Resource Utilization Analysis of Early Scheduling in Parallel State Machine Replication

Eliã Batista[1], Eduardo Alchieri[2], Fernando Dotti[1] and Fernando Pedone[3]
[1]*Escola Politécnica, Pontifícia Universidade Católica do Rio Grande do Sul, Brazil*
[2]*Departamento de Ciência da Computação, Universidade de Brasília, Brazil*
[3]*Universitá della Svizzera Italiana (USI), Switzerland*

***Abstract*—Most approaches to scaling the throughput of state machine replication resort to concurrent execution of non-conflicting requests. While these approaches enhance throughput, conflict detection and handling introduce overhead. Early scheduling is a promising technique that trades concurrency for expeditious scheduling decisions. In this technique, requests are grouped in classes and a fixed subset of threads is assigned to each class, respecting request dependencies. Early scheduling has been shown to provide significant performance improvements in state machine replication.**

This paper evaluates the impact of the restrictions imposed by the early scheduling technique. In particular, it shows that threads may be idle while pending independent requests are available to be executed, leading to poor processor utilization. We characterize resource underutilization for workloads with different rates of conflicting requests, number of threads, and number of request classes. The paper opens up new opportunities to further enhance the early scheduling technique.

***Index Terms*—State Machine Replication, Concurrent Execution, Scheduling**

I. Introduction

State machine replication (SMR) is a well-established approach to fault tolerance [20], [28]. The approach offers strong consistency and allows application programmers to focus on the inherent complexity of the application, while remaining oblivious to the difficulty of handling replica failures [11]. This is one of the reasons SMR has been successfully used in many contexts and applications (e.g., [6], [12], [16]).

Replica consistency in SMR is based on deterministic execution of requests in the same order. This simple replication model is challenged by modern multi-core servers since deterministic execution often translates into single-threaded replicas. Based on the observation that *independent* requests can be executed concurrently while *conflicting* (or *dependent*) requests[1] must be serialized to keep replicas consistent [28], and the fact that many workloads are dominated by independent requests (e.g., [5], [9], [19], [21], [23], [24]), a number of works have proposed to explore intra-replica concurrency for request execution. In this context, some proposals follow more closely the SMR approach (e.g., [2], [3], [19], [23], [25]) than others (e.g., [8], [13], [17], [18], [26]).

An important aspect in the design of multi-threaded SMR is how to schedule requests for execution on worker threads. Scheduling has been an active area of research for several decades, with algorithms tailored to different situations. The SMR scheduling problem can be classified as online, without processing times information, non-realtime, and with possibly interdependent jobs [22]. There is no discussion in the scheduling literature on the costs or techniques to detect and represent dependencies among jobs or requests (i.e., the job dependency graph is typically given). Also, there is a lack of discussion on synchronization costs to enforce job dependencies upon execution. Differently from scheduling problems in other application areas, these aspects matter when scheduling requests in online services. In modern online services, due to high throughput and potential concurrency, the overhead to manage dependencies gains in importance and may become a system bottleneck. This is an important aspect of concern when discussing approaches for parallel execution in SMR.

Different approaches to Parallel State Machine Replication (P-SMR) show different tradeoffs between the concurrency level allowed in the system and the overhead for conflict detection and handling [1]: to allow maximum request execution concurrency, a costly conflict detection is performed at scheduling time while this overhead may be considerably reduced if we renounce some concurrency during execution. Based on these observations, scheduling techniques can be classified in three categories [1]: (i) In *late scheduling* [19], all scheduling decisions are made at the replicas after the requests are ordered, providing maximum execution concurrency but introducing high overhead for conflict detection at the replicas. (ii) In *static scheduling* [23], scheduling decisions are made before requests are ordered for execution, providing low execution concurrency with low overhead since there is no request scheduling at the replicas. (iii) In *early scheduling* [3], part of the scheduling decisions are made before requests are ordered. These restrictions must be respected at the replicas, allowing an intermediary level of concurrency for request executions and also imposing an intermediary overhead for conflict detection and handling at the replicas.

Among these P-SMR approaches, early scheduling has been shown to outperform the other approaches in many scenarios [3]. The main reason is that by restricting concurrency, scheduling can be done more efficiently at the replicas. The idea is to group service requests in classes and then specify how classes must be synchronized. Then, a fixed subset of worker threads is assigned to each class. As a result, requests

[1]Two requests conflict if they access common state and at least one of them updates the state, otherwise they are independent.

978-1-7281-6622-3/19/$31.00 ©2019 IEEE

are scheduled to one or more worker threads, and may execute concurrently or serially, depending on the classes of the requests and how they relate. For example, two requests that belong to a class that admits concurrency can execute in parallel; but two requests that belong to conflicting classes will be executed sequentially, after the involved worker threads synchronize. The mapping of request classes to working threads was modeled as an optimization problem where the goal is to maximize execution concurrency [3], according to a pre-defined workload.

Although early scheduling shows performance gains, it restricts concurrency according to the class definitions and class-to-thread mapping. To understand why, consider a sharded application, where each shard is associated with read and write request classes. Threads are mapped to classes according to the class definition and expected shard load. In general, if a shard has expected load higher than the others, more threads will be mapped to implement that shard. However, with workloads transiently deviating from the expected values, early scheduling does not distribute workload evenly across threads. Consequently, some threads may be overloaded while others are idle.

In this paper, we revisit the early scheduling approach. By conducting a series of experiments with many different workloads, we identify scenarios in which idle and overloaded threads coexist during system execution. We quantify load unbalance among threads considering the amount of requests each thread has to execute and the time demanded by synchronization. The study opens up new opportunities to further enhance the early scheduling technique.

The paper continues as follows. Section II introduces the system model and some preliminary definitions. Section III surveys related work. Section IV presents the early scheduling technique. Section V evaluates early scheduling looking at possible performance enhancements. Section VI concludes the paper.

II. Background

A. System Model

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. The system is asynchronous: there is no bound on message delays and on relative process speeds. We assume the crash 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 have access to an atomic broadcast communication abstraction, defined by primitives $broadcast(m)$ and $deliver(m)$, where m is a message. Atomic broadcast ensures the following properties [10], [14][2]:

- *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'.

[2]Atomic broadcast needs additional synchrony assumptions to be implemented [7], [11].

B. Consistency

Our consistency criterion is *linearizability*. A linearizable execution satisfies the following requirements [15]:

- There exists a total order among any two operations that respects the real-time ordering of operations across all clients. One operation precedes another in real time if the first operation finishes at a client before the second operation starts at a client.
- It respects the semantics of the operations as defined in their sequential execution.

C. Command Independence

To keep strong consistency, instead of sequentially executing commands, it has been observed that it suffices for a replica to execute sequentially only commands that access the same variables and one of the commands modifies the shared variables (conflicting or dependent commands). The other commands (independent commands) can be executed concurrently without violating consistency [28]. The notion of command interdependency is application-specific. Recently, several replication models have exploited command dependencies to parallelize the execution on replicas.

More formally, command or request[3] conflict 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

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

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 (i.e., they do not share any objects or only read shared objects), then the requests can be executed concurrently at replicas (e.g., by different worker threads at each replica).

[3]We use command and request with the same meaning.

III. Related Work

In [1], a classification of approaches to Parallel SMR is introduced with the following classes:

- *Pipelined SMR* is a technique whereby replicas implement staging to enhance throughput. Replicas are organized in a series of modules connected through shared totally ordered message queues (e.g., [27]). Although staging improves the system throughput, there is always only one thread sequentially executing the commands.
- *Late Scheduling* proposes that commands delivered at replicas be evaluated for conflict and scheduled concurrently for execution whenever they are independent from the ones under execution or pending. In [19], a parallel SMR is proposed where replicas are augmented with a deterministic scheduler. Based on application semantics, the scheduler serializes the execution of conflicting requests according to the delivery order and dispatches non-conflicting requests to be processed in parallel by a pool of worker threads. Since the scheduling decision is taken at the replica, before execution, the scheme has been dubbed late scheduling.
- *Early Scheduling* emerges from the observation that the overhead needed to keep command dependency information in late scheduling can be significant. Early scheduling (e.g., [2], [3]) trades concurrency for expeditious decisions at replicas. With application semantics, requests are grouped in classes and subsets of threads are assigned to implement classes. Threads to classes assignment is performed a priori. Since classes can conflict, requests from conflicting classes are serialized by involving threads from those different classes that synchronize to execute conflicting commands implemented by a thread level execution model. With this scheme, when commands arrive, the scheduler simply schedules them according to the mode (sequential or concurrent) to the set of pre-assigned threads. The complexity of dependency detection and according scheduling is thus bounded. The next section will dive into more details.
- *Static Scheduling* is a more strict idea of Early Scheduling. It completely eliminates scheduling decisions at replicas. P-SMR [23] adopts this approach. Clients map requests to different multicast groups based on request information which is application specific. Non-conflicting requests can be sent to distinct groups, while conflicting ones are sent to the same group(s). At the replica side, each worker thread is associated to a multicast group and processes requests as they arrive. When a request arrives through more than one group, associated threads synchronize to execute, imposing an order on all involved threads (multicast groups).

Although the scheduling classification above encompasses several existing proposals to P-SMR, there are approaches to concurrent request execution in SMR-like architectures that do not fall into any of the identified categories. We call here SMR-like those architectures that depart from the principle of request independency and introduce additional cooperation among replicas, beyond the basic assumption of request ordering.

Rex [13] and CRANE [8] add complexity to the execution phase by introducing consensus about replicas synchronization events to solve non-determinism due to concurrency. Rex uses an execute-agree-follow strategy. A primary replica logs dependencies among requests during execution, based on shared variables locked by each request. This creates a trace of dependencies which is proposed for agreement with other follower replicas. After agreement replicas replay the execution restricted to the trace of the first executing server. CRANE [8] solves non-determinism with the input determinism of Paxos and the execution determinism of deterministic multithreading [26]. CRANE implements an additional underlying consensus on synchronization events such that replicas see the same sequence of calls to synchronization primitives.

Eve [18] and Storyboard [17] use optimistic approaches that may lead to additional overhead in case replicas do not agree on the result. In Eve, this is done with optimistic execution and comparing results (consensus). If replicas diverge, roll-back and conservative re-execution is performed. With Storyboard, replicas have (a priori) forecasts of sequences of locks needed by requests. When execution deviates from expected, replicas have to establish a deterministic execution.

IV. Early Scheduling

Several approaches to parallel SMR resort to application semantics to parallelize independent commands. While this allows concurrency, it introduces scheduling overhead to decide which commands are independent and which thread should execute each command. The Early Scheduling approach [2], [3] proposes a way to classify requests in *request classes* and a fast scheduling algorithm based on classes.

A. Request Classes

The notion of request classes was introduced in [2] to denote application knowledge. Consider a service with a set R of possible requests. 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 $\mathcal{R} = C \rightarrow \mathcal{P}(C) \times \mathcal{P}(R)$,[4] that is, any class in C may conflict with any subset of classes in C, and is associated to a subset of requests in R. A conflict among classes happens when any two requests from those classes conflict, according to the conflict definition $\#_R$ above. Moreover, we introduce the restriction that a non-empty non-overlapping subset of requests from R is associated to each class.

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

a) 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. Read operations in a shard do not conflict. Writes conflict with reads and writes. Global writes conflict with any global or local operation. Global reads do not conflict with reads, global or local.

We model this application with the following classes. Read class C_{R1} in partition 1 conflicts with the write class C_{W1} on the same partition and with the global write class C_{Wg}. The read class C_{R2} in partition 2 conflicts with the write class C_{W2} on the same partition and with the global write class C_{Wg}. The class C_{Wg} also conflicts with itself, with the write classes and with the overall reading class C_{Rg}. Writing classes C_{W1} and C_{W2} also conflict with themselves and with the overall reading class C_{Rg}. Class C_{Rg} also conflicts with itself. This is denoted in Figure 1, where classes are nodes and conflicts are edges.

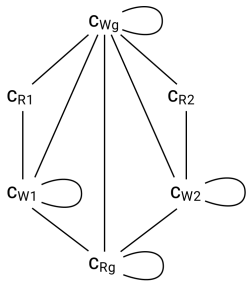


Fig. 1: Request class definition with two shards.

B. Classes, Threads and Execution

Central to the idea of Early Scheduling is that the scheduling algorithm avoids the Late Scheduling overhead, i.e., it does not have to evaluate every other pending command to decide how to schedule a new incoming one. It suffices to know the request's class to associate an appropriate worker thread. With this, the scheduling overhead is bounded, independently of the population of pending requests.

1) Execution Model: To accomplish such a straightforward scheduling algorithm, the Early Scheduling adopts a replica execution model that will synchronize requests from conflicting classes. A replica will have one scheduler thread and n worker threads. Each worker thread has a separate input FIFO queue. The scheduler receives each request r totally ordered from consensus and decides to which worker thread(s) to associate.

a. If scheduled to one worker only, r can be processed concurrently with other requests.
b. If scheduled to more than one worker thread, then r depends on preceding requests assigned to these workers. Therefore, all workers involved in r must synchronize before one worker among these executes r.

2) Class to threads mapping: With this execution model, the following class-to-thread-mapping rules can be applied to ensure linearizable executions:

i. *Every class is associated with at least one worker 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 workers associated to c_2 must be included in the set of workers associated to 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 worker in common.* The common worker ensures that requests in the classes are serialized.

These rules result in several possible class-to-threads mappings. 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 $T = \{t_0, .., t_{n-1}\}$, the n worker threads at a replica.

a) Example: Following our example from Figure 1, considering 4 worker threads available, a possible mapping following the rules above is depicted in Table I.

TABLE I: 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		$\{t_0,$	$t_1,$	$t_2,$	$t_3\}$	

C. Algorithms

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

Algorithm 1 Early scheduler.

```
1: variables:
2:     queues[0, ..., n − 1] ← ∅                    // one queue per worker thread
3: on deliver(req):
4:     if req.class.smode = Seq then                 // if execution is sequential
5:         ∀t ∈ CtoT(req.classId)                    // for each conflicting thread
6:             queues[t].fifoPut(req)                // synchronize to exec req
7:     else                                          // else assign req to one thread in round-robin
8:         queues[next(CtoT(req.classId))].fifoPut(req)
```

Algorithm 2 Worker threads for early scheduling.

```
1: variables:
2:     myId ← id ∈ {0, ..., n − 1}            // thread id, out of n threads
3:     queue[myId] ← ∅          // the queue with requests for this thread
4:     barrier[C]                                  // one barrier per request class
5: while true do
6:     req ← queue.fifoGet()              // wait until a request is available
7:     if req.class.smode = Seq then                  // sequential execution:
8:         if myId = min(CtoT(req.classId)) then          // smallest id:
9:             barrier[req.classId].await()                // wait for signal
10:            exec(req)                                   // execute request
11:            barrier[req.classId].await()                // resume workers
12:        else
13:            barrier[req.classId].await()                // signal worker
14:            barrier[req.classId].await()                // wait execution
15:    else                                          // concurrent execution:
16:        exec(req)                                      // execute the request
```

Each worker thread (Algorithm 2) takes one request at a time from its queue in FIFO order (line 6) and then proceeds depending on the synchronization mode of the class. If the class is sequential, then the thread synchronizes with the other threads in the class using barriers before the request is executed (lines 8–14). In the case of a sequential class, only one thread executes the request. If the class is concurrent, then the thread simply executes the request (lines 15–16).

Safety and liveness are argued in [3], where it is shown that these algorithms generate linearizable executions and that every request is eventually executed.

V. Early Scheduling Analysis

Early scheduling restricts concurrency to allow fast scheduling decisions. This section analyzes early scheduling in different scenarios to understand how these restrictions affect thread utilization and load balancing.

A. Environment

Experiments were conducted in seven nodes connected by a local-area network (cluster). Three server nodes implement BFT-SMaRt replicas, one per node. BFT-SMaRt [4] is a well-established framework to develop SMR. Each server node has the following configuration: AMD Opteron® Processor 6366 HE @ 2271.490Mhz, 64 cores; 125GB RAM; operating system Linux Ubuntu 4.15.0 (buildd@lgw01-amd64-001); gcc version 7.3.0 (Ubuntu 7.3.0-16ubuntu3), 64 bits; Java Virtual Machine and OpenJDK version 11.0.3; OpenJDK 64-Bit Server VM.

Four client nodes were configured to run client processes. Each client node has the following configuration: Intel® Xeon® L5420 @ 2.50GHz processor with 8 cores; 8GB RAM; operating system Linux Ubuntu 4.15.0, (buildd@lgw01-amd64-014) (gcc version 7.4.0 (Ubuntu 7.4.0-1ubuntu1 18.04.1)), 64 bits; Java Virtual Machine and OpenJDK version 11.0.3; OpenJDK 64-Bit Server VM.

B. Application

The experiments were performed using a linked list application. The application was implemented to support separate data shards, that is, each replica has an internal partitioned state. There are commands to read from the list and write in the list, accessing a single shard or all shards. A read operation checks whether an element is in one shard or in all shards, and a write operation includes an element in one shard or in all shards. Duplicated elements are not included in some shard, i.e., the write operation checks if some element already is in some shard before inclusion.

We conducted experiments with 2, 4 and 8 shards, in a system with 6, 10 and 18 request classes, respectively. In a deployment with n shards, there are n local (i.e., single-shard) reads classes, n local writes classes, one global (i.e., all shards) read class, and one global write class. Each replica was configured to run t worker threads, where each shard is assigned two threads (the read and write classes of each shard are mapped to the same two threads), and consequently, $t = 2n$. Figure 1 and Table I present the case for 2 shards.

C. Metrics

We consider three distinct metrics:

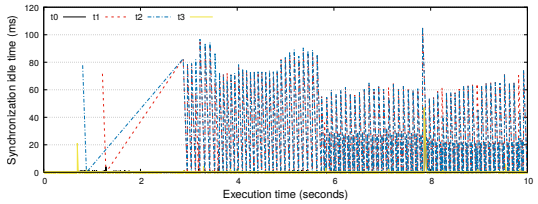
1) *Synchronization idleness.* This metric represents the average wait time for a thread to synchronize with all other threads in the same class before executing a synchronizing command. It is obtained as follows:
   - i High precision system nano-time is collected right before the first barrier *await* instruction, at lines 9 and 13 in Algorithm 2.
   - ii In the thread responsible for executing the request, a second system time is collected right after the first barrier *await* instruction, before the *exec* instruction of line 10.
   - iii In waiting threads, the second measure of system time is collected after the second barrier *await* instruction of line 14.
   - iv The amount of waiting time (difference between the two instants of time collected as described above) is stored per second for each thread.
2) *Queue idleness.* This metric represents the average time that a thread waits for new commands in its queue. It is obtained as follows:
   - i High precision system nano-time is collected right before the *fifoGet* instruction, at line 6 in Algorithm 2, which blocks the thread until a new command is available.
   - ii The second system time is taken right after line 6.
   - iii The waiting time (difference between the two instants of time collected as described above) is stored per second for each thread.
3) *Queue size.* This metric represents the average size of a thread's queue. It is obtained by counting how many commands were returned by the *fifoGet* instruction, line 6 in Algorithm 2. In the implementation, this instruction actually returns a batch of commands available at the scheduler, in the same order as scheduled. The amount of commands is stored per second for each thread.

TABLE II: Threads to classes mappings for 4 shards and 8 threads

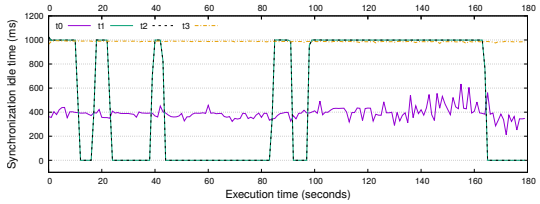
$C =$	$\{seq, conc\}$	$\times \mathcal{P}$ (	$\{t_0,$	$t_1,$	$t_2,$	$t_3,$	$t_4,$	$t_5,$	$t_6,$	$t_7\}$	)
$C_{R1}=$	$conc$		$\{$		$t_2,$		t_4			$\}$	
$C_{R2}=$	$conc$		$\{t_0,$						t_6	$\}$	
$C_{R3}=$	$conc$		$\{$			$t_3,$		t_5		$\}$	
$C_{R4}=$	$conc$		$\{$	$t_1,$						$t_7\}$	
$C_{W1}=$	seq		$\{$		$t_2,$		t_4			$\}$	
$C_{W2}=$	seq		$\{t_0,$						t_6	$\}$	
$C_{W3}=$	seq		$\{$			$t_3,$		t_5		$\}$	
$C_{W4}=$	seq		$\{$	$t_1,$						$t_7\}$	
$C_{Rg}=$	seq		$\{t_0,$				$t_4,$		$t_6,$	$t_7\}$	
$C_{Wg}=$	seq		$\{t_0,$	$t_1,$	$t_2,$	$t_3,$	$t_4,$	$t_5,$	$t_6,$	$t_7\}$	

TABLE III: Threads to classes mappings for 8 shards and 16 threads

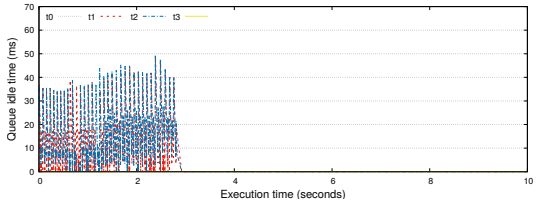
$C =$	$\{seq, conc\} \times \mathcal{P}$	$(\{t_0,$	$t_1,$	$t_2,$	$t_3,$	$t_4,$	$t_5,$	$t_6,$	$t_7,$	$t_8,$	$t_9,$	$t_{10},$	$t_{11},$	$t_{12},$	$t_{13},$	$t_{14},$	$t_{15}\})$
$C_{R1}=$	$conc$	$\{t_0,$	t_1														$\}$
$C_{R2}=$	$conc$	$\{$		$t_2,$	t_3												$\}$
$C_{R3}=$	$conc$	$\{$				$t_4,$	t_5										$\}$
$C_{R4}=$	$conc$	$\{$						$t_6,$	t_7								$\}$
$C_{R5}=$	$conc$	$\{$								$t_8,$	t_9						$\}$
$C_{R6}=$	$conc$	$\{$										$t_{10},$	t_{11}				$\}$
$C_{R7}=$	$conc$	$\{$												$t_{12},$	t_{13}		$\}$
$C_{R8}=$	$conc$	$\{$														$t_{14},$	$t_{15}\}$
$C_{W1}=$	seq	$\{t_0,$	t_1														$\}$
$C_{W2}=$	seq	$\{$		$t_2,$	t_3												$\}$
$C_{W3}=$	seq	$\{$				$t_4,$	t_5										$\}$
$C_{W4}=$	seq	$\{$						$t_6,$	t_7								$\}$
$C_{W5}=$	seq	$\{$								$t_8,$	t_9						$\}$
$C_{W6}=$	seq	$\{$										$t_{10},$	t_{11}				$\}$
$C_{W7}=$	seq	$\{$												$t_{12},$	t_{13}		$\}$
$C_{W8}=$	seq	$\{$														$t_{14},$	$t_{15}\}$
$C_{Rg}=$	seq	$\{t_0,$		$t_2,$		$t_4,$		$t_6,$		$t_8,$		$t_{10},$		$t_{12},$		t_{14}	$\}$
$C_{Wg}=$	seq	$\{t_0,$	$t_1,$	$t_2,$	$t_3,$	$t_4,$	$t_5,$	$t_6,$	$t_7,$	$t_8,$	$t_9,$	$t_{10},$	$t_{11},$	$t_{12},$	$t_{13},$	$t_{14},$	$t_{15}\}$



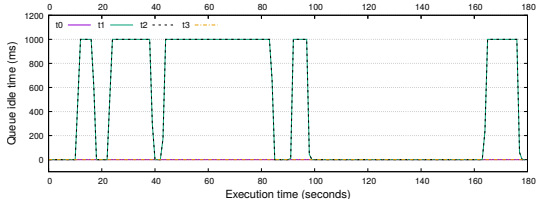
(a) Synchronization idleness



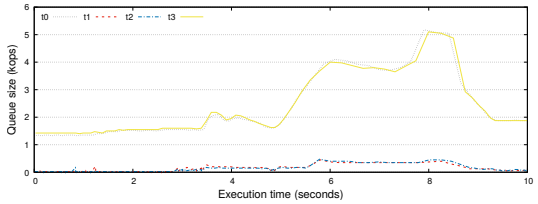
(b) Synchronization idleness



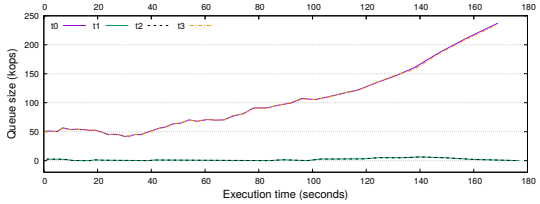
(c) Queue idleness



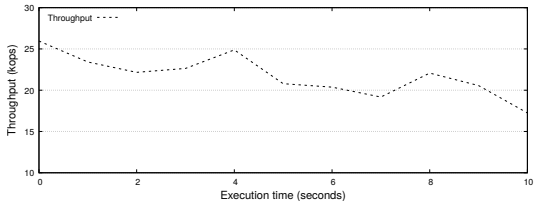
(d) Queue idleness



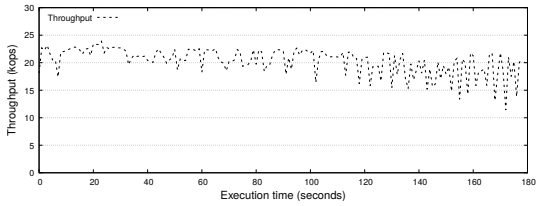
(e) Queue size



(f) Queue size



(g) Throughput



(h) Throughput

Fig. 2: Metrics and throughput in 10 seconds (left) and an entire execution (right), balanced workload 1-1-1, 2 shards (4 threads), and light operations.

D. Workloads

On the client side, we configured each node to run 10, 40 or 50 processes, according with number of shards (2, 4 and 8, respectively), sending requests to servers with mixed workload of read and write commands. Each client process sends batches of 50 commands per request, without interval between each request. This configuration results in performance near its peak.

Several executions were performed submitting the application to different workloads, ranging the percentages of reads, writes, local and global operations. The percentage of writes and global operations ranged from low to high levels due to observation that as the level of conflicts increase, it directly affects the metrics that we are monitoring, especially thread synchronization idleness.

We also consider balanced and unbalanced workloads. In the balanced workload, each shard receives a similar number of local requests. In the skewed workload, each client process sends about 50% of its commands to only one shard (except for the experiments with 2 shards where one shard received 80%), and the remaining commands are equally distributed across the remaining shards. We carried out experiments with different command execution costs, ranging from light, moderate to heavy costs (i.e., lists with 1K, 10K and 100K elements, respectively).

We represent different workloads with notation α-β-γ, where α is the percentage of local writes (i.e., writes in a single shard); β is the percentage of global operations (i.e., operations involving all shards); and γ is the percentage of global writes in the global operations. For example, workload *25-5-25* has 25% of local writes and 5% of global operations, where 25% of the global operations are global writes. Each experiment lasts 4 minutes, where results for the first minute are discarded (system warm-up). In the remaining three minutes, we collect data to compute average and standard deviation values for the metrics described in Section V-C.

E. Results

We start by presenting the results during a single execution with 2 shards, balanced workload *1-1-1* and light operation costs to observe how these metrics evolve during the time. Due to high amount of data collected in a run, and to understand how we thereafter consolidate the data, we first present only 10 seconds of a single execution in Figure 2 (left), where we can see thread behavior with respect to each metric and the system throughput.

In this specific interval, we can observe that in the first 3 seconds there is low rate of synchronization idleness, due to high incidence of concurrent commands. This is also the reason why there is queue idleness, and low amount of commands in the queues. Since there are few commands in the queues, threads are more prone to wait for new commands. It incurs that system throughput is higher than in the next 7 seconds, when more sequential commands arrive, causing the threads to spend more time in the synchronization barriers, increasing the quantity of commands in the queues, decreasing the queue idleness (i.e., threads do not need to wait due to command availability in the queues), and decreasing the system throughput.

Based on these data, each metric was aggregated per second of execution, and we present the results of an entire experiment execution in Figure 2 (right).

Synchronization idleness (Figure 2(b)): Each thread spends different amounts of time waiting for synchronization in the barriers. In this case of balanced workload *1-1-1* with light operation costs, threads $t0$ and $t3$ are the most idle. This behavior reflects the class-to-threads mapping (Table I), where both threads are associated with larger number of classes. Notice that thread $t3$ is the most idle because it never executes synchronized commands. This happens because $t3$ has the highest $thread\_id$ (line 8 of Algorithm 2).

Queue idleness (Figure 2(d)): Queue idleness is inversely proportional to synchronization idleness. Threads $t1$ and $t2$ now are the most idle, due to faster execution of commands. This happens because both threads receive fewer commands, and more often need to get more commands from the scheduler. Thus, they are more prone to find their queue empty, resulting in waiting. Threads $t0$ and $t3$, however, do not need to wait since they spend much time in the barriers, waiting for sequential executions, causing their queues to always have new commands to execute.

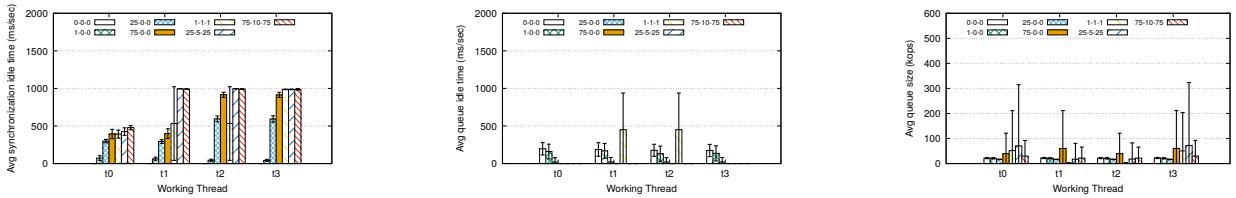
Queue size (Figure 2(f)): We can see how idleness of threads impacts their accumulated work. As threads $t0$ and $t3$ are more often idle waiting for synchronization, the size of their queues keeps increasing during the execution, while queue sizes of threads $t1$ and $t2$ are lower and more constant.

Figure 3 presents the consolidated results for a system with 2 shards, considering different workloads composed of light operations. We vary both percentage of conflicts and request distribution among the shards.

Synchronization idleness (Figure 3(a)): We can observe in this experiment that idleness increases together with conflict percentage. Moreover, according with the classes to threads mappings (Table I), thread $t0$ continued to be less idle than others (lowest $thread\_id$ is always responsible for executing commands) and $t3$ is the most idle in majority of workloads. However, for workloads with high degree of conflicts (*25-5-25* and *75-10-75*), $t0$ executes most of commands while the others remain almost all time only waiting for synchronizations. Notice that for workload *0-0-0* (only reads, which are concurrent commands) there is no synchronization idleness.

Queue idleness (Figure 3(b)): In general, the amount of queue idleness decrease with more conflicting workloads due to increasing synchronization idleness. While the percentage of conflicts in a workload gets higher, all threads spend more time in the synchronization barriers and, consequently, more time are available to them receive new commands in their queues, decreasing the time needed to wait for new commands.

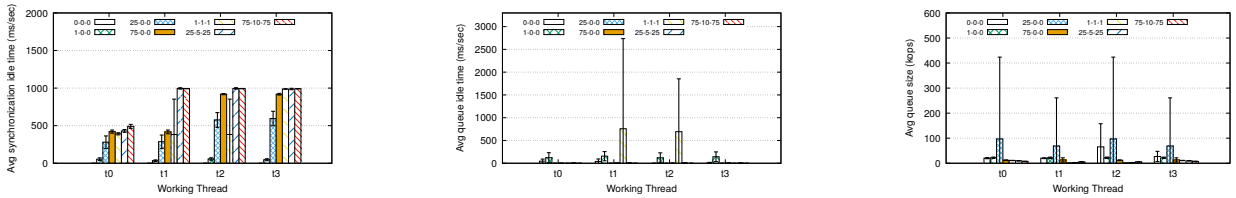
Queue size (Figure 3(c)): This experiment shows that the difference between queue sizes among threads in the same workload increases in some cases, especially in cases with intermediary levels of conflict in the workload. This happens



(a) Average synchronization idleness (light costs)

(b) Average queue idleness (light costs)

(c) Average queue size (light costs)



(d) Average synchronization idleness (light costs)

(e) Average queue idleness (light costs)

(f) Average queue size (light costs)

Fig. 3: Results for 2 shards (4 threads), with balanced workloads (top) and skewed workloads (bottom).

again because of the static classes to threads mappings. Notice the particular case of threads $t1$ and $t2$, which are associated with less amount of request classes and have, in average, less commands in their queues than the other two threads.

Skewed workloads: Figures 3(d), 3(e) and 3(f) present the results for same conflict percentages and shards/threads configurations but for skewed workloads, where most of commands are addressed to shard 1. This experiment shows that average thread idleness continues in high levels for most cases. It is however important to note the increasing on queue sizes variation and differences for most cases, where threads $t0$ and $t2$ accumulate more commands in their queues since they belong to the overloaded partition.

We can also observe high levels of standard deviation in some of the analyzed workloads. This happens because, depending on the demand from clients, in some measurement intervals the threads execute more sequential than concurrent commands, and vice-versa. Moreover, the synchronizations demanded to execute a sequential command is not needed for concurrent ones. This unbalance between sequential and concurrent commands execution leads to high variance.

Impact of the number of shards in the system: Figures 4 and 5 present the results for a system configured with 4 and 8 shards, respectively, considering balanced and skewed workloads where shard 1 received more requests. For better presentation, we exclude workloads *75-0-0* and *75-10-75* since they are the ones with most conflicting requests and always presented the same behavior with high levels of idleness.

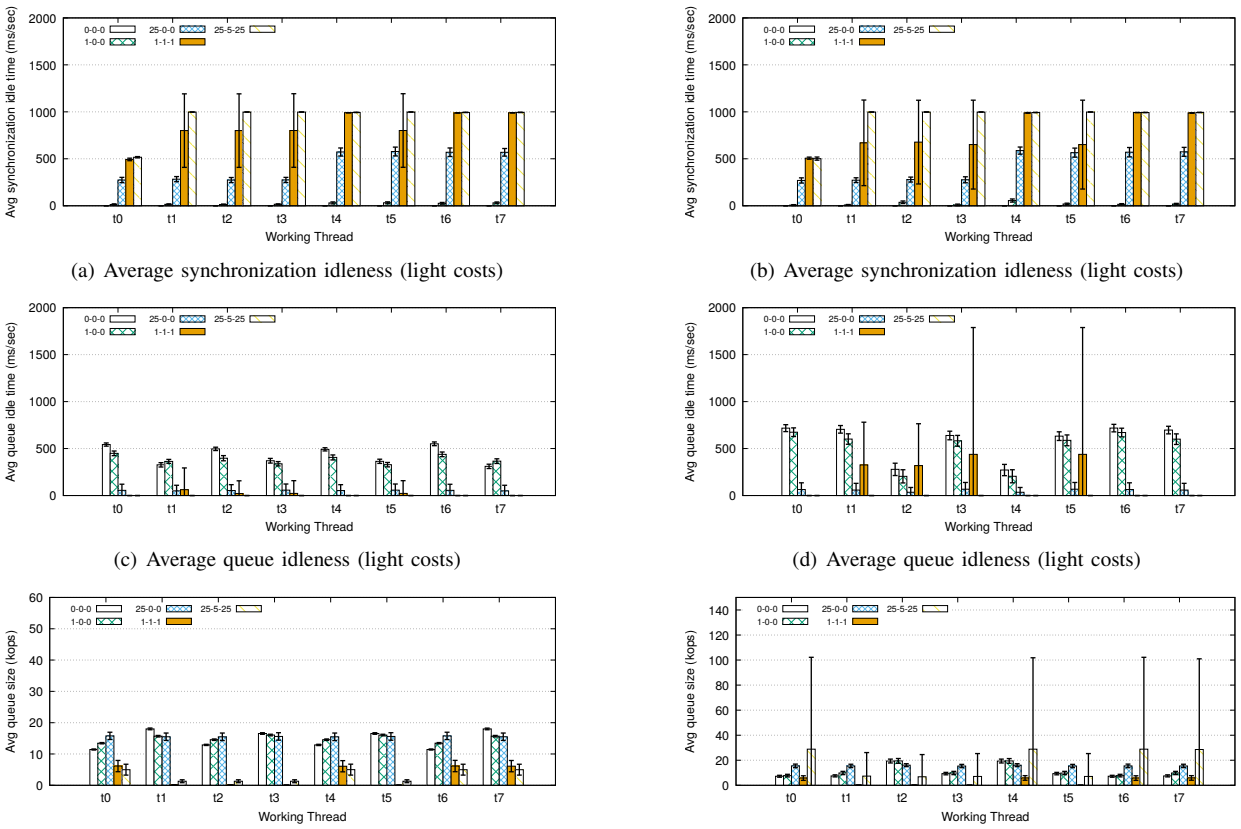
In general, synchronization idleness again presented high levels for most workloads, and queue idleness increases for skewed workloads (mainly for the scenario with 4 shards and 8 threads presented in Figure 4(d)). It is also important to note that queue sizes suffers with more variation and differences in quantity among threads in the skewed workloads scenario. This behavior can be observed in Figure 5(f) where threads $t0$ and $t1$, associated to shard 1 (Table III), receive more commands than all other threads and, consequently, their queues contain more requests to execute.

In this specific scenario with 8 shards and skewed workloads (Figure 5, right), we can observe how the static mappings of classes to threads affect performance. In these scenarios, thread $t1$ almost always have a larger amount of accumulated commands waiting for execution in their queues (Figure 5(f)) but, at the same time and for most of the workloads considered, $t1$ together with almost all threads also present high leves of idleness (Figure 5(b)).

Based on these results we can observe that a better distribution of work among the threads has a potential to improve system performance. For example, threads in idle states can receive commands originally designated, by the static mapping, to other overloaded threads. The main challenge is that this redistribution must respect all the conflict dependencies.

Impact of different execution costs (Figure 6): The final set of experiments studies how operation costs affect threads behavior, considering a system with 4 shards and 8 threads. We aggregated metrics averages of all threads, then we could range operation cost for all considered workloads. Operation costs affect threads idleness since they spend more time executing heavier commands and, consequently, are less prone to become idle. Light operations incur in faster command execution, thus allowing threads to have more sequential commands to execute, increasing the amount of barrier synchronization.

Figure 6(a) shows that thread synchronization idleness decreases slightly with higher operation costs. Skewed workloads present the same behavior (Figure 6(d)). For queue idleness, we can observe the same phenomenon both in balanced (Figure 6(b)) and skewed (Figure 6(e)) workloads scenario. Finally, the queue size is smaller for higher operation costs. This happens because servers need more time to process requests and send replies to clients. Consequently, clients remain most



(a) Average synchronization idleness (light costs)

(b) Average synchronization idleness (light costs)

(c) Average queue idleness (light costs)

(d) Average queue idleness (light costs)

(e) Average queue size (light costs)

(f) Average queue size (light costs)

Fig. 4: Results for 4 shards (8 threads), with balanced workloads (left) and skewed workloads (right).

of time blocked waiting for replies and few requests are issued in the system

VI. Conclusion

Although early scheduling improves performance of P-SMR, it restricts concurrency, resulting in thread idleness and unbalanced load among threads. We quantify these phenomena for many different configurations and discuss their reason in the paper. The study identifies novel directions in which early scheduler can be further improved. In particular, a better distribution of work among idle threads could translate into increased performance, something that we are currently investigating.

References

[1] E. Alchieri, F. Dotti, P. Marandi, O. Mendizabal, and F. Pedone. Boosting state machine replication with concurrent execution. In *Eighth Latin-American Symposium on Dependable Computing (LADC)*, 2018.

[2] E. Alchieri, F. Dotti, O. M. Mendizabal, and F. Pedone. Reconfiguring parallel state machine replication. In *SRDS*, 2017.

[3] E. Alchieri, F. Dotti, and F. Pedone. Early scheduling in parallel state machine replica. In *ACM SoCC*, 2018.

[4] A. Bessani, J. Sousa, and E. E. P. Alchieri. State machine replication for the masses with bft-smart. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014.

[5] C. E. Bezerra, F. Pedone, and R. V. Renesse. Scalable state-machine replication. In *IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014.

[6] M. Burrows. The chubby lock service for loosely coupled distributed systems. In *OSDI*, 2006.

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

[8] H. Cui, R. Gu, C. Liu, T. Chen, and J. Yang. Paxos made transparent. In *ACM Symposium on Operating Systems Principles*, 2015.

[9] D. da Silva Boger, J. da Silva Fraga, and E. Alchieri. Reconfigurable scalable state machine replication. In *Latin-American Symposium on Dependable Computing*, 2016.

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

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

[12] L. Glendenning, I. Beschastnikh, A. Krishnamurthy, and T. Anderson. Scalable consistency in Scatter. In *SOSP*, 2011.

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

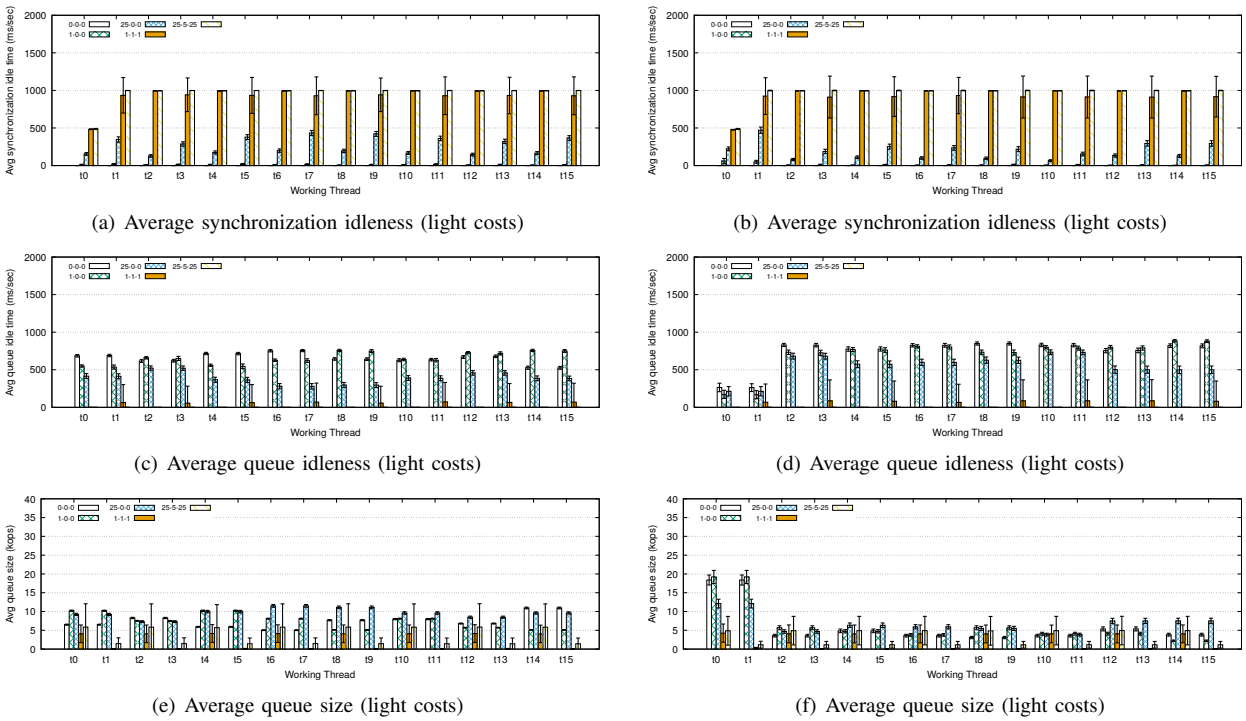
[14] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In S. Mullender, editor, *Distributed Systems*, pages 97–145. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

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

[16] J. D. J. C. Corbett and M. E. et al. Spanner: Google's globally distributed database. In *OSDI*, 2012.

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

[18] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin.



(a) Average synchronization idleness (light costs)

(b) Average synchronization idleness (light costs)

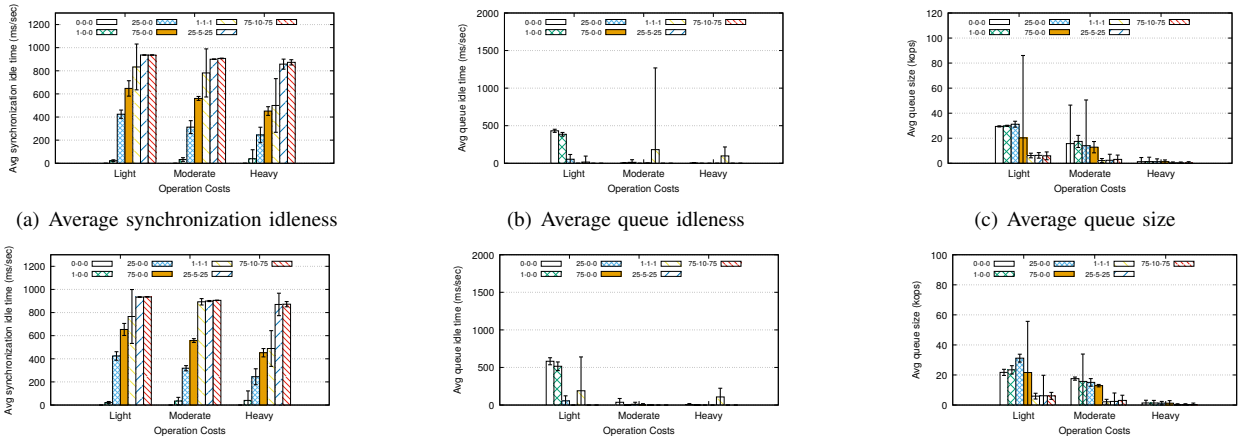
(c) Average queue idleness (light costs)

(d) Average queue idleness (light costs)

(e) Average queue size (light costs)

(f) Average queue size (light costs)

Fig. 5: Results for 8 shards (16 threads), with balanced workloads (left) and skewed workloads (right).



(a) Average synchronization idleness

(b) Average queue idleness

(c) Average queue size

(d) Average synchronization idleness

(e) Average queue idleness

(f) Average queue size

Fig. 6: Results for 4 shards (8 threads), different operation costs, with balanced workloads (top) and skewed workloads (bottom).

All about Eve: execute-verify replication for multi-core servers. In *Symposium on Operating Systems Design and Implementation*, 2012.

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

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

[21] L. H. Le, C. E. Bezerra, and F. Pedone. Dynamic scalable state machine replication. In *IEEE/IFIP International Conference on Dependable Systems and Networks*, 2016.

[22] J. Leung, L. Kelly, and J. H. Anderson. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. CRC Press, Inc., Boca Raton, FL, USA, 2004.

[23] P. J. Marandi, C. E. Bezerra, and F. Pedone. Rethinking state machine replication for parallelism. In *ICDCS*, 2014.

[24] P. J. Marandi and F. Pedone. Optimistic parallel state-machine replication. In *IEEE Int. Symposium on Reliable Distributed Systems*, 2014.

[25] O. M. Mendizabal, R. T. S. Moura, F. L. Dotti, and F. Pedone. Efficient and deterministic scheduling for parallel state machine replication. In *IPDPS*, 2017.

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

[27] N. Santos and A. Schiper. Achieving high-throughput state machine replication in multi-core systems. In *ICDCS*, 2013.

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