Comparison of State-Machine and Deferred-Update Replication

Paweł T. Wojciechowski, Tadeusz Kobus, Maciej Kokociński

Abstract—Distributed replication can improve service or database availability and reliability by moving data closer to users and by processing many requests (or transactions) in parallel on different machines. In the paper, we analytically and experimentally compare two popular transactional replication schemes relying on atomic broadcast: state machine replication (SMR) and deferred update replication (DUR). In the best/worst case analysis, we calculate lower/upper bounds on the running time of SMR and DUR algorithms, considering a parallel architecture with multi-core processors and various optimizations, such as processing of read-only requests differently. In the worst case analysis of DUR, we consider conflict patterns. We then analyze the scalability of SMR and DUR systems. It reflects the capacity of SMR and DUR systems to effectively utilize an increasing number of processing cores. Next, we compare SMR and DUR experimentally under different levels of contention, using several benchmarks. We show throughput and abort rate (in DUR) with the increasing number of processor cores. The key result of our work is that neither system is superior in all cases. 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

Along with the emergence of cloud computing, where services in the cloud must be accessed by a large number of users in parallel, there was an explosion of interest in various approaches to distributed replication. Distributed replication can improve service availability and reliability by moving data closer to the users and processing many client requests in parallel. Essentially, a service is deployed on several interconnected machines, each of which may fail independently, and the replicas are coordinated, so that each one maintains a consistent state view regardless of which request is executed. The common feature of the two replication schemes is that they rely on the fault-tolerant total order broadcast primitive, which is used to make the state updates consistent among all the replicas despite failures. Both schemes gave rise to numerous systems that were developed for full replication of services without resorting to any central coordinator. However, direct comparison of these systems is difficult, since they vary considerably in the underlying architecture and offered guarantees. This motivated us to develop a performance model of both replication schemes, showing their potential advantages but abstracting from implementation details. We focus on parallelisation and show the lower/upper bounds on the time of processing concurrent requests on multicores, which can help to localize the bottlenecks and predict the impact of present and future optimizations. Since neither replication scheme seems strictly better than the other, we propose to combine them in order to improve efficiency.

State machine replication (SMR) [1] is a classical replication scheme where a client request (which may consist of a sequence of read and write operations) is executed by every state machine replica running on a non-faulty server, and the reply can be returned to the client by any of them. SMR coordinates state machine replicas, so that all client requests are executed by every replica in the same order. The services themselves must be deterministic: any replica being in the same state always produces the same effect upon the same request. Therefore, each client can observe a consistent state view regardless of which replica was used.

Deferred update replication (DUR) [2] is another scheme. Here each client request is executed only once by any one non-faulty server, as an atomic transaction that can read or modify local memory (or storage). Once the transaction is finished, upon successful transaction certification, the server consistently propagates the modified state to all other replicas. They can update their local state accordingly. Transaction certification means checking if a transaction does not conflict with any other concurrent transaction (executed on the local or remote server) that has been committed. If any conflicts due to concurrency were detected (either during transaction execution or during transaction commitment), the conflicting trans-
action is aborted and reexecuted.

In DUR, transactions can be committed or, on demand, aborted (and possibly restarted). Note that basic SMR does not offer such a transactional semantics (since rollback is not possible) but the execution of a client request by state machine can be regarded as a rudimentary transaction that is allowed to only commit. Therefore, we propose a common specification of both replications schemes, which defines transactional replication. It consists of two primitives, Request and Response, and several properties describing a replicated service process $P$ and the client-$P$ interaction. Then, in our model, we assume that all protocols implementing the SMR and DUR schemes guarantee these properties.

For a fair comparison of SMR’s and DUR’s performance, we require that both schemes use the same implementation of a fault-tolerant, total order broadcast protocol (or atomic broadcast, or abcast, in short): SMR uses it for forwarding client requests to all servers, and DUR uses it as part of an agreement coordination phase for sending read-sets and any state updates to all servers. As shown in [3], [4], agreement coordination relying on atomic broadcast has several advantages: it prevents deadlocks and allows better scalability than when using two-phase commitment (see also [5], [6]).

1.1 Motivations and contributions

The motivations to conduct our research were twofold. Firstly, there was no prior work on the rigorous evaluation and comparison of the SMR and DUR schemes. Secondly, reasoning about the advantages, limitations and possible optimization paths of these 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, transaction conflicts, and various workload types, make the model quite subtle.

The main contributions of our work are the following:

- We defined a model of SMR and DUR that describes the upper/lower time bounds on the execution of concurrent requests (with no delays), parameterized by the number of CPU cores, the workload type, and conflict patterns (in DUR). We considered optimized schemes, where the read-only requests are processed in an unconstrained way, in parallel with update requests. We then used our model to analytically compare the performance of a replicated service and its non-replicated counterpart;
- Our model shows precisely the impact of increasing parallelism on SMR’s and DUR’s throughput, by means such as optimizing abcast, optimizing the read-only requests, detecting conflicts earlier, and increasing the number of CPU cores;
- We proved a result showing when a service replicated using SMR and DUR can be faster than its sequential counterpart, considering various parameters, such as transaction reexecutions due to conflicts (in DUR);
- We formally analyzed the scalability of optimized SMR and DUR schemes, by inferring the speedup and efficiency of a parallel multi-core system;
- We examined the throughput and scalability of SMR and DUR experimentally, for two microbenchmarks: Hashmap and Bank. To replicate services, we used J Paxos [7]—an efficient SMR-based tool, and Paxos STM—a DUR-based tool that we have developed for conducting our research. Both tools use the same implementation of atomic broadcast based on the Paxos algorithm [8];
- Our performance model shows clearly that neither replication scheme is superior in all cases. Therefore we proposed to merge both schemes into a combined scheme that runs a client request either in the SMR or DUR mode, depending on the current workload. We call this novel scheme hybrid transactional replication (HTR). In [9], we give the HTR algorithm and show experimental results obtained using HTR-enabled Paxos STM.

This paper is a largely revised and extended version of [10] (e.g., the performance model and so all equations, lemmas and theorems have been changed and also new material has been added, such as a scalability analysis). To our best knowledge, we are the first to formally analyze parallelism and scalability embedded in 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 define the analytical model in §3 and §4. We present our evaluation experiments and compare the performance and scalability of SMR and DUR replication schemes in §5. Finally, we discuss the follow-up work on merging the two replication schemes into a new one in §6, and 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 [2] for a survey). Below we briefly describe some of the work most closely related to ours.

The SMR scheme evolved considerably since its initial formulation in [11], [12], [1]. Our model can be used to describe both the state-machine-based and the quorum-based [13] replication, both relying on a distributed agreement protocol (e.g., Paxos). In the former case, the main idea is to process all requests by all the replicas in the same order. On the other hand, in the quorum-based replication a transaction is executed only if the majority of sites vote to execute it.

The deferred update replication model [2] corresponds to the basic primary-copy replication (see e.g., [14]) that allows many concurrent master replicas. This approach
is also called multi primary-backup replication or multi-master replication. The key idea is that the update transactions are only processed on a master replica, with the updates propagated eagerly or lazily to slave replicas, but many concurrent master replicas are allowed.

For transaction certification, DUR systems often employ Two-Phase Commit (2PC) protocol [15]. Instead, we consider DUR relying on atomic broadcast (see e.g., [16], [17], [3] among others) which avoids blocking and limits the number of costly synchronization steps (see e.g., [3], [4], [17] and also [5], [6]). Various optimizations of the DUR scheme are possible. For instance, in the Postgres-R algorithm [18], designed for database replication, 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 (see also [24], where a more aggressive speculative strategy was used).

In [25], 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.

Sciaccia, Pedone, and Junqueira [26] 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. [27] built on this idea to scale DUR on multi-core processors. In [28], Sciaccia and Pedone researched the application of DUR to georeplicated storage systems, and also discussed two interesting optimizations of DUR for geo-replication which explore delaying and reordering of transactions.

In [29], the authors researched the tradeoffs of replicating data in distributed database systems that use the primary copy approach for handling data replication. They showed that the way of replicating data and the optimal number of replicas are sensitive to the concurrency control protocol used. Replication generally degraded the performance under the pessimistic control (two-phase locking). Under the time-stamp-based optimistic protocol (in which a conflicting transaction is aborted) and semi-optimistic protocol (which is a combination of the two), replication can improve response time. Moreover, with replication, the latter 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-Martinez, Alonso, and Kemme [30] analytically and experimentally compared quorum-based data replication schemes and concluded that the read-one-write-all-available approach outperforms quorum-based replication in most cases. But they have not compared the SMR and DUR schemes.

Paxos STM, our transactional system that we developed to experiment with replication schemes resembles distributed transactional memory systems, but our design is purely based on DUR and SMR. E.g., DiSTM [31] uses either a centralized mutual exclusion protocol or the leases managed by a designated coordinator to serialize concurrent transactions, which both create a bottleneck. Anaconda [32] extends DiSTM with object replication, caching, and a three-phase commit protocol. Similarly to us, D2STM [33] uses an optimistic transaction certification based on abcast and multiversioning. Also the transactional objects are replicated on all nodes, thus eliminating the need of fetching them from remote locations. In the follow-up work, the lease-based mechanism was adopted to limit abort rate under high contention [34]. However, all these systems use an ad hoc replication protocol built on top of a non-distributed TM while our system was built from the ground up to experiment with DUR and SMR.

3 System Model and Properties

Let us define the system model and properties of transactional replication (TR). A replicated process $P = \{P_1, ..., P_N\}$ consists of $N$ processes (or servers/replicas) $P_i$ ($i = 1, ..., N$) running on independent machines (nodes) connected via a network. Each process $P_i$ 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 process $P_i (i = 1..N)$. 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.

We assume a distributed asynchronous system: There is no central coordinator and the processes communicate solely by exchanging messages using bidirectional fail-loss links [35]. 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 the processes. However, we assume availability of a failure detector $\Omega$ [36], which is the weakest failure detector capable of solving consensus in a distributed asynchronous system in which processes or communication links may fail.

We ideally assume a crash-recovery failure model in which processes may crash independently and later on recover and rejoin computation. Processes can recover its local state either from stable storage or other replicas (as in JPaxos [7]). A process is correct if it is eventually permanently up (there is a time after which it never crashes). Otherwise, it is faulty, i.e. unstable or eventually permanently down (there is a time when it crashes and later never recovers) [37].

In addition to processes (servers) there is an unspecified number of clients that can submit requests to any one replica and await responses. We assume that the clients are independent and do not communicate with each other directly. The only possible interaction is through the replicated service. A client request is executed within an atomic transaction, where atomicity means that the transaction’s operations logically occur at a single instant in time, so the intermediate states are not visible to other concurrent transactions. Furthermore, atomicity prevents updates to the state from occurring only partially.

We assume a simple programming interface: To communicate with a replicated process $P$, a client outputs a request message $r$ of the form $<\text{Request } f\mid \text{args}>$, which is then handled by one or more replicas of $P$ (depending on the TR scheme) by executing an atomic transaction $T_x$ identified by a function $f$ that takes arguments $\text{args}$. Once the transaction completed, $P$ returns a response to $r$ to the client using a message $<\text{Response } r\mid \text{res}>$, where the value $\text{res}$ depends on $P$’s state read by $T_x$; if $T_x$ aborted $\emptyset$ is returned.

A client request (query) can be regarded as an atomic transaction that can execute any legal program containing operations $r(o)v$ and $w(o)v$, which respectively, read and write a value $v$ to object $o$ (which is part of the local state replicated on all nodes). We identify two types of transactions/requests: read-only transactions/requests (RO) that may only consist of read operations and update (read-write) ones (RW) that must contain at least one write.

All the writes of a transaction $T_x$ to some objects can be seen by other transactions running on the same replica $P_i$ only after $P_i$ updates state $S_i$, i.e. applies the object modifications to $S_i$. All transactions eventually commit, or explicitly abort. On commit of transaction $T_x$, a replicated state $S$ is updated with all objects modified by $T_x$. On abort, state $S$ remains unchanged.

We say that transaction $T_x$ precedes transaction $T_y$ (denoted $T_x \rightarrow T_y$) iff $T_x$ has completed execution before $T_y$ begins (on the same or other replica). If neither $T_x \rightarrow T_y$ nor $T_y \rightarrow T_x$, then $T_x$ and $T_y$ are concurrent. Concurrent transactions are executed optimistically (i.e., objects are not locked), so they may conflict. An update transaction (or request) $T_x$ conflicts with some concurrent but already committed transaction $T_{xy}$, resulting in $T_x$ being rolled back and reexecuted, if $T_x$ reads any object modified by $T_y$. We call the former transaction conflicting.

Below we define transactional replication in terms of the properties (or guarantees) imposed on the handling of requests by a replicated process $P$ (rules R1-R7) and the interaction between the clients and $P$ (rules C1-C4). In the specification, we use the symbol $\rightarrow$ to denote a causal order relation, defined as follows: if $r_1 \rightarrow r_2$, then request $r_2$ logically depends on result $\text{res}$ returned by $r_1$. We need two causal order properties: R6 for replicated process $P$ (which only concerns update transactions) and C4 for interaction between $P$ and the clients.

Properties of a replicated process $P$:

R1: Validity: If a process $P_j$ modifies object $o$ with $v$ during state update, then $w(o)v$ was executed by some process $P_i (i = j \ or \ i \neq j)$ as part of some transaction that commits.

R2: Termination: On commit of a transaction $T_x$, every correct process $P_i$ eventually applies $T_x$’s updates (modified objects) to its local state $S_i$.

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

R4: Agreement: No two correct processes update their state differently as the result of executing a transaction $T_x$.

R5: Atomicity: Operations of a transaction $T_x$ and any $T_y$’s updates to $S$ are performed atomically.

R6: Causal 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: Total 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 client sends a request $r$ to a correct process $P_i$ then replicated process $P$ executes $T_x$ 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 these requests \((r_1, r_2)\) respectively delivered to the client, then \( res_1 \) is delivered before \( res_2 \).

4 Performance Model

4.1 Definitions

We denote by \( n \) the number of all requests (or queries); \( n = n_{r} + n_{rw} \), where \( n_{r} \) and \( n_{rw} \) denote correspondingly, the number of read-only and update requests. \( t \) is the time required to atomically broadcast a request (query) in SMR (denoted \( t_{bo} \)), or read-sets and updates in DUR (denoted \( t_{ua} \)), where a read-set is a set of memory locations (or objects) that were read by a transaction. The times \( t_{q} \) and \( t_{a} \) can significantly differ, depending on the size of transmitted data. \( e \) is the time of executing the code of a transaction/request. \( t_{o} \) is the overhead time of the replication scheme (except broadcast time).

In replication protocols, we can identify the following five phases [2], each of which takes the amount of time given after a comma:

1. Sending a client request (or query), \( q \),
2. Replica coordination, \( t_{rc} \),
3. Request CPU execution, \( e + t_{o} \),
4. Agreement coordination, \( t_{ac} \),
5. Receiving a response (or answer) to request \( q, a \).

To simplify the model, we assume that the CPU time \( e \) of processing a request by a process is the same for all requests. Then, the total time \( T \) of processing \( n \) requests in parallel by a service replicated using scheme \( rs \) is \( q + T^{rs} + a \), where \( T^{rs} = \mathcal{P}(t_{rc}, e + t_{o}, t_{ac}) \), and function \( \mathcal{P} \) depends on the parallelism enabled by the replication scheme and the underlying execution environment.

For the SMR and DUR schemes, we deduce the upper and lower bounds on time \( T^{rs} \), denoted \( T_{upb}^{rs} \) and \( T_{lowb}^{rs} \). These bounds correspond respectively, to the worst and best cases when processing \( n \) given concurrent requests without any delay. In our theoretical model, we consider \( c \) processor cores per server, and abstract away any hardware restrictions on parallel executions. For simplicity, we assume at most \( c \) concurrent threads on each server at any time, and no thread interleaving.\(^1\) Thus, in our analysis, we can take as granted that transactions are indivisible. We also assume that all processes in \( P \) are interconnected using a perfect, reliable network that never saturates. All proofs are in the Appendix.

4.2 State machine replication: SMR and OptSMR

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:

- all requests are executed sequentially by each server in the same global order \( (t_{o} = t_{ac} = 0) \),
- replica coordination (using the abcast protocol) and the execution of requests can occur in parallel.

The above rules hold for the classical SMR scheme that does not recognize the type of requests (as in JPaxos [7]). By incorporating the readers/writers locks optimization, we obtain Optimized SMR (OptSMR), where RO transactions can access a consistent snapshot of object versions with no need for inter-node synchronization. To describe OptSMR, we replace the second point above by:

- request types (RO or RW) are known \( a \) priori,
- RO requests are executed sequentially and in the same order by all servers; each RO request is processed by any (but one) server (thus no coordination is required), in parallel with other RO requests,
- RO and RW requests are processed by replica \( P_i \) \((i = 1..N)\) by threads within a critical section guarded by the shared/exclusive locks (or readers/writers locks) [38]. 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 \( S_i \).

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/upper bound estimation (e.g., Paxos STM uses logical clocks as in [39]; see also [8]). Essentially, clock values are piggybacked on requests and responses, so the order can be established by delaying some actions.

Note that Optimized SMR cannot ensure linearizability [40] 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 transactions RO and RW 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.

In JPaxos and Paxos STM, the atomic broadcast protocol is optimized using batching and pipelining [41]. Batching means broadcasting a batch of several messages (if available) by only one protocol instance. Pipelining [8] allows the Paxos leader to initiate several instances of the protocol in parallel. Then, in our model we use \( \beta_1 \) and \( \beta_2 \) to denote respectively, the number of messages broadcast per \( t_{q} \) (or per \( t_{a} \)), and the number of concurrent instances of the abcast protocol at a time. We have \( \beta = \beta_1 \beta_2 \).

Below we compute the upper and lower bounds on the total time of processing \( n \) requests by SMR. The lower bound is also obtained for OptSMR. For simplicity, we
assume \( \frac{n}{\beta} = \lceil \frac{n}{\beta} \rceil \) in SMR, and \( \frac{n_{rw}}{\beta} = \lceil \frac{n_{rw}}{\beta} \rceil \) in OptSMR.

### 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 \( \max \) which returns the greater or equal argument of the two):

\[
T_{lowb}^{SMR} = \max(\frac{n}{\beta} t_q, n e) + \delta^{SMR} = \max(\frac{t_q}{\beta}, e) n + \delta^{SMR}
\]

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

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

### 4.2.2 Lower bound for OptSMR

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. However, on multi-core processors \( (c > 1) \), a RO request can also be processed in parallel with other RO requests (but serially wrt. RW requests).

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

\[
T_{lowb}^{OptSMR} = t_{rw}^{SMR} + \Delta, \text{ where } \Delta = \begin{cases} \Delta' & \text{if } \Delta' > 0 \\ 0 & \text{if } \Delta' \leq 0 \end{cases}
\]

where \( \Delta' = \begin{cases} \frac{n_{rw}}{N} e - (t_{rw}^{SMR} - n_{rw} e) & \text{if } \frac{t_q}{\beta} \geq e \\ \left( \frac{\Delta}{c-1} - (t_{rw}^{SMR} - n_{rw} e) \right) & \text{if } \frac{\Delta}{c-1} \leq t_{rw}^{SMR} - n_{rw} e. \end{cases} \) (4)

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

\[
T_{lowb}^{OptSMR} = t_{rw}^{SMR} + \Pi
\]

where \( \Pi = \frac{\Delta}{c-1} - (t_{rw}^{SMR} - n_{rw} e)(c-1) \) and \( \Delta = \frac{\Delta'}{c} \).

Note that extra cores do not make any difference for RW requests since they must be processed sequentially on each node. Note also that if \( c = 1 \), then as expected \( \Pi = \Delta \).

Below we compare SMR and the optimized SMR.

**Lemma 1.** The time difference of processing \( n \) requests by OptSMR compared to SMR is:

\[
T_{diff}^{SMR} = T_{lowb}^{SMR} - T_{lowb}^{OptSMR} = \begin{cases} \frac{n}{\beta} t_q - \Pi & \text{if } \frac{t_q}{\beta} \geq e \\ n_{rw} e - \Pi & \text{if } e \geq \frac{t_q}{\beta}. \end{cases}
\]

Note that by definition of \( \Pi \), if \( n_r = 0 \) then \( \Pi = 0 \). Thus, if \( n_r = 0 \) then \( T_{diff}^{SMR} \) is also 0, as expected.

Based on (7), we can compute the time difference for the simplest system, when \( c = 1 \) (so \( \Pi = \Delta \)) and \( N = 1 \):
Lemma 2. If $N = c = 1$ then
\[
T_{\text{diff OptSMR}}^{\text{SMR}} = \begin{cases} 
\left(\frac{t_q}{\beta} - e\right)n + \beta e & \text{if } \Delta' > 0 \\
n_{\text{rw}} t_q & \text{if } \Delta' \leq 0 
\end{cases}
\] (8)
when abcast time (network communication) is dominant, and
\[
T_{\text{diff OptSMR}}^{\text{SMR}} = \begin{cases} 
t_q & \text{if } \Delta' > 0 \\
n_{\text{rw}} e & \text{if } \Delta' \leq 0 
\end{cases}
\] (9)
when execution time (CPU processing) is dominant.

The time taken by an algorithm to execute on a single-core processor is called the sequential execution time, denoted $T_{\text{SEQ}}$. 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 OptSMR takes at least $T_{Nc} = T_{\text{lowb}}$ time units.

Now we can compare a highly available and reliable replicated system with a non-replicated counterpart that lacks these properties. The time difference of processing $n$ requests by OptSMR on $Nc$ processor cores compared to processing these requests sequentially by one core is:
\[
T_{\text{diff OptSMR}}^{\text{SEQ}} = T_{\text{SEQ}} - T_{\text{lowb}}^{\text{OptSMR}} = ne - t_{\text{rw}}^{\text{SMR}} - \Pi = ne - \max\left(\frac{t_q}{\beta}, e\right)n_{\text{rw}} - \delta^{\text{SMR}} - \Pi. 
\] (10)

Below we give the main results for the optimized SMR:

Theorem 1. In the best case, a service replicated on $N$ multi-core CPU servers ($c \geq 1$) using Optimized SMR is faster than its sequential counterpart if
\[
(n - \beta)e > \frac{n_{\text{rw}}}{\beta} t_q + \Pi 
\] (11)
when abcast time is dominant, and if
\[
n_{\text{rw}} e > t_q + \Pi 
\] (12)
when execution time is dominant.

Proof: It is straightforward by rewriting of (10). □

From the above lemma it is easy to show as below:

Lemma 3. Consider a system with single core CPUs ($c = 1$). Then, a service replicated on $N$ servers using OptSMR is not slower than its sequential counterpart if $\Delta > 0$. If $\Delta = 0$ then OptSMR is not slower if $(n - \beta)e > \frac{n_{\text{rw}}}{\beta} t_q$ when abcast time is dominant, or if $n_{\text{rw}} e > t_q$ when execution time is dominant.

We can reduce OptSMR to SMR by confusing RO and RW requests ($n_r = 0$) and suppressing multicores ($c = 1$). Then, we get as expected:

Theorem 2. A service replicated on $N$ servers using SMR is always slower than its sequential counterpart.

Proof: In SMR, we confuse RO and RW requests, so $n = n_{\text{rw}}$ (or $n_r = 0$) and $\Pi = \Delta = 0$. Hence, by Lemma 3, when abcast time is dominant, SMR is not slower if $(n - \beta)e > \frac{n_{\text{rw}}}{\beta} t_q$. From definition of abcast time dominance $\frac{n_{\text{rw}}}{\beta} t_q \geq ne$. Hence $(n - \beta)e \geq ne$, so $0 \geq \beta e$, but then we get contradiction since $\beta > 0$. When execution time is dominant, SMR is not slower if $n_{\text{rw}} e \geq t_q$, so $0 \geq t_q$, which is false since $t_q > 0$. Hence SMR is slower than its sequential counterpart. □

4.2.3 Upper bound for SMR and OptSMR
In the worst case, there is no concurrency, which means sequential execution, and $\beta = 1$. Then, the upper bound on the time of processing $n$ available requests is:
\[
T_{\text{upb}}^{\text{SMR}} = T_{\text{upb}}^{\text{OptSMR}} = n(t_{\text{rw}} + e + t_{\text{ac}}) = ne + nt_q. 
\] (13)

4.2.4 SMR and OptSMR 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 with increasing the number of processors. A parallel system is scalable if efficiency can be kept constant as the number of processors is increased, provided that the problem size is also increased [42], [43]. Then, we get

Theorem 3. The system replicated using OptSMR scales and using SMR does not scale.

Proof: The speedup and efficiency of a system replicated using OptSMR are given by:
\[
S_{\text{OptSMR}} = \frac{T_{\text{SEQ}}}{T_{\text{lowb}}^{\text{OptSMR}}} \quad E_{\text{OptSMR}} = \frac{T_{\text{SEQ}}}{T_{\text{lowb}}^{\text{OptSMR}}} \cdot \frac{Nc}{n} 
\] (14)
Consider $\Pi > 0$. Then we have
\[
E_{\text{OptSMR}} = \frac{T_{\text{SEQ}}}{T_{\text{lowb}}^{\text{OptSMR}}} \cdot \frac{Nc}{n} = \left(\frac{n_{\text{rw}}}{t_{\text{rw}}} + \Pi\right)Nc = \left(\frac{n_{\text{rw}}}{t_{\text{rw}}} + \Pi\right)Nc. 
\] (15)

For a given problem instance, the efficiency of Optimized SMR drops with increasing the number of processors. In order to ensure that the OptSMR 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, OptSMR scales.

If $\Pi = 0$, then
\[
E_{\Pi=0}^{\text{OptSMR}} = \begin{cases} 
\frac{ne}{(t_{\text{rw}} + \beta e)Nc} & \text{if } \frac{t_q}{\beta} \geq e \\
\frac{ne}{(t_q + n_{\text{rw}} e)Nc} & \text{if } \frac{t_q}{\beta} \leq e
\end{cases} 
\] (16)
In this case, the OptSMR system also scales if the number of RO requests (in the numerator) is sufficient to compensate for increasing the number of cores (in the denominator).

If the system is unoptimized, we confuse RO and RW requests \((n_r = 0)\) and \(\Pi = 0\), so we get

\[
E_{\text{SMR}}^r = \frac{T_{\text{SEQ}}^r \cdot N_c}{T_{\text{lowb}} \cdot N_c} = \frac{n_e}{T_{\text{lowb}} \cdot N_c} = E_{\Pi=0, n_r=n_r^w}^{\text{OptSMR}}.
\]

It is easy to see that the efficiency \(E_{\text{SMR}}^r\) 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 OptDUR

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:

- clients can send their requests to any one non-faulty server (replica), which then processes each request by executing a local atomic transaction (so no replica coordination is needed),
- each transaction operates on its own local copies of replicated shared objects. On commit of a RW transaction, all replicas must first agree upon the globally consistent state and then update the objects. For this, the transaction’s read-set and the updates are atomically broadcast to all servers (replicas) for agreement coordination \((t_{ac} = t_{ba})\). However, no agreement coordination is required for RO transactions, since they do not update state,
- concurrent transactions are subject to lock-free concurrency control. Thus, to avoid any inconsistencies before a transaction executes a read operation and on transaction commit, it must be certified—i.e. the system verifies if the transaction does not conflict with any other concurrent transactions that have been committed. This procedure is performed locally, with no extra network communication required. If certification fails due to some conflicts, the transaction is aborted and reexecuted.

By certifying transactions on every read, any conflicts can be detected as soon as possible, so that a conflicting transaction can be aborted before its completion. For simplicity, we ignore this optimization in the model and assume a uniform CPU execution time which is equal \(e_0 = e + t_o\) for RW and \(e'_0 = e + t'_o\) for RO transactions, where \(t_o\) and \(t'_o\) are the overhead times per transaction of certification and housekeeping operations specific for DUR, such as collecting read-sets and updates. \(t_o\) also includes the mean time of applying the updates to the local state of all RW transactions (taking 0 for aborted ones). We also assume a uniform abcast time \(t_a\).

By incorporating multiversioning, we get Optimized DUR (OptDUR)\(^3\) that requires additional rules:

- 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,
- RO transactions are never conflicting and always commit, so do not need certification; they also do not incur other overhead \((t_o = 0)\).

DUR and OptDUR can be implemented, e.g., using the DUR and MvDUR algorithms [39]. The algorithms are quite subtle, so our performance model only approximates their behaviour 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. 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 compute the lower and upper bounds 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 \((n = n_r^w)\). Then, we include conflicts. Finally, we extend the model with read-only transactions and analyze the optimized DUR.

\(^3\) OptDUR is implemented by, e.g. Paxos STM.

\(^4\) In 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.
In the best case, transactions are executed concurrently by all \( N_c \) processor cores, and the agreement coordination phase and the code of transactions proceed in parallel, too. Then, the lower bound on the time of processing \( n \) given requests has two outcomes, depending on which of the parallel parts in function \( \max \) is greater, as follows.

We begin from the unoptimized abcast (\( \beta_1 = \beta_2 = 1 \)) and later we only consider the optimized abcast protocol, where \( \beta_1 = N_c \). Note that \( \beta_1 > N_c \) may delay abcast, causing objects to be updated less frequently, so increasing likelihood of conflicts and thus downgrading the optimization.

a) unoptimized abcast and \( K = n_r = 0 \):

\[
T_{\text{lowb}}^{K=n_r=0} = \max \left( n_{rw} t_u, \left\lceil \frac{n_{rw}}{N_c} \right\rceil e_o + \delta_{\text{DUR}} \right),
\]

where

\[
\delta_{\text{DUR}} = \begin{cases} 
  e_o & \text{if } \max(n_{rw} t_u, \left\lceil \frac{n_{rw}}{N_c} \right\rceil e_o) = n_{rw} t_u \\
  t_u & \text{if } \max(n_{rw} t_u, \left\lceil \frac{n_{rw}}{N_c} \right\rceil e_o) = \left\lceil \frac{n_{rw}}{N_c} \right\rceil e_o .
\end{cases}
\]

If \( n_{rw} t_u \geq \left\lceil \frac{n_{rw}}{N_c} \right\rceil e_o \) (\( \delta_{\text{DUR}} = e_o \)) then we say that the abcast time is dominant. If \( \left\lceil \frac{n_{rw}}{N_c} \right\rceil e_o \geq n_{rw} t_u \) (\( \delta_{\text{SMR}} = t_u \)) then the execution time is dominant. We illustrate both cases for \( N_c = 2 \), \( n = 3 \), and \( \beta_1 = \beta_2 = 1 \) in Fig. 3.

b) optimized abcast and \( K = n_r = 0 \): The optimized abcast protocol (assumed in all equations in the rest of this section) broadcasts \( \beta_1 \) messages in one instance and \( \beta_2 \) instances of abcast can proceed in parallel, where \( \beta_1 = N_c \) and \( \beta_2 \) is as large as required. Thus, we have:

\[
T_{\text{lowb}}^{K=n_r=0} = \frac{n_{rw}}{N_c} e_o + t_u .
\]

Note from the above that DUR with optimized abcast appears in our model as execution time dominant, regardless of the time required to broadcast a message.

c) Conflicts possible (\( K \geq 0 \)), \( n_r = 0 \), and optimized abcast: If \( K \) transactions have conflicted, each of them must be reexecuted. Thus we have from (19):

\[
T_{\text{lowb}}^{K=n_r=0} = \left\lceil \frac{n_{rw} + K}{N_c} \right\rceil e_o + t_u .
\]

Let us compare processing requests in DUR in parallel and with processing them by DUR sequentially. In the latter case there is no performance gain due to parallelism, but also there is no overhead caused by transaction reexecution due to conflicts (since transactions never conflict). The time \( T_{\text{seq}}^{DUR} \) of processing \( n_{rw} \) requests sequentially can be obtained from (19), by requiring \( N_c = 1 \):

\[
T_{\text{seq}}^{DUR} = n_{rw} e_o + t_u .
\]

Then, the time difference between sequential and parallel processing by DUR is equal:

\[
T_{\text{DIFF}}^{DUR} = T_{\text{seq}}^{DUR} - T_{\text{lowb}}^{K=n_r=0} = n_{rw} e_o - \left\lceil \frac{n_{rw} + K}{N_c} \right\rceil e_o + t_u .
\]

From the above we obtain:

**Lemma 4**. In DUR, processing update transactions in parallel is faster than processing the same transactions sequentially if \( n_{rw} > \left\lceil \frac{n_{rw} + K}{N_c} \right\rceil \), where \( K \geq 0 \).

**Proof**: Straightforward from (22).

Now let us compare a DUR-based replicated system and a non-replicated counterpart that processes all client requests sequentially on a single-core CPU, so it lacks the desired properties of availability and reliability. The difference between the time of processing \( n_{rw} \) update requests by DUR on \( N_c \) CPU cores and the time \( T_{\text{seq}}^{\text{DUR}} \) of processing them sequentially by one core is:

\[
T_{\text{DIFF}}^{\text{DUR}} = T_{\text{seq}}^{\text{DUR}} - T_{\text{seq}}^{\text{DUR}} = n_{rw} e_o - \left\lceil \frac{n_{rw} + K}{N_c} \right\rceil e_o - t_u .
\]

Based on the above we give below the main result for the unoptimized DUR:

**Theorem 4**. In the best case and assuming only RW requests, a service replicated on \( N \) multi-core CPU servers (\( c \geq 1 \)) using the unoptimized DUR is faster than its sequential counterpart if \( n_{rw} e_o > \left\lceil \frac{n_{rw} + K}{N_c} \right\rceil e_o + t_u .
\)

**Proof**: Straightforward from (23).

d) Read-only transactions: In the optimized DUR, RO transactions still require certification and may conflict, but they do not update state, so they do not require the agreement coordination phase. Therefore, we use \( e'_o \) to denote the 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 \( K_r \) (where \( K_{rw} + K_r = K \)), then we can approximate the lower bound by extending (20) as follows:

\[
T_{\text{lowb}}^{K=n_r=0} \approx \left\lceil \frac{n_{rw} + K_{rw}}{N_c} \right\rceil e_o + \tau_1 ,
\]

where

\[
\tau_1 = \max(t_u, \left\lceil \frac{n_{rw} + K_r}{N_c} \right\rceil e'_o ).
\]
4.3.2 Lower bound for OptDUR
In Optimized DUR, read-only transactions never conflict (thus \( K_r = 0 \)) and do not require certification, so we approximate their execution time as \( e \). By modifying (24) we then obtain the lower bound for OptDUR:

\[
T_{\text{lowb}}^{\text{OptDUR}} \approx \left[ \frac{n_{rw} + K_r e}{N_c} \right] e_o + \tau_2, \quad \text{where} \quad \tau_2 = \max(t_u, \left[ \frac{n_r}{N_c} e \right]).
\]  

(25)

If \( t_u \geq \left[ \frac{n_r}{N_c} e \right] \) (so \( \tau_2 = t_u \)), then we say that the abcast dominates RO requests. If \( \left[ \frac{n_r}{N_c} e \right] \geq t_u \) (so \( \tau_2 = \left[ \frac{n_r}{N_c} e \right] \)), then the RO requests dominate abcast.

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

We can easily compare DUR and OptDUR in terms of performance from (25) and (24):

Lemma 5. The time difference of processing \( n \) requests by DUR compared to OptDUR is:

\[
T_{\text{diff}}^{\text{Dur}} = T_{\text{lowb}}^{\text{DUR}} - T_{\text{lowb}}^{\text{OptDUR}} = \tau_1 - \tau_2.
\]  

(26)

Thus, if \( \tau_1 = \tau_2 = t_u \), the time difference \( T_{\text{diff}}^{\text{DUR}} \) is equal zero, which is as expected since if the abcast time dominates RO requests the performance gain due to the RO requests optimization is counteracted by the high cost of broadcasting updates of the last update request.

If RO requests dominate abcast for \( \tau_1 \) and \( \tau_2 \), then the time difference \( T_{\text{diff}}^{\text{DUR}} = \left[ \frac{n_r + K_r e}{N_c} \right] e_o - \left[ \frac{n_r}{N_c} e \right] e \). Then, if we approximate \( e_o = e \), we obtain:

\[
T_{\text{diff}}^{\text{DUR}} = \left[ \frac{n_r + K_r}{N_c} \right] e_o - \left[ \frac{n_r}{N_c} \right] e \approx \left[ \frac{K_r}{N_c} \right] e.
\]  

(27)

Note that if in a DUR-based system no RO requests conflicted (thus \( K_r = 0 \)), then the time difference would be zero, so DUR would behave like OptDUR. However, by equaling \( e_o \) and \( e \) for RO requests we neglected any differences between the time costs of certification in DUR and multiversioning in OptDUR.

4.3.3 Upper bound for DUR and OptDUR
Now let us compute the upper bound on processing \( n \) given requests by a service replicated using the OptDUR scheme with the optimized abcast protocol. In the worst case, all concurrent transactions conflict with each other and there are no RO transactions (\( n_r = 0 \), thus \( n = n_{rw} \)), so OptDUR behaves like DUR.

We consider two conflict patterns, assuming at most \( c \) concurrent transactions on each node.

a) Transactions never wait for conflicts to be resolved:

In this case, only the first transaction commits and the remaining \( N_c - 1 \) 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, we have:

\[
T_{\text{upb}}^{\text{DUR}} = e_o + (n - N_c) e_o + (N_c - 1) e_o + t_u = n e_o + t_u.
\]  

(28)

The number of conflicts is:

\[
K_r = (n - N_c)(N_c - 1) + \frac{(N_c - 1)N_c}{2} = n(N_c - 1) + \frac{N_c - (N_c)^2}{2}.
\]  

(29)

b) Transactions wait until conflicts are resolved:

In this case, we execute the first \( N_c \) transactions in parallel. Since they all conflict, we wait with processing the next batch of \( N_c \) 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 \( N_c \), we use \( \mu \) to describe their time of execution. Thus, we have:

\[
T_{\text{upb}}^{\text{DUR}} = \left[ \frac{n}{N_c} \right] + (N_c - 1) \left[ \frac{n}{N_c} \right] + \mu e_o + t_u,
\]

where \( \mu = n - \left[ \frac{n}{N_c} \right] N_c \)

\[
= n e_o + t_u.
\]  

(30)

The number of conflicts is:

\[
K_b = \frac{(N_c - 1)N_c}{2} \left[ \frac{n}{N_c} \right] + \frac{(\mu - 1)\mu}{2}
\]

\[
= \frac{(N_c)^2}{2} \left[ \frac{n}{N_c} \right] + \frac{\mu^2 - n}{2} \approx \frac{n(N_c - 1)}{2}.
\]  

(31)

Thus, the upper bound is the same for both conflict patterns. However, the number of conflicts differ since if \( n \geq N_c \) then \( K_b < K_r \), which also means that in case b) less processor power is used. Note that the upper bound is the same as the time of processing all transactions sequentially by DUR (see (21)), but in the latter case \( K = 0 \), which means that even less processor power is used.

Below we give the positive result for both schemes:

Theorem 5. The total time of executing by DUR (or OptDUR) \( n \) given transactions that may conflict and thus be reexecuted, assuming at most \( c \) concurrent transactions on each node, is never longer than the sequential execution of these transactions by DUR (or OptDUR).

Proof: We already showed in this section that the upper bound on the time of processing concurrent transactions that all conflict is equal the time of their sequential execution by DUR and OptDUR. The rest of the proof is by contradiction. Let us assume that in the sequential execution history some transactions are conflicting, so they are reexecuted, thus giving the upper bound greater than \( n e_o + t_u \). However, by the conflict definition in §3, the conflicting transactions must be concurrent. But this is not possible since in the sequential history for any two transactions \( T_x \) and \( T_y \), either \( T_x \rightarrow T_y \) or \( T_y \rightarrow T_x \).
4.3.4 OptDUR scalability

Note that the number of transaction conflict depends on the number of requests, and also on the number of processors/cores. We say that the number of conflicts explodes for a given parallel system if it grows at least linearly with the problem size. For instance, $K_a$ and $K_b$ explode. Then, we get

**Theorem 6.** If the number of conflicts does not explode, then the system replicated using OptDUR scales.

**Proof:** The best parallel algorithm matching the specification of OptDUR takes at least $T_{j_{lowb}}^{OptDUR}$ time units. The speedup and efficiency a system that is replicated using OptDUR are respectively:

$$s_{OptDUR} = \frac{T_{Seq}^{OptDUR}}{T_{j_{lowb}}^{OptDUR}}$$

$$E_{OptDUR} = \frac{T_{Seq}^{OptDUR}}{N_c} \approx \frac{\frac{nc}{\sigma + t_u N_c} + \frac{nc}{\sigma + t_e N_c}}{n} \approx \frac{nc}{\sigma + K_{rw} t_u e}$$

where $\sigma = (n_{rw} + K_{rw}) e_o$.

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

$$E_{OptDUR} \approx \frac{n}{n + K_{rw} t_u}$$

Thus, if abcast dominates RO requests (i.e., $t_u \geq \frac{nc}{\sigma + K_{rw} e}$), then the OptDUR-based system scales if the number of requests $n$ is large enough to compensate for increasing the number of cores and the number of conflicts does not explode. If RO transactions predominate ($t_u \leq \frac{nc}{\sigma + K_{rw} e}$), then the system scales perfectly if the number of conflicts does not explode.

For example, if $K_{rw} = K_{br}$ then the number of conflicts explodes and $E_{j_{lowb}}^{OptDUR} \approx \frac{n}{n + K_{rw} t_u} \leq \frac{2n}{2n_{rw}(1 + N_c)}$, 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 $n_{rw} = 0$, then the efficiency is $\frac{2}{1 + N_c}$. 

5 EXPERIMENTAL EVALUATION

In this section, we present the results of experimental evaluation of SMR and OptDUR under different workload types and 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) are duplicated on $N$ machines, each one equipped with a $c$-core processor. The replicated service was implemented in two variants: using JPaxos (SMR) and Paxos STM (OptDUR).

Both TR systems use the implementation of abcast based on Paxos [8], with support of request batching and pipelining. Typically for SMR, request types are not recognized by JPaxos, so it ensures the unoptimized SMR. Paxos STM can execute requests in parallel on multi-core processors. Moreover, multiversioning ensures that read-only transactions are guaranteed to commit successfully, thus our system supports the optimized variant of DUR.

Both systems allow replicas to crash and later recover and seamlessly rejoin. Notably, nonvolatile storage is scarcely used during regular (nonfaulty) system operation since the current state can be obtained by a recovering replica from other live replicas (if only 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 run both systems with the recovery protocol enabled, so they were fully fault-tolerant.

The Hashtable benchmark provides 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 (or transactions) which are issued to the hashtable. As in our model, we consider two types of transactions: A read-only (RO) transaction atomically executes a series of get operations on a randomly chosen set of keys. A read-write (RW) transaction executes a series of get operations, followed by a series of update operations (either put or remove). The Hashtaeble is prepopulated with $\frac{h}{2}$ random integer values from a defined range, thus giving the saturation of 50%. This saturation level is preserved during each run of the benchmark: If a randomly chosen key points at an empty element, a new value is inserted using put; otherwise, remove is executed.

We use three configurations of the Hashtable benchmark: Default, Prolonged, and High-Contention, which represent various workload types, modeled by varying the number and length of operations inside RO and RW transactions (see Fig. 4). In all cases, each RO transaction scans a vast amount of data using 100 get operations. In contrary, RW transactions have fewer operations (ten times less in Default and Prolonged and two times less in High-Contention), where 20% are update operations. In Prolonged Hashtaeble, each RO and RW request is prolonged 1 ms which allows us to simulate a computation-heavy workload for execution time dominance.

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. In Paxos STM, different test scenarios allow us to simulate variable contention in the access to data shared by concurrent transactions.

The Bank benchmark provides a replicated array of 10k bank accounts. A run of this benchmark consists of a load of RW and RO transactions accessing the accounts. A RW transaction transfers money between two

<table>
<thead>
<tr>
<th>Hashtaeble:</th>
<th>RO requests</th>
<th>RW requests</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
<td>100 get</td>
<td>8 get + 2 put/remove</td>
</tr>
<tr>
<td>Prolonged</td>
<td>100 get+1 ms</td>
<td>8 get + 2 put/remove+1ms</td>
</tr>
<tr>
<td>High-Contention</td>
<td>100 get</td>
<td>40 get + 10 put/remove</td>
</tr>
</tbody>
</table>

Fig. 4. No. of operations per Hashtaeble's configuration.
accounts, by executing two get and two put operations on a local replica of the array. A RO transaction computes a balance, by reading a local array and summing up the funds. As before, Bank was evaluated under three ratios of RW to RO requests: 10%, 50%, and 90%.

We run tests in a cluster of eight nodes connected via a private 1Gb Ethernet network. Each node had Intel Xeon X3230 2.66 GHz, L2 cache 2x4 MB CPU, 4GB RAM ECC DDR2, 800MHz Quad-Core processor, running OpenSUSE 10.3 (kernel 2.6.22.19) with Sun JRE 1.6.0. All four processor cores were available for use.

JPaxos was configured to have at most two concurrent instances of consensus, the maximum batch size 64KB, and no batching delay. We experimentally established an optimal number of threads in Paxos STM to be 20 for the Hashtable benchmark and 80 for Bank (these values were used in all our tests). Such a high number of threads (far exceeding the number of physical cores) is necessary to fully exercise the hardware potential due to threads blocking on network I/O operations.

### 5.1 Benchmark results

For all benchmark services and test scenarios, we present throughput—the number of input requests (or transactions) committed per second. In Paxos STM-based service, concurrent transactions may conflict and abort, so we also present the abort rate—the percentage of transactions aborted due to conflicts of all transaction executions (calculated as \( \frac{K}{n+K} \times 100\% \), where \( n \) is the number of requests/transactions and \( K \) is the number of conflicts). The abort rate gives a useful insight into the level of contention. Below we present the results obtained.

The experimental evaluation results in Fig. 5 and 6 corroborate the model predictions in §4. Not surprisingly, the sequential, non-replicated implementations of our benchmarks outperform JPaxos-based replicated counterparts, as predicted by Theorem 2. More interestingly, comparison of Paxos STM and sequential services gave variable results. For instance, Paxos STM was the clear winner for Prolonged Hashtable (see Fig. 5b and 6b). This is credited to low abort rate in this benchmark and Paxos STM’s ability of executing transactions in parallel, thus taking the advantage of the multi-core hardware. By Theorem 4, the performance difference is proportional to the number of conflicts (abort rate) and inversely proportional to the number of cores used. 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 increasing service reliability (and so also availability). JPaxos does not scale (Theorem 3) and is insensitive to the request type since transactions never conflict. It performs poorly compared to Paxos STM when the workload is execution time dominant, as all requests are processed sequentially (see Fig. 5b). In contrast, Paxos STM allows for concurrent transactions that can exploit the underlying parallel architecture. Therefore, the system is able to scale, as formally proven by Theorem 6. This is especially visible in test scenarios involving many RO transactions. However, the OptDUR system suffers under high contention (measured by \( K \)). Then, SMR is clearly a better choice since it delivers predictable and stable performance (see Fig. 5c and High-Contention Hashtable for 50% and 90% RW). Also, the overhead of abcast and network may overshadow the gain of parallelism in OptDUR and, in consequence, reduce scalability. Formally, we have shown this by (33). However, in OptDUR if RO requests dominate over abcast to maintain efficiency at a constant level it is enough to decrease \( K \), which can be achieved by various approaches to contention management.

The results also show that our optimizations of abcast give OptDUR considerable performance boost. This is especially visible in Bank Benchmark (see Fig. 5d) 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 using a single message.
tions can execute concurrently with a SM transaction, thus achieving a high level of parallelism.

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. On one hand, HTR introduces rollback capabilities to the SMR scheme. On the other hand, HTR equips DUR with the support for irrevocable operations, i.e., operations that 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. We tested HTR with three different benchmarks: Bank, Distributed STMBench7, and a Twitter-like social networking service, which generate workloads variable in the level of contention and the size of exchanged messages. The evaluation results in [9] indicate that for all used benchmarks HTR performs as least as well as either SMR or DUR independently, and outperforms them when the number of replicas grows. Specifically, HTR counters the negative aspects of both SMR and DUR and delivers a consistent level of performance.

7 CONCLUSIONS

We studied the potential and limitations of the SMR and DUR replication schemes. The key corollary one can draw from our work is that neither scheme is superior in all cases. This is due to the differences between the two approaches in sensitivity to various workloads. Execution dominated workloads are handled much better when using DUR since this approach can (inherently) 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.

The theoretical predictions were supported by the experimental results. Since only DUR exercises the ability to scale, one would expect it to perform better than SMR. However, the results show that sometimes the overhead of the transactional machinery makes SMR a better choice. One can also observe the high footprint of using either replication scheme compared to the performance.
of a non-replicated (thus prone to failures) variant. However, the fault-tolerance is worth the price. Moreover, the costs of expensive inter-node communication can be partially compensated by parallel request execution in DUR. In workloads that exhibit large request execution times, this may even result in much higher performance of DUR compared to a non-replicated service.

**References**


Paweł T. Wojciechowski received his Ph.D. degree in Computer Science from the University of Cambridge in 2000. 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 holds a Habilitation degree from Poznan University of Technology, Poland, where he is currently an Assistant Professor in the Institute of Computing Science. He has led many research projects and coauthored dozens of papers. His research interests span topics in concurrency, distributed computing, and programming languages.

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

Maciej Kokocinski is currently pursuing a Ph.D. degree and working as a Research Assistant in the Institute of Computing Science, Poznan University of Technology, Poland, where he also received B.S. and M.S. 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.