Early Scheduling on Steroids: Boosting Parallel State Machine Replication

Eliã Batista[a,c,*], Eduardo Alchieri[b], Fernando Dotti[c], Fernando Pedone[a]

[a] *Faculty of Informatics, Università della Svizzera italiana, Via Giuseppe Buffi 13, Lugano, Switzerland*
[b] *Departamento de Ciência da Computação, Universidade de Brasília, Campus Darcy Ribeiro, Asa Norte, Brasília, Brazil*
[c] *Escola Politécnica, Pontifícia Universidade Católica do Rio Grande do Sul, Av. Ipiranga 6681, Porto Alegre, Brazil*

Abstract

State machine replication (SMR) is a standard approach to fault tolerance in which replicas execute requests deterministically and often serially. For performance, some techniques allow concurrent execution of requests in SMR while keeping determinism. Such techniques exploit the fact that independent requests can execute concurrently. A promising category of early scheduling solutions trade scheduling freedom for simplicity, allowing to expedite decisions during scheduling. This paper generalizes early scheduling and proposes a general method to schedule requests to threads, restricting scheduling overhead. Moreover, it explores improvements to the original early scheduling mechanism, namely the use of busy-wait synchronization and work-stealing techniques. We integrate early scheduling and its proposed improvements to a popular SMR framework. Performance results of the basic mechanism and its improvements are presented and compared to more classic approaches, where it is shown that early scheduling with our proposed enhancements can outperform the original early scheduling and other systems by a large margin in many scenarios.

Keywords: State machine replication, synchronization, work-stealing, scheduling

1. Introduction

State machine replication (SMR) is a well-established approach to fault tolerance [1, 2]. In this technique, server replicas execute client requests deterministically, in the same order. Consequently, replicas transition through the same sequence of states and produce the same output for each request. While SMR has been successfully used in many different applications and contexts (e.g., [3, 4, 5]), modern multi-core servers challenge the SMR model since deterministic execution often translates into single-threaded replicas. In order to address this limitation, several techniques have been proposed (e.g., [6, 7, 8, 9, 10]). Techniques that introduce concurrency in SMR build on the observation that *independent* requests can execute concurrently while *conflicting* requests must be serialized and executed in the same order. Two requests conflict if they access common state and at least one of the requests is an update; otherwise the requests are independent. An important aspect in the design of parallel state machine replication (P-SMR) is how to schedule requests on threads, while respecting conflict requirements. Proposed solutions fall in two main categories: late and early scheduling.

With the *late scheduling* approach, requests are assigned to working threads after the requests are ordered across replicas. In [11], for example, each replica has a directed dependency graph that stores not-yet-executed requests and the order in which conflicting requests must be executed. A scheduler at each replica delivers requests in order and includes them in the dependency graph. Threads remove requests from the graph and execute them respecting the dependencies. In late scheduling, the scheduler and threads contend for access to the shared graph, causing synchronization overhead. It has been observed that the cost of tracking dependencies and the synchronization overhead may outweigh late scheduling's concurrency advantage [12].

In *early scheduling* [12], on the other hand, the rationale is to expedite scheduling decisions, even if at the cost of some reduction in concurrency. The idea is that part of the scheduling decisions are made before requests are ordered by the clients (or client proxies). Clients classify requests according to specific classes, derived from application semantics. At the server side, requests are assigned to a worker thread in constant time based on the request's class. For instance, consider a service based on the typical readers-and-writers concurrency model. Scheduling becomes simpler if we adopt the following execution models: any read request is scheduled on any thread; each write request is scheduled on all threads; all threads having the same request must synchronize (e.g., using a barrier) to execute it.

Although previous research has shown that early scheduling can outperform late scheduling by a large margin, specially in workloads dominated by read requests [13, 9, 12], we observe that the early scheduling execution model may restrict concurrency since it assigns requests to specific threads while imposing high thread synchronization on writes. It has been also shown [14] that early scheduling creates unbalanced load on threads, reducing performance. In this paper, we investigate techniques to enhance the early-scheduling execution model. In particular, we extend early scheduling with two well-established strategies: busy-wait synchronization and work-stealing scheduling. The first strategy reduces the cost of synchronization and

*Corresponding author
Email address: delime@usi.ch (Eliã Batista)

the second strategy balances the load among threads. While these techniques are well-established, using them in the context of early scheduling required us to address aspects that had not been previously considered. For example, work-stealing in this case must account for interdependencies between requests.

In a nutshell, this paper makes the following contributions.

- First, we present the early scheduling technique. We explain the notion of classes of requests and show how a programmer can use them to express the allowed concurrency of an application. We present a set of rules to map request classes to worker-threads and the worker-threads execution model. Together these elements ensure linearizable executions (i.e., strong consistency).
- Second, we identify some limitations to the original early scheduling technique, we propose and fully implement enhancements to overcome these limitations. In particular, we generalize the well-known work-stealing technique to account for conflicting requests and investigate the use of a busy-wait approach to eliminate the synchronization costs of barriers.
- Third, we report a large set of experiments that we conducted to compare early scheduling and the proposed enhancements to late scheduling and classical state machine replication based on sequential execution of requests.

This paper continues as follows. Section 2 presents the system model, consistency criteria, and background on P-SMR. Section 3 discusses the original early scheduling and its performance limitations. Section 4 discusses enhancements to early scheduling based on the identified performance limitations. Section 5 reports on our vast experimental evaluation, with the basic model and enhancements. Section 6 surveys related work, and Section 7 concludes the paper.

2. Background

2.1. System model and consistency

We assume a distributed system composed of interconnected processes that communicate by exchanging messages. There is an unbounded set of client processes and a bounded set of replica processes. Each replica implements one (sequential approach) or more threads (parallel approaches) that execute client requests. The system is asynchronous: there is no bound on message delays and on relative process speeds. We assume the crash-stop failure model and exclude arbitrary behavior. A process is *correct* if it does not fail, or *faulty* otherwise. There are up to f faulty replicas, out of $2f + 1$ replicas.

Processes use an atomic broadcast communication abstraction, defined by primitives *broadcast*(m) and *deliver*(m), where m is a message. Atomic broadcast ensures the following properties [15, 16][1]:

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

Our consistency criterion is *linearizability*. An execution is linearizable if there is a total order of its requests that satisfies the following requirements [19]:

- It respects the real-time ordering of requests across all clients. There exists a real-time order among any two requests if one request finishes at a client before the other request starts at a client.
- It respects the semantics of the requests as defined in their sequential execution.

2.2. Request independence

To ensure linearizability, it has been observed that it suffices to execute sequentially only dependent (or conflicting) requests. The independent requests can be executed concurrently without violating consistency [2]. The notion of request dependency or conflict is application-specific. Recently, several replication models have exploited request conflicts to parallelize the execution on replicas.

More formally, request conflicts can be defined as follows. Let R be the set of requests available in a service (i.e., all the requests that a client can issue). A request can be any deterministic computation involving objects that are part of the application state. We denote the sets of application objects that replicas read and write when executing a request r as r's *readset* and *writeset*, or $RS(r)$ and $WS(r)$, respectively.

Definition 1 (Request conflict)**.** The conflict relation $\#_R \subseteq R \times R$ among requests is defined as

$$(r_i, r_j) \in \#_R \text{ iff} \left(\begin{array}{c} RS(r_i) \cap WS(r_j) \neq \emptyset \vee \\ WS(r_i) \cap RS(r_j) \neq \emptyset \vee \\ WS(r_i) \cap WS(r_j) \neq \emptyset \end{array} \right)$$

Requests r_i and r_j *conflict* if $(r_i, r_j) \in \#_R$. We refer to pairs of requests not in $\#_R$ as *non-conflicting* or *independent*. Consequently, if two requests are independent, they can be executed concurrently at replicas.

2.3. Late Scheduling

In this category of protocols, replicas deliver requests in total order and then a scheduler assigns requests to threads. The scheduler must respect dependencies. More precisely, if requests r_i and r_j conflict and r_i is delivered before r_j, then r_i

[1] Atomic broadcast needs additional synchrony assumptions to be implemented [17, 18]. These assumptions are not explicitly used by the protocols proposed in this paper.

must execute before r_j. If r_i and r_j are independent, then there are no restrictions on how they should be scheduled.

CBASE [11] is a protocol in this category, where a deterministic scheduler delivers requests in total order and includes them in a dependency graph (DAG). In the DAG, vertices represent delivered but not yet executed requests and directed edges represent dependencies between them. Request r_i depends on r_j (i.e., $r_i \rightarrow r_j$ is an edge in the graph) if r_i is delivered after r_j, and r_i and r_j conflict.

The DAG is shared with a pool of threads. The threads choose requests for execution from the DAG respecting their interdependencies: a thread can execute a request if it is not under execution and it does not depend on any other requests in the graph. After the thread executes the request, it removes it from the graph and chooses another one.

3. Early Scheduling

Several approaches to P-SMR resort to application semantics to parallelize independent requests. While executing requests in parallel improves performance, the scheduling of these requests introduces overhead. The central idea of early scheduling is to rely on a simple execution model, based on the concept of classes and thread mappings, to avoid the late scheduling overhead. In early scheduling, a request is assigned to a thread based on the request's class. In this section, we present request classes, the execution model, and the class-to-threads mapping of early scheduling. We then introduce algorithms and discuss their performance.

3.1. Request classes

In our model, each class has a descriptor and conflict information, as defined next.

Definition 2 (Request classes)**.** Let R be the set of requests available in a service (same as considered in request conflicts). Let $C = \{c_1, c_2, ..., c_{nc}\}$ be the set of class descriptors, where nc is the number of classes. We define request classes as $\mathcal{R} = C \rightarrow \mathcal{P}(C) \times \mathcal{P}(R)$,[2] that is, any class in C may conflict with any subset of classes in C, and is associated with a subset of requests in R. A conflict among classes happens when any two requests from those classes conflict, according to the conflict definition $\#_R$. Moreover, we introduce the restriction that a non-empty non-overlapping subset of requests from R is associated with each class.

Example. Consider a service partitioned in 2 shards where requests can be classified as read-only and read-write, per shard and globally. Different shards can be read and written independently. We model this application with the following classes, denoted in Figure 1, where classes are nodes and conflicts are edges. Local read classes C_{R1} and C_{R2} in shards 1 and 2, respectively, conflict with the corresponding local write class C_{W1} or C_{W2}, on the same partition, and with the global write class C_{Wg}. The class C_{Wg} also conflicts with itself, with write classes and with global read class C_{Rg}. Local write classes also conflict with themselves and with the global read class C_{Rg}.

[2] We denote the power set of set S as $\mathcal{P}(S)$.

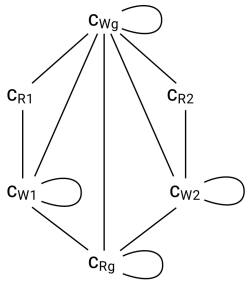


Figure 1: Classes and conflict definition with two shards.

3.2. Execution model

The central idea of early scheduling is to rely on a simple execution model, based on the concept of classes and thread mappings, to avoid the late scheduling overhead by preventing the evaluation of every other pending request when scheduling a new one. To accomplish such a straightforward scheduling algorithm, early scheduling adopts an execution model that will synchronize requests from conflicting classes. A replica will have one scheduler thread and n worker threads. Once a replica delivers (using an atomic broadcast) a request, it is handed over to the scheduler, which then schedules the request to one or more threads.

a. If scheduled to one thread only, r can be processed concurrently with other requests.
b. If scheduled to more than one thread, then r depends on preceding requests assigned to these threads. Therefore, all threads involved in r must synchronize before only one of them (called the *executor*) executes r.

3.3. Class-to-threads mapping

With this execution model, the following class-to-thread-mapping rules must be applied to ensure linearizable executions:

i. *Every class is associated with at least one thread,* to ensure that requests are eventually executed.
ii. *If a class is self-conflicting, it is sequential.* Each request is scheduled to all threads of the class and processed as described in the previous section.
iii. *If two classes conflict, at least one of them must be sequential.* The previous requirement may help decide which one.
iv. *For conflicting classes c_1, sequential, and c_2, concurrent, the set of threads associated with c_2 must be included in the set of threads associated with c_1.* This requirement ensures that requests in c_2 are serialized w.r.t. c_1's.
v. *For conflicting sequential classes c_1 and c_2, it suffices that c_1 and c_2 have at least one thread in common.* The common thread ensures that requests in the classes are serialized.

These rules result in several possible mappings of classes to threads. Creating such a mapping can be modeled as an

optimization problem with the following objectives, detailed in [12]: minimizing the number of threads in sequential classes; and maximizing the number of threads in concurrent classes, while assigning threads to concurrent classes in proportion to their relative weight (i.e., the number of requests expected for these classes). This mapping is static and defined a priori for a given application, hence, it is created only once, at system startup. A mapping is defined as follows.

Definition 3 (CtoT)**.** $CtoT = C \rightarrow \{seq, conc\} \times \mathcal{P}(T)$ where: C is the set of class names; $\{seq, conc\}$ is the sequential or concurrent synchronization mode of a class; and $\mathcal{P}(T)$ the possible subsets of the threads set $T = \{t_0, .., t_{n-1}\}$, n is the number of threads at a replica.

Example. Following our example from Figure 1, considering 4 threads, a possible mapping is depicted in Table 1.

Table 1: A possible mapping of 4 threads in Figure 1

$C =$	$\{seq, conc\}$	$\times \mathcal{P}$ (	$\{t_0,$	$t_1,$	$t_2,$	$t_3\}$	)
$C_{R1}=$	*conc*		$\{t_0,$		$t_2,$	$\}$	
$C_{R2}=$	*conc*		$\{$	$t_1,$		$t_3\}$	
$C_{W1}=$	*seq*		$\{t_0,$		$t_2,$	$\}$	
$C_{W2}=$	*seq*		$\{$	$t_1,$		$t_3\}$	
$C_{Rg}=$	*seq*		$\{t_0,$			$t_3\}$	
$C_{Wg}=$	*seq*		$\{t_0,$	$t_1,$	$t_2,$	$t_3\}$	

The global read class C_{Rg} is defined as *seq* and conflicts with itself to allow more concurrency [12]. By following the class-to-threads mapping rules we observe the following. Since C_{Rg} conflicts with C_{W1} and C_{W2}, and if C_{Rg} were configured as concurrent, all threads assigned to C_{Rg} would have to be included in C_{W1} and C_{W2}. Doing so would synchronize C_{W1} and C_{W2} since their threads would not be disjoint. A more efficient solution (identified by the mentioned optimization model) is to define C_{Rg} as sequential and associate it with one thread from C_{W1} and one thread from C_{W2}. As a result, multi-shard reads synchronize with local-shard writes, but local writes to different shards can execute concurrently.

3.4. Algorithms

Algorithms 1 and 2 present the execution model for the scheduler and threads, respectively. Whenever a request is delivered by an atomic broadcast protocol, the scheduler (Algorithm 1) assigns it to one or more threads. The function *CtoT* returns the set of threads associated with the class. If a class is sequential, then all threads in the set will receive the request to synchronize the execution (lines 4–6). Otherwise, requests are associated with a unique thread in the set (lines 7–8), following a round-robin policy (function *next*).

Each thread (Algorithm 2) takes one request at a time from its queue in FIFO order (line 6) and then proceeds depending on its class synchronization mode. If it is sequential, the thread synchronizes with other ones in the class using barriers before the request is executed (lines 8–14), and only one thread (function *min* returns the thread with the smallest id) executes the request. If it is concurrent, then the thread simply executes the request (lines 15–16).

Algorithm 1 Early scheduler.

```
1:  variables:
2:      queues[0, ..., n − 1] ← ∅                          {one queue per thread}
3:  on deliver(req):
4:      if req.class.mode = seq then                       {if execution is sequential}
5:          ∀t ∈ CtoT(req.class)                           {for each conflicting thread in the mapping}
6:              queues[t].fifoPut(req)                     {synchronize to execute the request}
7:      else                                               {otherwise assigns it to some thread in round-robin}
8:          queues[next(CtoT(req.class))].fifoPut(req)
```

Algorithm 2 Threads for early scheduling.

```
1:  variables:
2:      myId ← id ∈ {0, ..., n − 1}                        {thread id, out of n threads}
3:      queue[myId] ← ∅                                    {a queue of requests}
4:      barrier[C]                                         {one barrier per request class}
5:  while true do
6:      req ← queue.fifoGet()                              {wait until there are requests available}
7:      if req.class.mode = seq then                       {class requires sequential execution}
8:          if myId = min(CtoT(req.class)) then            {if thread has smallest id}
9:              barrier[req.class].await()                 {wait for all threads before executing}
10:             exec(req)                                  {execute request}
11:             barrier[req.class].await()                 {resume other threads}
12:         else
13:             barrier[req.class].await()                 {wait for all threads before execution}
14:             barrier[req.class].await()                 {wait until execution is done}
15:     else                                               {if it is a concurrent execution}
16:         exec(req)                                      {simply execute the request}
```

3.5. Correctness and performance

Safety and liveness are argued in [12], where it is shown that early scheduling produces linearizable executions and ensures that all requests are eventually executed. The performance of early scheduling has been also considered and compared to other approaches in the literature. Although early scheduling performs well in general when compared to late scheduling, the study reported in [14] sheds some light on how the restrictions imposed by the early scheduling execution model affect its performance.

It has been shown that if on the one hand the percentage of time that a thread waits on synchronization increases with conflict rates, on the other hand, as conflicts increase, threads spend less time waiting for new requests to execute. This happens because while waiting on synchronization barriers, requests arrive at the thread queues.

Another finding is that there is a difference in the size of the queues associated with the threads. This is a consequence of the static classes-to-threads mappings. For example, some threads may be associated with fewer request classes than other threads and have, on average, fewer requests in their queues.

Further details about the impact of the restrictions imposed by the early scheduling technique can be found in [14]. Motivated by these findings, in the next section we discuss techniques to boost the performance of early scheduling.

4. Improving early scheduling

In this section, we explore enhancements to early scheduling (also referred to as the basic technique), introducing additional techniques to improve resource utilization. We consider both a variation of the synchronization mechanism, replacing the barriers by a busy-wait approach, as detailed next, and the suitability of work-stealing techniques.

4.1. Busy waiting

With the basic technique, threads synchronize using barriers. A call to a barrier introduces overhead (i.e., a context switch from user-space to kernel-space). When increasing the number of threads, there will be more such calls, and the system will experience performance degradation. The impact of this phenomenon was already observed in [12]. Hence, we consider a busy-wait approach to thread synchronization, aiming to avoid the overhead introduced by barriers.

We modify the original early scheduling thread execution model to keep threads active while synchronizing requests (Algorithm 4). We avoid locks and use atomic variables with the atomic operations described in Algorithm 3. To ensure proper synchronization among threads, we introduce the concept of *cycle* per class. A cycle is the process through which all threads belonging to the same class synchronize to execute one request, and is defined as follows.

Entering a cycle: Threads process their input queue in sequence (Algorithm 4, line 4). If the request is sequential, it has to enter the synchronization cycle. This boils down to atomically incrementing a class variable counting how many threads reached the cycle (line 9) and, either busy-waiting (in line 13) or executing the request on behalf of the class (line 10) if all threads entered the cycle.

Leaving a cycle: A thread waits for the executor to set the request class's *mark* variable to 0 for the corresponding cycle (line 11). Since a released thread could immediately find another request of the same class in its input queue while others are still in the previous cycle, we implement two cycles (0 and 1) per class (see the matrix $mark[classes] \times [2]$ in Algorithm 3) to avoid the faster thread reentering the same cycle, leading to inconsistencies. Each thread keeps track of the next cycle for a class in which it will enter next (Algorithm 4, line 2), and it switches between 0 and 1 whenever it enters a new cycle for a class. This ensures that all threads belonging to a class use the same synchronization atomic variables without interference among cycles.

Algorithm 3 Busy-wait general definitions.

```
1: shared variables:
2:     mark[c_1, ..., c_nc][2] ← [0,0], [0,0]...[0,0]  {one atomic integer per class and cycle}
3:     access functions:
4:         mark[class][cycle].get()                    {atomically read and return the value}
5:         mark[class][cycle].set()                    {atomically set the value}
6:         mark[class][cycle].incGet()                 {atomically increment and return the value}
```

Safety: The barrier in the original early scheduling thread (Algorithm 2) ensures that all involved threads synchronize to execute the sequential request and do not advance before finishing their execution. In the busy-waiting version, a sequential request is executed when all involved threads reach the request in their input queues. After executing, the executor signals the other threads to stop waiting. Thus, the mechanism keeps the same key property during request execution: (a) a request is only executed after all threads have arrived to the request in their queues, and (b) one thread executes the request while the other threads wait for the request execution to finish.

Algorithm 4 Busy-wait at threads.

```
1: variables:
2:     cycles[c_1, ..., c_nc] ← [0, ..., 0]            {array of cycles, one per class}
3: while true do
4:     req ← queue.fifoGet()
5:     class ← req.class                               {assigns variable with the request's class}
6:     if class.mode = seq then                        {sequential execution}
7:         cycles[class] ← 1 − cycles[class]           {recomputes current cycle for the class}
8:         cycle ← cycles[class]                       {assigns variable with the current cycle}
9:         if mark[class][cycle].incGet() = CtoT(class).len then
10:            exec(req)                               {the last thread to increment becomes the executor}
11:            mark[class][cycle].set(0)               {notify other threads that execution is done}
12:        else
13:            while mark[class][cycle].get() ≠ 0      {busy-wait until execution is done}
14:    else
15:        exec(req)
```

Liveness: For a given class, all its threads initiate in cycle 0 and deterministically switch to the next when a sequential request is processed. Since all threads have the same requests of their class in the input queue, eventually all will switch to the next cycle and complete the number of threads to execute it. Moreover, since all threads have the same order of common requests, they will not build cycles while synchronizing to execute different requests. Thus, the synchronization mechanism does not block.

4.2. Work-stealing

Work-stealing is a prominent scheduling paradigm [20] in which underused processors take the initiative to steal work from busy processors. Based on this idea, we propose a work-stealing algorithm for early scheduling. Unlike typical work-stealing approaches, we need to ensure that stealing does not violate the order of conflicting requests to preserve linearizability. More concretely, requests from a victim thread v can only be stolen and executed concurrently by a stealer thread s if all requests in v's queue are independent and do not conflict with requests assigned for execution by s. In the following, we detail these ideas.

Stealer threads. When a thread becomes idle, it turns into a potential stealer. A thread becomes idle in two moments:

- when there are no requests in its input queue; and
- when synchronizing with other threads before executing a request and the other threads involved are not ready.

In any case, the *stealer* will look for work to steal in other threads' input queues; possibly steal one or more requests; execute them; signal their execution; and then resume its usual role, checking its input queue.

Victim threads. Any thread v that has a non-empty queue containing only independent requests is a potential victim. We consider independent requests only to ensure that the pending requests stolen from v's queue can be executed concurrently with the ones currently being executed by v. Notice that to guarantee linearizability, if a thread is executing a sequential request, then its enqueued requests cannot be stolen and executed concurrently with the sequential request.

Stolen work. As discussed above, the stolen requests will be executed concurrently with the ones executed by the victim. Therefore, those requests have to be concurrent.

4.2.1. Algorithms

In this section, we detail the work-stealing algorithm and how it can be integrated in the early scheduling technique to improve performance. We first detail the basic algorithm and then discuss the optimizations and improvements that were incrementally incorporated into it.

Conservative work-stealing. The first and simplest idea is to steal work while waiting for new requests. Algorithm 5 shows general definitions used by our work-stealing algorithms. Algorithms 6 and 7 show the new execution model.

Algorithm 5 Work-stealing general definitions.

```
1: shared variables: consistent under concurrent manipulation (thread-safe)
2:     ∀t ∈ T
3:         readyQueue ← ∅                {separate queue holding requests ready for execution}
4:         execQueue ← ∅                 {separate queue holding requests under execution}
5:         readyFlag ← 0     {1: there is sequential request in readyQueue; 0: otherwise}
6:         execFlag ← 0      {1: there is sequential request in execQueue; 0: otherwise}
7:         marker[t_0, ..., t_{n-1}] ← [0, ..., 0]   {1 at entry s means that s stole requests from t
                                                      and did not finish executing them yet; 0: otherwise}
```

Each worker thread t is augmented with two separate queues (Algorithm 5): 1) the *readyQueue* holds pending requests delivered by the scheduler; 2) the *execQueue* holds the requests that are currently under execution by t (i.e., t transfers requests from the first queue to the second, ensuring that the second queue contains only requests that will be executed by t and cannot be stolen by other threads). Each thread has an array of atomic flags (called *marker*): If a victim v has value 1 at entry s in *marker*, it means that stealer s has stolen work from v and has not finished its execution, otherwise the value is 0. Hence, when v is about to execute sequential requests, it needs to verify these flags to find whether it needs to wait for a stealer to finish.

Algorithm 6 Work-stealing scheduler.

```
1: on deliver(Request: req):
2:     if req.class.mode = seq then
3:         atomic:
4:             ∀t ∈ CtoT(req.class)
5:                 t.readyQueue.fifoPut(req)
6:                 t.readyFlag ← 1            {indicates a sequential request in t's queue}
7:         endAtomic
8:     else
9:         next(CtoT(req.class)).readyQueue.fifoPut(req)
```

The scheduler inserts requests in the threads' *readyQueue*, according to its synchronization class (Algorithm 6). Moreover, the scheduler updates a flag in each related thread when assigning them with sequential requests. This flag signals a sequential request in the *readyQueue* of each thread t.

A stealing attempt will take place if a stealer finds its queue empty (Algorithm 7). The stealing procedure verifies the stealing conditions for each possible victim v (lines 30–32). If satisfied, the thread steals all requests from v's *readyQueue* and sets the marker indicating that requests have been stolen (lines 33–35). Once finished execution, the stealer signals v (line 40) and verifies its own queue again.

Algorithm 7 Work-stealing at thread t.

```
1:  constant:
2:      myId ← id ∈ {0, ..., n − 1}                          {thread id, out of n threads}
3:  Thread t is as follows:
4:      while true do
5:          atomic:
6:              execQueue ← readyQueue                  {transfers all requests to execQueue}
7:              execFlag ← readyFlag          {may indicate execution of sequential request}
8:              readyQueue ← ∅                                          {clear readyQueue}
9:              readyFlag ← 0              {reset flag of sequential requests in readyQueue}
10:         endAtomic
11:         if execQueue ≠ ∅ then                                {if there is something to execute}
12:             while req ← execQueue.fifoGet() do
13:                 if req.class.mode = seq then
14:                     if myId = min(CtoT(req.class)) then
15:                         for all s ∈ T \ myId do              {if it was stolen by someone else}
16:                             wait until marker[s] = 0          {wait until the stealer finishes}
17:                         barrier[req.class].await()
18:                         exec(req)
19:                         barrier[req.class].await()
20:                     else
21:                         barrier[req.class].await()
22:                         barrier[req.class].await()
23:                 else
24:                     exec(req)
25:         else                                  {no requests available, then will try to steal}
26:             Steal(myId)
27: procedure Steal(s ∈ T)                                              {s is the stealer thread}
28:     for all v ∈ (T \ {s}) do                                {tries to steal from all other threads}
29:         atomic:
30:             if v.readyFlag = 0 ∧                                 {only steal concurrent requests}
31:                v.execFlag = 0 ∧          {if victim is not executing sequential requests}
32:                v.readyQueue ≠ ∅ then                  {and there is something to steal}
33:                 s.execQueue ← v.readyQueue                          {steal all requests}
34:                 v.readyQueue ← ∅                          {clear victim's readyQueue}
35:                 v.marker[s] ← 1                                  {signal s stolen from v}
36:         endAtomic
37:         if s.execQueue ≠ ∅ then                                 {steal succeeded, will execute}
38:             for all req in s.execQueue do
39:                 exec(req)                                          {execute all stolen requests}
40:             v.marker[s] ← 0.          {notifies the victim when finished execution}
41:             break for all      {steal and execute once, then try its own queue again}
```

Safety: We argue that the algorithms presented in this section preserve the order of conflicting requests. The key idea is to impose conditions on the contents of queues using flags that mark if the respective queues have conflicting requests. Notice that queues and flags are accessed in the same atomic blocks, ensuring that the flags are consistent with the respective queue's contents.

The stealing procedure ensures that queues have only concurrent requests. When stealing happens, the victim continues execution while the stealer starts processing the stolen requests. From this point, we have two possibilities: either the stealer or the victim finishes processing first. The first case is simple: independent requests were finished concurrently by the stealer and the victim proceeds to process normally. In the second case, the victim will process new incoming requests from its *readyQueue* again. If the new incoming requests are again concurrent, then they can be processed concurrently with the stealer. Otherwise, to process a conflicting request the stealer has to finish first. This is ensured in line 16 of Algorithm 7, stating that the victim will wait for all stealers to finish. The stealer, when finishing processing, will signal the specific victims (line 40 of Algorithm 7).

From the above discussion, we conclude that no conflicting requests are reversed, either by preventing stealing to take place or by having the victim wait for stealers to finish, which are the

only possible cases.

Liveness: By construction, we can observe in Algorithm 7 that, once a thread s steals from a victim v, s unconditionally processes all the contents of its *execQueue*. Also, we observe that a victim, when processing its *execQueue*, either proceeds independently or awaits stealers to finish. Since stealers unconditionally process their contents, eventually the victim will progress. Also, notice that if the victim is waiting for a stealer s_1 to finish, it is because the victim's ready flag is set and it will not become a victim of another stealer s_2. This ensures that once a victim has a conflicting request to process, eventually all current stealers will have finished and no new stealig attempt will succeed, ensuring progress.

We now argue that the execution is deadlock free. Suppose thread t_1 has an empty *readyQueue* and tries to steal from thread t_2, which is executing concurrent requests only from its *execQueue* and has concurrent requests only in its *readyQueue*. In this case, t_1 becomes stealer and t_2 victim. Now suppose that t_2 finishes its *execQueue*, finds its *readyQueue* empty, and tries to steal from t_1. Moreover, assume that t_1 is still processing stolen requests from t_2 but in the mean time its *readyQueue* is populated with concurrent requests. In this case, t_2 steals from t_1 and we have a stealing cycle. Both threads nonetheless will make progress since the stolen work is independent and unconditionally processed. While non-conflicting requests are issued, threads can freely steal from each other as stealers are idle. Depending on the workload, this process may cause threads just to switch work.

Moderate work-stealing. We now present an extension to the conservative work-stealing algorithm to allow for a thread to steal also while waiting for synchronization of a sequential request. By using atomic integers, we replace the first blocking step (barrier) before a sequential request execution (lines 17 and 21 of Algorithm 7) for a non-blocking mechanism. This mechanism is similar to the one described in the busy-wait approach, which provides a way for threads to signal the arrival at a specific class, without being blocked. Such a strategy allows us to identify the last thread to arrive (the executor), and the first arriving threads (the stealers) which can steal work while waiting for all threads to reach this point. After the executor finishes, it signals the stealers and proceeds to the barrier, waiting for synchronization (as in lines 19 and 22 of Algorithm 7). The stealers keep checking if the execution is done, and once it does, they proceed to the barrier as well.

This optimization introduces another opportunity for stealing, in addition to the one described before. However, for the steal to happen now we need an additional restriction: while synchronizing for a request of class c_1, stealer thread s cannot steal requests from any class c_2 that conflict with c_1. This prevents the stealer from reversing the order of conflicting requests, ensuring safety.

To argue for liveness, we recall that due to the stealing conditions, only concurrent requests can be stolen. From the victim's perspective, the order is not violated due to the nature of the requests. Regarding the stealer, the stealing conditions prevent stealing of requests that conflict with the one the stealer is currently waiting for synchronization. In such a case, it cannot steal because it cannot tell the right order among them. Therefore, the same arguments as in the previous algorithm apply: a stealer executes unconditionally; eventually all stealers of a victim finish; if the victim has concurrent requests it continues processing; and if the victim has a sequential request, stealers cease to steal, finish their current stolen works, and the victim synchronizes in the sequential request.

Aggressive work-stealing. In the moderate work-stealing approach, the executor and the stealers synchronize after the execution using a barrier. We describe next how this synchronization barrier can be eliminated.

The scheduler does not need any modifications. For the threads, after the executor has processed the synchronizing request, instead of waiting for all threads to finish their stolen work, they can independently resume processing their input queues, as far as requests are concurrent. While doing so, whenever a synchronizing request is found, the procedure already discussed is adopted. However, now we have to deal with a situation where some threads are still finishing the work stolen previously while others are already stealing again due to a new synchronizing request. We have already solved a similar problem before by introducing cycles in Algorithm 4. Therefore, we use the same mechanisms in this case. For the same reasons as in case of the previous algorithms and from the cycle mechanism, this solution does not reverse the order of conflicting requests.

A further enhancement introduced in this algorithm is the choice of victims. In the previous algorithms, stealers started searching to steal from the thread with the smallest *id* and, if not possible, trying with the next thread, and so on, following the order of thread *ids*. This procedure while simple, possibly led to contention on threads with low *ids*. We modify the search by having a stealer start with the thread *id* that comes next to its own *id* in the space of thread *ids*.

Optimistic work-stealing. Despite the absence of barriers, the aggressive work-stealing approach still has a significant level of contention on the shared state (e.g., queues, flags). We address this shortcoming with optimistic synchronization [21]. In this approach, a thread does not use *locks* while searching for a condition in the shared state. This reduces overhead by decreasing the usage of mutual exclusion mechanisms only to successful situations.

Based on this idea, we propose an optimistic work-stealing algorithm (see Algorithm 8). This algorithm includes all the optimizations we discussed so far. The *Steal()* procedure shows the algorithm augmented with the optimistic validation and the restriction that requests in conflicting classes cannot be stolen. This is ensured by the procedure that checks the conflicts of the request classes. In the case of a conflicting class, there will be no stolen requests (line 56 of Algorithm 8).

Regarding safety and progress conditions, the same arguments discussed in the previous algorithm are valid here. The only change presented is in the process of validation, but validation itself does not change. The stealing procedure execution

Algorithm 8 Optimistic work-stealing.

```
1: constant:
2:     myId ← id ∈ {0, ..., n − 1}
3: variables:
4:     cycles[c_1, ..., c_nc] ← [0, ..., 0]
5:     sync[c_1, ..., c_nc][2] ← [0, ..., 0][0, 0]
6:     access functions:
7:         sync[class][cycle].get()
8:         sync[class][cycle].set()
9:         sync[class][cycle].incGet()
10: Thread t is as follows:
11:     while true do
12:         atomic:
13:             execQueue ← readyQueue
14:             execFlag ← readyFlag
15:             readyQueue ← ∅
16:             readyFlag ← 0
17:         endAtomic
18:         if execQueue ≠ ∅ then
19:             while req ← execQueue.fifoGet() do
20:                 class ← req.class
21:                 if class.mode = seq then
22:                     for all s ∈ T \ myId do
23:                         wait until marker[s] = 0
24:                     cycles[class] ← 1 − cycles[class]
25:                     cycle ← cycles[class]
26:                     if sync[class][cycle].incGet() = CtoT(class).len then
27:                         exec(req)
28:                         sync[class][cycle].set(0)
29:                     else
30:                         while sync[class][cycle].get() ≠ 0 do      {exec not finished yet}
31:                             Steal(myId, class)      {steal informing class of current req}
32:                 else
33:                     exec(req)
34:         else
35:             Steal(myId, null)                          {steal without inform a class}
36: procedure Steal(s ∈ T, c ∈ C)
37:     for all i ∈ [0, ..., T.length] do                  {one attempt for each thread}
38:         v ← pickVictim(s)                                          {choose a victim}
39:         if Validation(c, v) then                  {evaluate conditions without lock}
40:             atomic:                          {lock when steal conditions satisfied }
41:                 if Validation(c, v) then                    {reevaluate conditions}
42:                     s.execQueue ← v.readyQueue
43:                     v.readyQueue ← ∅
44:                     v.marker[s] ← 1
45:                     endAtomic                      {unlock after committing the steal}
46:                     if s.execQueue ≠ ∅ then                  {execute stolen requests}
47:                         for all req in s.execQueue do
48:                             exec(req)
49:                         v.marker[s] ← 0
50:                         break for all                              {stops stealing for now}
51:                 else
52:                     endAtomic                        {unlock when reevaluation fails}
53: procedure Validation(c ∈ C, v ∈ T)          {the validation as a separate function}
54:     return v.readyFlag = 0 ∧
55:         v.execFlag = 0 ∧
56:         (c = null ∨ NoConflict(v, c)) ∧          {verify conflicts with the class (if any)}
57:         v.readyQueue ≠ ∅
```

flow has only been augmented with a pre-validation step (line 39) which is executed without mutual exclusion. The final validation (line 41) will take place if the former succeeds, and it will be consistently executed inside a critical section, ensuring the same properties of the previous algorithm. This strategy resorts to the idea that reevaluation succeeds most of the time, thus called optimistic.

5. Experimental evaluation

We implemented all algorithms described in the previous sections and conducted an experimental performance evaluation.[3] We compare the results with a sequential version, a standard late scheduler (which we implemented based on the algorithm from [11]), and the original early scheduling [12].

5.1. Environment

We implemented all algorithms in BFT-SMART [22], a well-established framework to develop SMR. BFT-SMART was implemented in Java and uses an atomic broadcast protocol that executes series of consensus instances to order sets of requests. BFT-SMART can be configured to tolerate crash-stop failures only or Byzantine failures. In all our experiments, it was configured to tolerate crash-stop failures. To further improve the performance of the BFT-SMART ordering protocol, we implemented interfaces to enable clients to send a batch of requests inside the same message.

The experimental environment was configured with 7 machines connected via 1Gbps switched network. The software installed on the machines was Linux Ubuntu with kernel 4.15.0 (64 bits) and 64-bit Java virtual machine version 11.0.3. BFT-SMART was configured with 3 replicas hosted in separate machines (AMD Opteron® processor with 32 physical cores and 64 logical cores through hyper-threading and 126 GB of RAM) to tolerate up to 1 replica crash, while up to 200 clients were distributed uniformly across another 4 machines (Intel® Xeon® processor with 8 physical cores and 8 GB of RAM).

5.2. Applications

We implemented two applications: a linked list and a key-value store, both supporting disjoint data shards. In our implementation, each shard is represented by separate data structures inside each replica. In the linked-list application, each shard is a separate linked-list object. In the key-value store, each shard is a different tree map object. In both cases, each data structure is accessed only by requests specifically addressed to the respective shard, or by global requests. Replicas order all requests before execution, no matter the target shard. There are requests to read and write in both the list and key-value store, accessing a single or all shards. A *read* request implements a *contains* operation in the list and a *containsKey* operation in the key-value store. A read checks whether an element is in one or all shards. A *write* request implements an *add* operation in the list and a *put* operation in the key-value store. A write adds an element in one or all shards. The write request checks if some element already is in some shard before the addition.

We conducted experiments with single and multiple shards. In a deployment with n shards, there are n local read classes, n local write classes, one global read class, and one global write class. The mappings respect the conflict structure between read and write classes, as in Figure 1. We first present the linked list configuration and results, and then the results of the experiments using the key-value store in Section 5.8.

5.3. Workload configuration

The workload is generated synthetically by the clients. We configured the clients to vary the percentages of reads, writes, local and global requests in the workload from low to high, due

[3] Our source code, and instructions to run our experiments, can be found at https://github.com/elbatista/smr-workstealing.

to the observation that as the level of conflicts increases, it directly affects thread idleness in the original early scheduler [14]. We also consider balanced and unbalanced workloads. In the balanced workloads, each shard receives a similar number of local requests. A local request is a request that executes within only one shard. In the skewed workloads, each client process sends the majority of its requests to only one shard, the remaining requests are equally distributed across the other shards. There are three different request execution costs: *light*, *moderate* and *heavy* (i.e., lists with 1K, 10K and 100K elements, respectively). Clients create a configurable rate of concurrent and conflicting requests, in our experiments from 1% up to 50% of conflicting requests. Clients group requests over time based on an exponential distribution that determines the number of contiguous concurrent requests.

5.4. Single-sharded systems

We start by presenting the results for a single-sharded environment, varying the percentage of conflicting requests, request costs, and number of threads.

5.4.1. Busy-wait algorithm

Figure 2 presents the results of the busy-wait approach. It shows, with 8 to 12 threads, a performance slightly better than the original early scheduling with low conflicts (1% of writes) and *light* costs (Figure 2(a)). In this scenario, there is high concurrency execution in early scheduling, and low barrier synchronization. However, as we increase the number of threads, the overhead in the early scheduling also increases, due to higher rates of system calls caused by the barriers. This phenomenon becomes evident when considering that the busy-wait approach keeps high throughput as we increase the number of threads, while the early scheduling throughput degrades.

Figures 2(b) and 2(c) show that the throughput of the busy-wait approach is about four times higher than the throughput of the original early scheduling when we increase the percentage of writes in the workload (this is a scenario of more overhead and barrier synchronization for the early scheduling). Figure 2(d) shows that, even with *moderate* request execution costs, busy-wait can still outperform early scheduling by a large margin if conflict rates are low.

Figures 2(e) to 2(i), on the other hand, show the impact of increasing request costs and conflict rates. The higher request costs substantially impact the application throughput, bounding all parallel applications to similar performance levels. Furthermore, higher costs lead the parallel techniques to approximate the sequential performance.

5.4.2. Work-stealing algorithms

We now evaluate our work-stealing implementations. Figure 3 compares the performance of each work-stealing algorithm described in Section 4.2 and the early scheduling. The first two work-stealing algorithms (conservative and moderate) do not show performance gains when compared to early scheduling. This result is expected since the stealing opportunities are rare in the conservative version (only when queues are empty). The moderate version, in turn, still has the expensive barrier synchronization.

The aggressive version improves performance by removing the barrier overhead and decreasing contention by using a different stealing strategy to choose victims. Finally, the optimistic version outperforms all other approaches, showing a gain of more than twice the throughput of any other version in some cases. This optimistic version includes all improvements that were incrementally aggregated to each version, and the gain obtained when decreasing overhead by cutting usage of mutual exclusion objects. As it shows the best performance, from now on we present only the optimistic version when referring to work-stealing.

5.5. Busy-wait vs. work-stealing

Figure 4 shows that busy-wait and work-stealing present similar performance. Although work-stealing can afford better load balance, it introduces additional overhead w.r.t. the busy-wait approach. Busy-wait avoids both complex stealing management logic and expensive mutual exclusion resources. On the contrary, it relies on simple atomic variables, which are architecture-native structures, cheaper than the ones used in work-stealing, leading it to present, in most considered cases, a performance slightly better than work-stealing.

Both work-stealing and busy-wait approaches largely outperform the other techniques. The behavior observed for the busy-wait approach is also true for the work-stealing: as the level of conflicts increases, work-stealing performs better, reaching performance about four times higher than the original early and late scheduling algorithms (e.g. Figure 4(b)). Likewise, higher execution costs severely impact work-stealing throughput as well (Figures 4(e) and 4(f)).

5.6. Multi-sharded systems

In our multi-sharded application, client processes randomly choose a shard to issue requests to. We considered balanced and skewed workloads, different numbers of shards, percentage of conflicting requests and percentage of global requests. We set the number of threads to 32 (uniformly distributed among the read and write classes of each shard) and also used only *light* operation costs since this configuration showed, in general, the best performance in the previous experiments. Figure 5 presents results for 2, 4 and 8 shards, with 1% global requests issued by the clients. Similar results were obtained with 15% global requests, hence omitted. The percentage of writes is for both global and local requests (e.g., with 5% of writes, we have experiments with 5% of global write requests and 5% of local write requests).

Varying the number of shards does not have much impact on sequential and late scheduling. However, the same is not observed for early scheduling: the higher the number of shards, the higher the concurrency of requests among shards, thus better the performance. The same occurs for busy-wait and work-stealing. However, in the high conflicting balanced workload scenario (Figures 5(b), 5(c)) our approaches show far better results than other ones, due to the advantage of increasing both

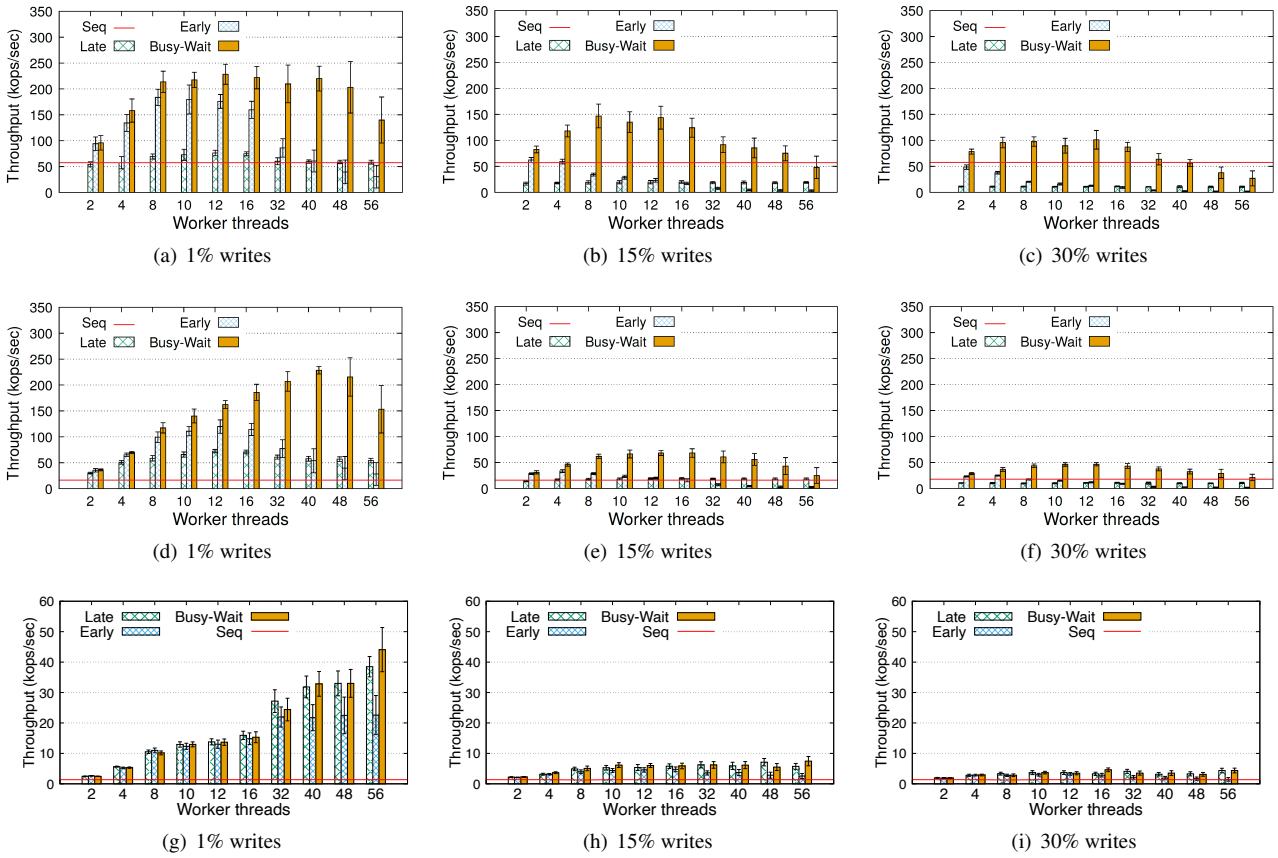


Figure 2: Busy-wait algorithm for single shard, and light (top), moderate (middle) and heavy (bottom) operation costs.

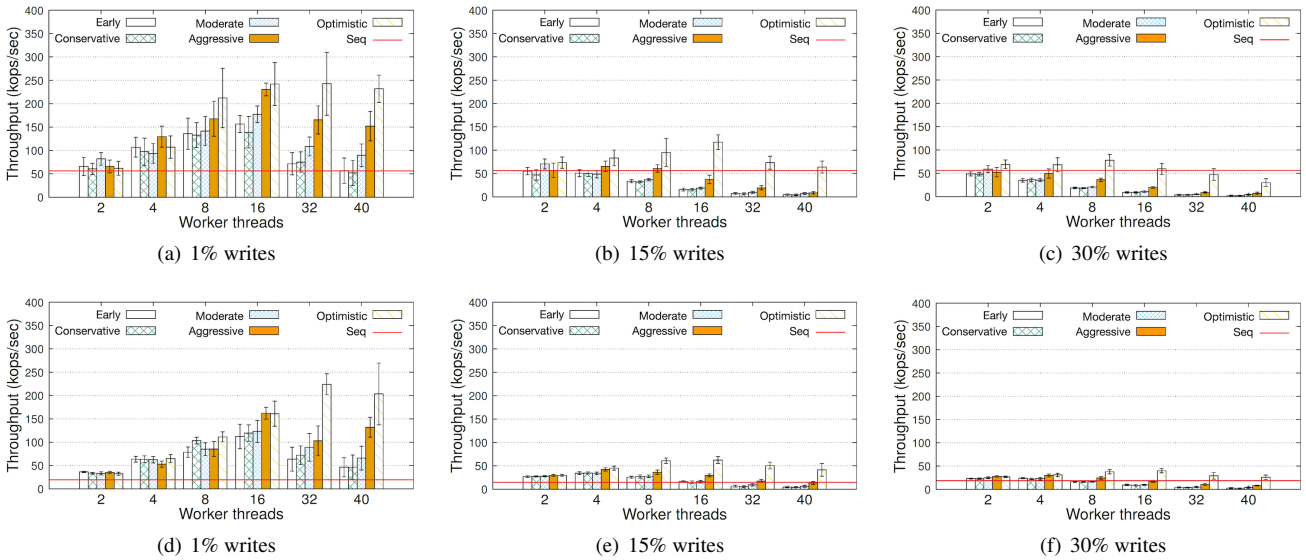
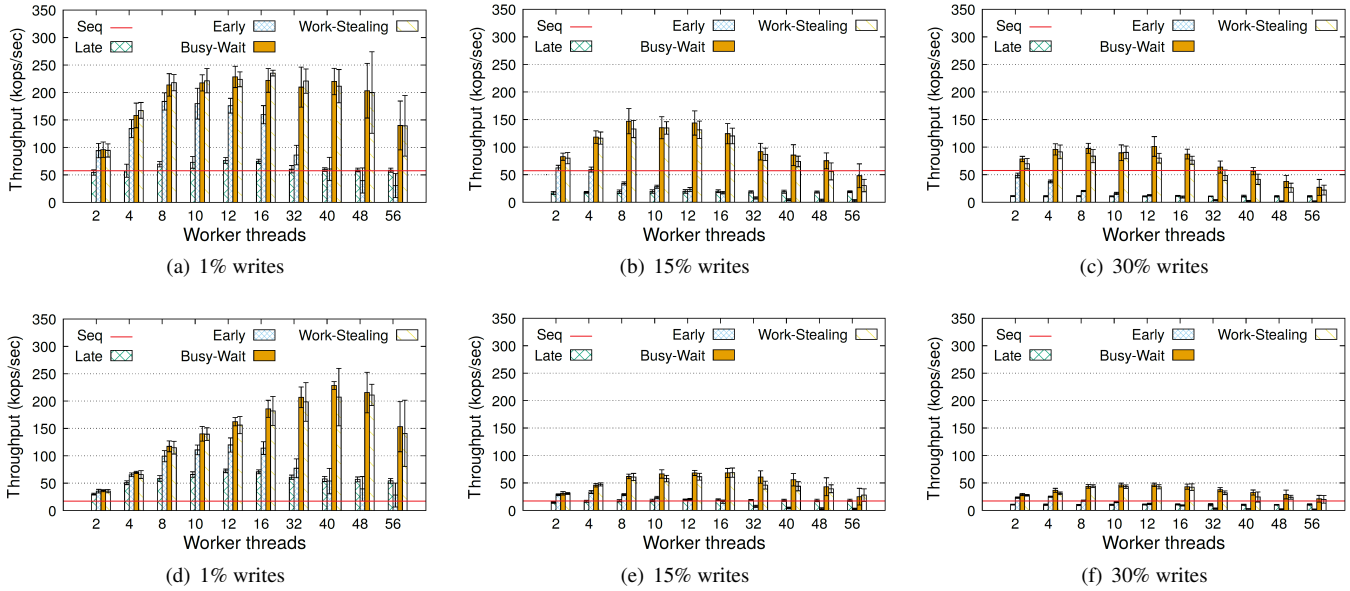


Figure 3: Work-stealing algorithms for single shard, and light (top) and moderate (bottom) operation costs.

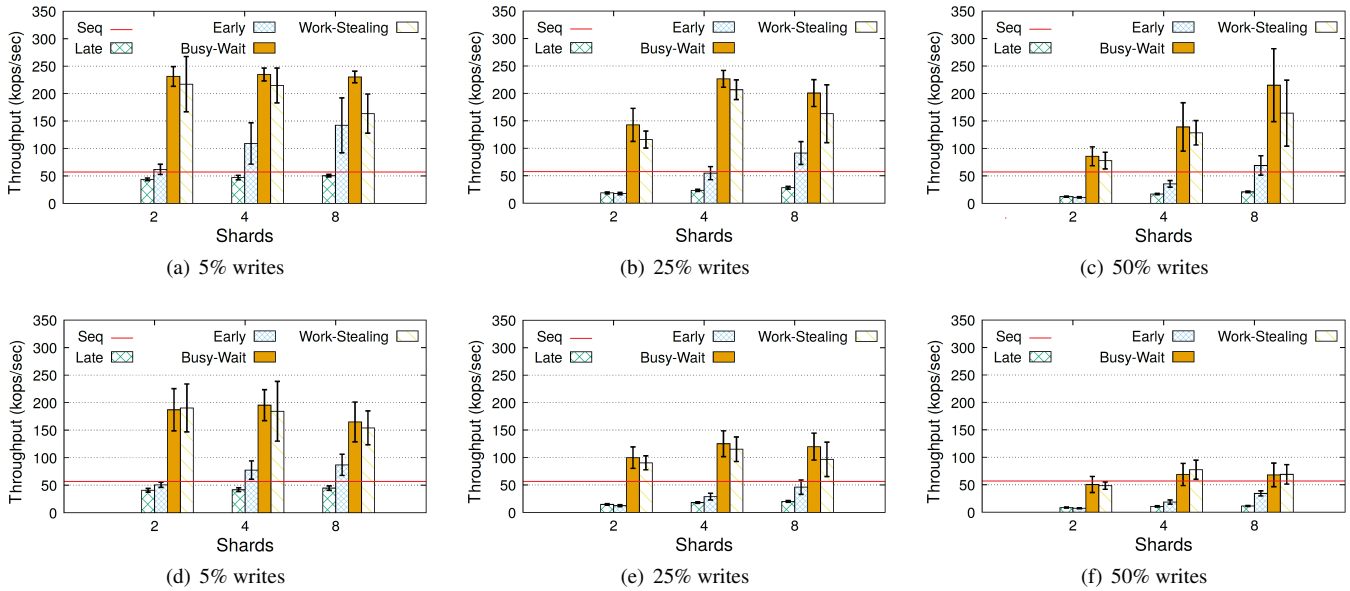
the conflicting requests and the number of shards. Increasing conflicts is not beneficial for the early scheduling (high contention and overhead caused by the barriers), but it is for our approaches. Busy-wait does not suffer the effects of barrier overhead and work-stealing benefits from higher opportunities to steal, either due to the large number of conflicting requests



(a) 1% writes (b) 15% writes (c) 30% writes

(d) 1% writes (e) 15% writes (f) 30% writes

Figure 4: Results for single-shard and balanced workloads, with light (top) and moderate (bottom) operation costs.



(a) 5% writes (b) 25% writes (c) 50% writes

(d) 5% writes (e) 25% writes (f) 50% writes

Figure 5: Results for multi-shard, 1% global requests, balanced workload (top) and skewed workload (bottom).

or the higher number of shards.

Skewed workloads impact more the early scheduling and our approaches than late and sequential. However, the work-stealing approach is less impacted in more concurrent scenarios. Figures 5(a) and 5(d), with 8 shards, show that the throughput of the busy-wait approach falls from about 240k ops/sec in the balanced scenario to 160k in the skewed one. This represents a loss of 44%. On the other hand, the throughput of the work-stealing approach falls from 160k ops/sec to about 150k, representing only 7% of loss. Similar phenomenon occurs for the case with 4 shards and 50% of writes, where work-stealing shows better performance than busy-wait in the skewed scenario. This happens because work-stealing can minimize the skewed effect by distributing requests among threads when there is enough stealing capacity.

5.7. *Additional experiments*

We also conducted additional experiments to evaluate three other aspects of our techniques that we describe next.

5.7.1. Request execution costs

In our experimental evaluation, we analyzed the performance of all studied approaches when exposed to three distinct request execution costs: light, moderate, and heavy. We now quantify these execution costs. Figure 6 presents the time a request waits before its execution starts and the time it takes to complete its execution. We compute the average times and compare the sequential (Seq), early scheduling (Early), busy-wait (B-Wait), and work-stealing (WS) approaches. The *waiting time* starts when a request is delivered by BFT-SMaRt's ordering layer (consensus protocol) and finishes before it starts executing. It includes scheduling and synchronization overhead in the parallel techniques. The *execution time* is the interval between the moment the request is delivered to the application layer and the application logic finishes the execution.

Since we want to understand how much of the total execution time of a request is spent in the actual execution of the request and how much time is overhead, we configured the system with a single client (i.e., in the absence of contention). When execution costs are high, the gains from removing barriers and improving load balancing become irrelevant. In fact, in our experiments, we show that with heavy costs, our optimizations do not improve the early scheduling approach. Moreover, as the workload becomes heavier, and the percentage of conflicts rises, all parallel techniques approximate the sequential performance (Figures 2(h) and 2(i)).

5.7.2. Stealing throughput

An important aspect of our work-stealing approach is that stealing can only happen when the workload meets certain conditions: there must be a victim thread whose queue contains only requests that do not conflict with each other and with requests in the stealer's queue. In this section, we analyze how much stealing is possible in different workload configurations. We call *stealing throughput* the throughput of stolen requests only, within the overall throughput. Figure 7 presents the results, showing that the amount of stolen work increases with the number of shards and the rate of writes in the workload. A high number of writes, however, cuts both ways: although increasing the percentage of writes improves the possibilities of stealing (recall that threads steal requests when waiting for synchronization of conflicting requests), at the same time it also decreases the amount of requests that can be stolen at a time, due to the conflict rate. The good performance of our work-stealing approach is due to both the fact that it reduces load unbalance and does not suffer from barrier synchronization costs.

5.7.3. External sensitivity

The last aspect that we evaluate regards the sensitivity of our work-stealing technique to external events that can possibly interfere with the stealing capacity. In particular, we consider the impact caused by the garbage collection process in the Java VM. Figure 8 presents the results for stealing throughput when running a configuration with light requests, 8 shards, 32 threads, and 5% of writes. We consider four different garbage-collection algorithms: Concurrent Mark Sweep (CMS), the G1 Young Generation (G1), Mark Sweep Compact (MSC), and PS MarkSweep/Scavenge (PSMS/S). As we can see, there is no relevant interference in the amount of stealing, except when running with the MSC algorithm, which imposes a slightly lower steal (and overall) throughput rate. In all other experiments in the paper we used G1.

5.8. Key-value store application

We also implemented a key-value store application using the Java TreeMap data structure, to further extend our performance analysis. In our implementation, the keys are integers, and the values are strings. Since the map provides much faster data access than the list, the throughput of the application increased significantly. In this case, we changed the factor for the request costs, initializing the key-value store with more elements (1M). Figure 9 shows the results for the multi-sharded key-value store application. Notice the throughput now reaches higher levels of requests per second, due to faster data access. However, despite the sequential version being much more efficient than in the list, our techniques again surpassed all others in most cases.

6. Related work

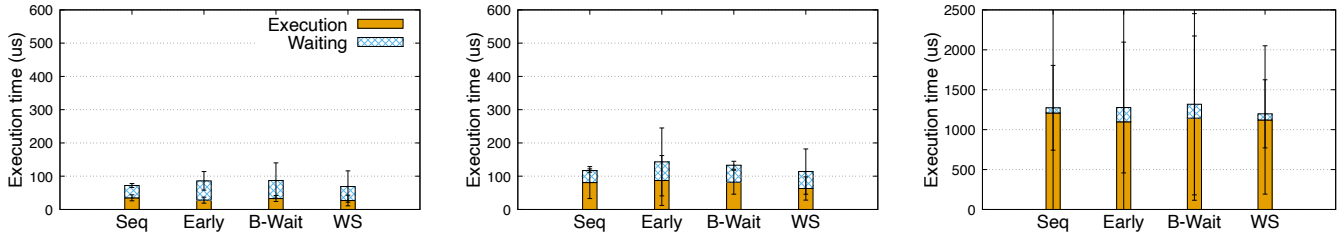
This paper is at the intersection of two areas of research: work-stealing and state machine replication.

6.1. Work-stealing

The first scheduler with a randomized work-stealing algorithm for multi-threaded applications was proposed by Blumofe and Leiserson in [20]. They have shown that the work-stealing technique has lower communication costs than the work-sharing approach. Work-sharing and work-stealing are both techniques that optimize load balance. The difference is: in work-sharing, the scheduler always attempts to assign new threads to different processors. In work-stealing, the idle processes take the initiative to steal work from busy ones. Many variants of the work-stealing algorithm were presented in the literature. For example, a threshold-based queueing model of shared-memory multiprocessor scheduling is presented in [23]. In [24] the authors argued that work-sharing outperforms work-stealing under light to moderate loads, while work-stealing outperforms work-sharing under high loads. In [25], the authors present a work-stealing algorithm that uses locality information (affinity) and outperforms the standard work-stealing approach.

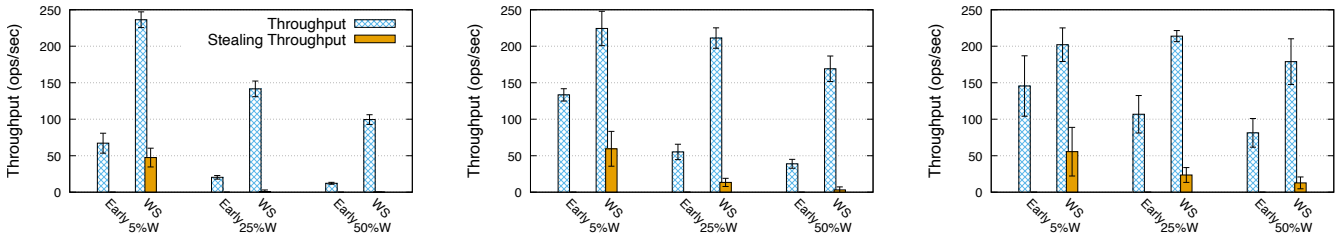
Work-stealing has also been investigated in a variety of other contexts. Among them, applications to thread scheduling, such as list scheduling [26], Fork-Join parallel programming [27], false sharing [28] and parallel batched data structures [29].

Our contribution applies work-stealing concepts in the context of early-scheduling parallel state machine replication. According to early-scheduling, threads are associated with request classes. Differently from previous works, this imposes additional restrictions in the execution model of both stealer and victim threads.



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

Figure 6: Average execution and synchronization/waiting time of a single request for each cost configuration used in the experimental evaluation.



(a) 2 Shards (b) 4 Shards (c) 8 Shards

Figure 7: Average stealing throughput in the Work-Stealing technique in various experiment configurations.

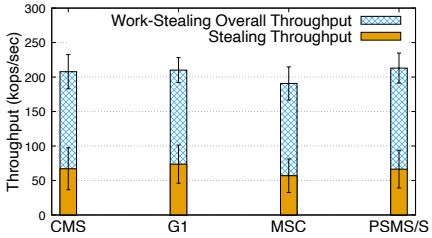
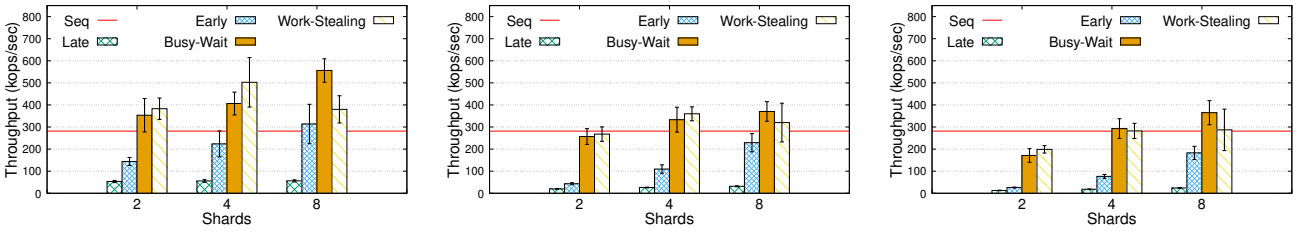


Figure 8: Average stealing throughput considering the impact of different garbage collectors (external sensitivity).



(a) 5% writes (b) 25% writes (c) 50% writes

Figure 9: Results for multi-shard, 1% global requests, balanced workload in a key-value store application.

6.2. State machine replication

In this section, we review proposals that introduce concurrency in the execution of requests in SMR. In CBASE [11], replicas are augmented with a deterministic scheduler. Based on application semantics, it serializes the execution of conflicting requests respecting delivery order and dispatches requests that do not conflict to parallel execution. Conflict detection is done by a dependency graph, which organizes requests to be processed as soon as possible (whenever dependencies have been executed). CBASE is an example of late scheduling.

In [30], the same approach is used: a scheduler evaluates new transactions delivered at a replica against pending trans-

actions and builds a DAG with dependencies that have to be respected during execution. For fast detection of dependencies, an index structure indexes all items accessed by pending transactions. When a new transaction arrives, the ones it depends on are detected evaluating each item accessed by the new transaction against the index structure, avoiding a graph traversal.

In [13] the authors avoid a central scheduler by statically mapping requests to different multicast groups. Non-conflicting requests are multicast to different groups that partially order requests across replicas. Conflicting requests are multicast to the same group. At a replica, each thread is associated with a group and processes requests as they arrive. Requests delivered by different groups are executed concurrently.

Rex [6] and CRANE [31] introduce consensus on replica synchronization events to solve non-determinism due to concurrency. In Rex, a primary replica logs dependencies among requests during execution. The trace is proposed for agreement with other replicas. After agreement, replicas replay the execution according to the trace. CRANE [31] solves non-determinism with the input determinism of Paxos and the execution determinism of deterministic multi-threading [32]. It implements additional consensus on synchronization events such that replicas see the same sequence of primitives.

Eve [8] and Storyboard [7] use optimistic approaches that may lead to additional overhead in some cases. In Eve, replicas compare the results of optimistic execution using consensus and if they diverge, roll-back and conservatively re-execute requests. With Storyboard, replicas have (a priori) forecasts of sequences of locks needed by requests. When the execution deviates from the expected, a deterministic re-execution is necessary.

Calvin [33] is a distributed transactional storage system. To ensure determinism across replicas, transactions are totally ordered. To concurrently execute transactions at nodes, a lock manager is used. It ensures that locks on objects are acquired following the total order of transactions. Transactions have to lock all needed objects before starting the execution. If two transactions conflict, then the total order is imposed due to locks; otherwise they can be concurrently executed. In [34], a Very Light Lock manager is proposed to reduce locking overhead. It keeps the same assumptions and requirements as Calvin.

In [35], the authors present a BFT SMR that can tolerate all but one replicas active in normal-case operation to fail. It runs a extended consensus protocol that exploits passive replication to save resources. In the absence of faults, it requires only $f+1$ replicas to agree on client requests. When suspecting faulty behavior, the system triggers a transition protocol that activates f extra replicas to bring the system into a consistent state again.

Another approach to improving the performance of SMR is to weaken the total order requirement of requests at the replicas. In particular, only conflicting requests must be ordered consistently at the replicas. Generalized Paxos [36], Generic Broadcast [37], Egalitarian Paxos [38], Mencius [39], and Alvin's Partial Order Broadcast (POB) [40] are examples of protocols that adopt this approach. The techniques proposed by such protocols are orthogonal to the aspects discussed in this paper. The exploration of execution parallelism out of request dependencies from generalized consensus is reported in Alvin with the proposed *parallel concurrency control layer* [40] and in [41] by replacing the sequential execution phase of EPaxos with a parallel one. Investigating the complementarity of these approaches and the mechanisms proposed here comprises an interesting direction for future research.

7. Conclusions

This paper reports on our efforts to increase the performance of P-SMR through enhancements to a scheduling approach that simplifies the work done by the scheduler, called early scheduling. We studied scenarios where this technique could lead to poor resource utilization, and investigated how to apply well-established concurrent techniques (e.g., busy waiting, work-stealing) in order to improve early scheduling.

Many works have proposed different approaches to P-SMR. However, we could not find any previous research using work-stealing concepts. In this work, we introduced this alternative, shedding some light in this not-yet-addressed aspect in the literature. Experimental results have shown relevant performance gains of up to four times in some cases, opening up new opportunities for further improving the performance of the early scheduling technique.

Acknowledgements

We wish to thank the anonymous reviewers for their constructive comments and suggestions. This work was achieved in cooperation with HP Brasil Indústria e Comércio de Equipamentos Eletrônicos LTDA. using incentives of Brazilian Informatics Law (Law number 8.2.48 of 1991), and also with CAPES (Brazil), FAPERGS (RS-Brazil), and the Swiss National Science Foundation (SNSF).

References

[1] L. Lamport, Time, clocks, and the ordering of events in a distributed system, Communications of the ACM 21 (7) (1978) 558–565.

[2] F. B. Schneider, Implementing fault-tolerant service using the state machine aproach: A tutorial, ACM Computing Surveys 22 (4) (1990) 299–319.

[3] M. Burrows, The chubby lock service for loosely coupled distributed systems, in: Proceedings of the 7th Symposium on Operating Systems Design and Implementation, 2006.

[4] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, T. Anderson, Scalable consistency in scatter, in: Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, 2011.

[5] J. D. J. C. Corbett, M. E. et al, Spanner: Google's globally distributed database, in: Prooceedings of the 10th Symposium on Operating Systems Design and Implementation, 2012.

[6] Z. Guo, C. Hong, M. Yang, L. Zhou, L. Zhuang, D. Zhou, Rex: Replication at the speed of multi-core, in: European Conference on Computer Systems, 2014.

[7] R. Kapitza, M. Schunter, C. Cachin, K. Stengel, T. Distler, Storyboard: Optimistic deterministic multithreading., in: Workshop on Hot Topics in System Dependability, 2010.

[8] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, M. Dahlin, All about eve: execute-verify replication for multi-core servers, in: OSDI, 2012.

[9] O. Mendizabal, R. Moura, F. Dotti, F. Pedone, Efficient and deterministic scheduling for parallel state machine replication, in: International Parallel & Distributed Processing Symposium, 2017.

[10] B. Li, W. Xu, R. Kapitza, Dynamic state partitioning in parallelized byzantine fault tolerance, in: 2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W), 2018, pp. 158–163. doi:10.1109/DSN-W.2018.00056.

[11] R. Kotla, M. Dahlin, High throughput byzantine fault tolerance, in: IEEE/IFIP Int. Conference on Dependable Systems and Networks, 2004.

[12] E. Alchieri, F. Dotti, F. Pedone, Early scheduling in parallel state machine replication, in: ACM SoCC, 2018.

[13] P. J. Marandi, C. E. Bezerra, F. Pedone, Rethinking state-machine replication for parallelism, in: IEEE International Conference on Distributed Computing Systems, 2014.

[14] E. Batista, E. Alchieri, F. Dotti, F. Pedone, Resource utilization analysis of early scheduling in parallel state machine replication, in: 9th Latin-American Symposium on Dependable Computing (LADC), 2019.

[15] X. Défago, A. Schiper, P. Urbán, Total order broadcast and multicast algorithms: Taxonomy and survey, ACM Computing Surveys 36 (4) (2004) 372–421.

[16] V. Hadzilacos, S. Toueg, Fault-tolerant broadcasts and related problems, in: S. Mullender (Ed.), Distributed Systems, ACM Press/Addison-Wesley Publishing Co., 1993, pp. 97–145.

[17] T. D. Chandra, S. Toueg, Unreliable failure detectors for reliable distributed systems, Journal of ACM 43 (2) (1996) 225–267.

[18] M. J. Fischer, N. A. Lynch, M. S. Paterson, Impossibility of distributed consensus with one faulty process, Journal of the ACM 32 (2) (1985) 374–382.

[19] M. Herlihy, J. M. Wing, Linearizability: A correctness condition for concurrent objects, ACM Transactions on Programing Languages and Systems 12 (3) (1990) 463–492.

[20] R. D. Blumofe, C. E. Leiserson, Scheduling multithreaded computations by work stealing, J. ACM 46 (5) (1999) 720–748.

[21] M. Herlihy, N. Shavit, The Art of Multiprocessor Programming, Revised Reprint, 1st Edition, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2012.

[22] A. Bessani, J. Sousa, E. Alchieri, State machine replication for the masses with BFT-SMaRt, in: IEEE/IFIP International Conference on Dependable Systems and Networks, 2014.

[23] M. S. Squillante, R. D. Nelson, Analysis of task migration in shared-memory multiprocessor scheduling, in: Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, 1991, pp. 143–155.

[24] D. L. Eager, E. D. Lazowska, J. Zahorjan, A comparison of receiver-initiated and sender-initiated adaptive load sharing (extended abstract), in: Proceedings of the 1985 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, 1985, pp. 1–3.

[25] U. A. Acar, G. E. Blelloch, R. D. Blumofe, The data locality of work stealing, in: Proceedings of the Twelfth Annual ACM Symposium on Parallel Algorithms and Architectures, 2000, pp. 1–12.

[26] M. Tchiboukdjian, N. Gast, D. Trystram, Decentralized list scheduling, CoRR abs/1107.3734 (2011). arXiv:1107.3734.
URL http://arxiv.org/abs/1107.3734

[27] K. Agrawal, C. E. Leiserson, J. Sukha, Helper locks for fork-join parallel programming, in: Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2010, pp. 245–256.

[28] R. Cole, V. Ramachandran, Analysis of randomized work stealing with false sharing, in: 2013 IEEE 27th International Symposium on Parallel and Distributed Processing, 2013, pp. 985–998. doi:10.1109/IPDPS.2013.86.

[29] K. Agrawal, J. T. Fineman, K. Lu, B. Sheridan, J. Sukha, R. Utterback, Provably good scheduling for parallel programs that use data structures through implicit batching, in: Proceedings of the 26th ACM Symposium on Parallelism in Algorithms and Architectures, 2014, pp. 84–95.

[30] G. Zhao, G. Wu, Y. Song, B. Qiao, D. Han, Index-based scheduling for parallel state machine replication, in: X. Wang, R. Zhang, Y.-K. Lee, L. Sun, Y.-S. Moon (Eds.), Web and Big Data, Springer International Publishing, Cham, 2020, pp. 808–823.

[31] H. Cui, R. Gu, C. Liu, T. Chen, J. Yang, Paxos made transparent, in: ACM Symposium on Operating Systems Principles, ACM, 2015, pp. 105–120.

[32] M. Olszewski, J. Ansel, S. Amarasinghe, Kendo: efficient deterministic multithreading in software, ACM Sigplan Notices 44 (3) (2009) 97–108.

[33] A. Thomson, T. Diamond, S. Weng, K. Ren, P. Shao, D. J. Abadi, Calvin: fast distributed transactions for partitioned database systems, in: K. S. Candan, Y. Chen, R. T. Snodgrass, L. Gravano, A. Fuxman (Eds.), Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2012, Scottsdale, AZ, USA, May 20-24, 2012, ACM, 2012, pp. 1–12. doi:10.1145/2213836.2213838.
URL https://doi.org/10.1145/2213836.2213838

[34] K. Ren, A. Thomson, D. J. Abadi, VLL: a lock manager redesign for main memory database systems, VLDB J. 24 (5) (2015) 681–705. doi:10.1007/s00778-014-0377-7.
URL https://doi.org/10.1007/s00778-014-0377-7

[35] R. Kapitza, J. Behl, C. Cachin, T. Distler, S. Kuhnle, S. V. Mohammadi, W. Schröder-Preikschat, K. Stengel, Cheapbft: Resource-efficient byzantine fault tolerance, in: Proceedings of the 7th ACM European Conference on Computer Systems, EuroSys '12, Association for Computing Machinery, New York, NY, USA, 2012, p. 295–308. doi:10.1145/2168836.2168866.
URL https://doi.org/10.1145/2168836.2168866

[36] L. Lamport, Generalized consensus and paxos, Technical Report MSR-TR-2005-33, Microsoft Research (2005).

[37] F. Pedone, A. Schiper, Handling Message Semantics with Generic Broadcast Protocols, Distrib. Comput. 15 (2) (2002).

[38] I. Moraru, D. G. Andersen, M. Kaminsky, There is More Consensus in Egalitarian Parliaments, in: SOSP, ACM, 2013.

[39] Y. Mao, F. P. Junqueira, K. Marzullo, Mencius: Building efficient replicated state machines for wans, in: 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08), USENIX Association, San Diego, CA, 2008.

[40] A. Turcu, S. Peluso, R. Palmieri, B. Ravindran, Be general and don't give up consistency in geo-replicated transactional systems, in: M. K. Aguilera, L. Querzoni, M. Shapiro (Eds.), Principles of Distributed Systems - 18th International Conference, OPODIS 2014, Cortina d'Ampezzo, Italy, December 16-19, 2014. Proceedings, Vol. 8878 of Lecture Notes in Computer Science, Springer, 2014, pp. 33–48. doi:10.1007/978-3-319-14472-6_3.
URL https://doi.org/10.1007/978-3-319-14472-6_3

[41] T. C. Junior, F. L. Dotti, F. Pedone, Parallel state machine replication from generalized consensus, in: International Symposium on Reliable Distributed Systems, SRDS 2020, Shanghai, China, September 21-24, 2020, IEEE, 2020, pp. 133–142. doi:10.1109/SRDS51746.2020.00021.
URL https://doi.org/10.1109/SRDS51746.2020.00021