM, starts from 34% in the 10% RW test run on three nodes, and reaches striking 96% in the 90% RW test run on 20 nodes (see abort rate in Fig. 4c). Therefore, on average, every RW transaction is reexecuted due to conflicts around 5 times in the former case, and 26 times in the latter case, before it finally commits (see Fig. 2). Compared to JPaxos, Paxos STM-based High-Contention Hashtable performs better only in the 10% RW scenario. In other cases, JPaxos greatly outperforms Paxos STM, which is especially visible in the 90% RW scenario, where on average High-Contention Hashtable performs 10 times better under JPaxos than under Paxos STM which suffers from a very high abort rate.

The non-replicated High-Contention SeqHashtable has a visibly worse throughput than Default SeqHashtable (see Fig. 5a,c). The lower performance can be attributed to the higher execution time of RW requests. However, the throughput variation across test scenarios is smaller. This is because the execution times of RO and RW requests are approximately the same. Therefore, since the overall throughput is lower, the difference among test scenarios is less noticeable as the number of RW requests increases. However, in all test scenarios, the performance of the non-replicated High-Contention SeqHashtable service trumps the performance of its fault-tolerant counterparts, and the throughput is a few times larger than the best throughput achieved by the implementations using JPaxos and Paxos STM.

5.4.4 Bank Benchmark

Contrary to the Hashtable benchmarks, the number of operations executed as the result of RO and RW requests are very different in the Bank benchmark: A RO request reads all 250 000 elements of the array, which represent bank accounts. On the other hand, a RW request is very short—it just reads two randomly chosen elements of the array and subsequently modifies them. In effect, the cost of executing a RO transaction is much higher compared to the cost of a RW transaction. However, a huge number of operations executed per each RO request does not lead to large abcast messages in JPaxos, as a RO request only contains a single command to read the bank accounts and calculate a total of funds. Therefore, the processing CPUs have become the bottleneck in this benchmark, not the computer network.

In the 10% and 50% RW test scenarios, the throughput of Bank replicated using JPaxos is constant across cluster sizes (see Fig. 4d), and it is clearly restricted by the CPU time of processing requests. The model predictions for SMR confirm our guess and also indicate the execution time dominance for all three test scenarios (see Fig. 3). In case of the 90% RW scenario, the situation is slightly different. For a small number of nodes (3-7), JPaxos-based Bank experiences the abcast time dominance, while for a larger cluster size, the execution time is dominant. In the range of 3-7 nodes, the throughput grows linearly, similarly as in case of other abcast time dominant tests. This behavior is attributed to a relatively low number of submitted requests, which make JPaxos underutilize the abcast protocol. However, on the contrary to other abcast time dominated benchmarks, when the throughput reaches its peak value, it stays constant instead of deteriorating with an increasing number of replicas. This is because the peak performance is not bounded in this test by the network bandwidth, but rather by the available processing power. In fact, the network is far from being saturated for every cluster size. Hence, for a high number of nodes, the throughput of JPaxos is only limited by the time of processing requests. This has been predicted by our performance model, and the throughput of SMR sets a ceiling for the performance of JPaxos-based Bank (see Fig. 4d). However, the model predicts the execution time dominance for every cluster size, while for a low number of nodes, the JPaxos system was first limited by the abcast protocol rather than by processing time.

Bank implemented using Paxos STM also experiences the abcast time dominance only for the 90% RW scenario. As in the most of other tests, the system improves the performance up to the point when the network becomes saturated. Since the contention levels are relatively small for the 90% RW scenario, the degradation of performance for a

higher number of replicas is relatively benign. In the 10% and 50% RW scenarios, JPaxos-based Bank is execution time dominant and the system scales almost linearly across the whole range of cluster sizes. This behavior can be attributed to the ability of Paxos STM to use the underlying parallel multicore architecture well and process many requests in parallel using a large number of available processor cores. These good results were heavily influenced by the optimizations of the abcast protocol, namely, batching and pipelining. Without these optimizations, the abcast protocol would perform much worse, given a very small size of agreement coordination messages that carry transaction read-sets and updates (each message has only 76 bytes), extremely short RW transaction execution times, and a large number of RW transactions performed concurrently.

Note that the performance results that we got for Bank are quite different from the results obtained by running the Hashtable benchmarks. In all variants of Hashtable, test scenarios with a larger number of RO requests gave a higher throughput for all cluster sizes. Now, the opposite is true—the best results are obtained for the 90% of RW requests. This behavior can be easily explained if we recall that it takes significantly more time to execute a RO request than a RW request.

The non-replicated SeqBank outperforms its replicated counterparts for the 90% RW scenario, with the throughput reaching 208k req/s (see Fig. 5d). However, in both 10% and 50% RW scenarios, SeqBank performs better than JPaxos-based Bank for all cluster sizes, but worse than Paxos STM-based Bank for larger cluster sizes. E.g., in the cluster with 20 nodes, in the 10% RW scenario, SeqBank has half the throughput of Paxos STM-based Bank, and in the 50% RW scenario, it reaches 3/4 of the throughput that is achieved by Paxos STM-based Bank. Note that the implementation of SeqBank does not incur any overhead that is characteristic for the replicated counterparts. This fact, together with a very short time of RW requests compared to RO requests, enabled SeqBank to perform five times better when the number of RW requests grew from 10% to 90%.

ACKNOWLEDGMENTS

The authors would like to thank the reviewers for their useful comments. We also thank the Poznań Supercomputing and Networking Center (PSNC) for providing computing resources. This work was funded from National Science Centre funds granted by decisions No. DEC-2012/06/M/ST6/00463 and DEC-2011/01/N/ST6/06762.

REFERENCES

[1] T. Kobus, M. Kokociński, and P. T. Wojciechowski, "Introduction to transactional replication," in *Transactional Memory: Foundations, Algorithms, Tools, and Applications. COST Action Euro-TM IC1001*, ser. LNCS. Springer, 2015, vol. 8913, pp. 309–340.

[2] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems (TOCS)*, vol. 16, no. 2, May 1998.