Waverunner: An Elegant Approach to Hardware Acceleration of State Machine Replication

Mohammadreza Alimadadi and Hieu Mai, *Stony Brook University;*
Shenghsun Cho, *Microsoft;* Michael Ferdman, Peter Milder, and
Shuai Mu, *Stony Brook University*

https://www.usenix.org/conference/nsdi23/presentation/alimadadi

This paper is included in the
Proceedings of the 20th USENIX Symposium on
Networked Systems Design and Implementation.

April 17–19, 2023 • Boston, MA, USA

978-1-939133-33-5

Waverunner: An Elegant Approach to Hardware Acceleration of State Machine Replication

Mohammadreza Alimadadi, Hieu Mai, Shenghsun Cho*, Michael Ferdman, Peter Milder, Shuai Mu
*Stony Brook University, *Microsoft*

Abstract. State machine replication (SMR) is a core mechanism for building highly available and consistent systems. In this paper, we propose Waverunner, a new approach to accelerate SMR using FPGA-based SmartNICs. Our approach does not implement the entire SMR system in hardware; instead, it is a hybrid software/hardware system. We make the observation that, despite the complexity of SMR, the most common routine—the data replication—is actually simple. The complex parts (leader election, failure recovery, etc.) are rarely used in modern datacenters where failures are only occasional. These complex routines are not performance critical; their software implementations are fast enough and do not need acceleration. Therefore, our system uses FPGA assistance to accelerate data replication, and leaves the rest to the traditional software implementation of SMR.

Our Waverunner approach is beneficial in both the common and the rare case situations. In the common case, the system runs at the speed of the network, with a 99th percentile latency of 1.8 μs achieved without batching on minimum-size packets at network line rate (85.5 Gbps in our evaluation). In rare cases, to handle uncommon situations such as leader failure and failure recovery, the system uses traditional software to guarantee correctness, which is much easier to develop and maintain than hardware-based implementations. Overall, our experience confirms Waverunner as an effective and practical solution for hardware accelerated SMR—achieving most of the benefits of hardware acceleration with minimum added complexity and implementation effort.

1 Introduction

Variants of State Machine Replication (SMR) are responsible for all reliable, consistent, and highly available online services. SMR is at the core of massive-scale systems such as cloud infrastructure coordination services [6, 26], large-scale distributed databases [13,25,63,69], and many other systems that require high availability and consistency [2, 14]. Due to their central nature in critical infrastructure, SMR implementations must be extremely robust and resilient. At the same time, the performance characteristics of the SMR dictate the performance of the overall service. As a result, mechanisms for reducing SMR operation latency and increasing throughput have received significant research attention.

A fundamental requirement of SMR implementations is that networked hosts must exchange multiple messages to agree on the shared state. While implementations that use traditional NICs and process all packets through the OS network stack are the most straight-forward, their performance is bounded by the large amount of CPU time spent on packet processing [8, 22, 28, 46, 57, 58, 68], limiting the throughput and drastically impacting the operation latency due to many traversals of the software network stack.

To overcome the CPU bottleneck, researchers have begun exploring hardware acceleration of network processing. For example, a recent work demonstrated the ZooKeeper broadcast protocol implemented entirely in reconfigurable hardware on an FPGA (Field-Programmable Gate Array). This implementation is able to approach line-rate throughput with operation latencies that are only marginally higher than the on-the-wire latency of the messages [30].

Unfortunately, although it is an impressive demonstration of hardware-acceleration capabilities, the hardware-only approach is too complex and brittle for practical deployment. Despite the improvements in the ease-of-use of hardware development toolchains, the ZooKeeper FPGA implementation required significant expertise, including a hardware version of the TCP/IP stack and all of the protocol details such as leader election and failure recovery. Implementing and debugging distributed protocols in hardware is significantly more difficult than user-level software implementations. Moreover, consensus algorithms—the core of SMR—are well known for being complex and error-prone in design and implementation. Properly capturing all corner-case behaviors in a hardware implementation is challenging and difficult to verify.

We observe that the complexity of SMR is actually in the uncommon routines. Indeed, the most common operation, the one that limits throughput and dictates operation latency, is extremely simple: a leader node receives requests and broadcasts them to the follower nodes, locally committing requests only after receiving acknowledgements from the followers. Other SMR routines, such as leader election and failure recovery, are indeed considerably more complex.However, these complex operations are used only in special circumstances, such as system bootstrap or replica failures. These operations are rare, not performance critical, and their traditional software implementations are fast enough for all practical purposes.

We propose a new approach to accelerate SMR by creating a hybrid hardware/software organization that implements only a small, simple, but performance-critical portion of the protocol in hardware and leaves the vast majority of the implementation in traditional user-level software. We showcase Waverunner, our approach for hardware acceleration of Raft [52], a well-known consensus protocol, where the entirety of the robust software implementation remains intact, adding only an extra high-performance hardware-accelerated path for the common-case operation. The Waverunner approach permits a clean separation of the complex operations from simple ones in the Raft application software, and reduces the complexity of transport protocol handling. Waverunner uses UDP for the common routines, but leaves all complex cases, such as error handling and view changes, to the traditional TCP-based software. Moreover, by restricting the Waverunner hardware to only the common case, we are able to leverage FPGA HLS (high-level synthesis) tools to automatically generate the hardware from its C++ description, significantly reducing implementation and adoption effort. Although our prototype system is based on Raft, the Waverunner technique is generic and can be easily adapted for other distributed protocols.

Waverunner is notable for its performance and simplicity. We achieve line-rate operation of Raft (85.5 Gbps after accounting for Ethernet frame overheads on a 100 Gbps network), with the vast majority of the SMR broadcast requests handled in three wire-delay latencies (median latency is 1.8 μs). At the same time, we retain the robustness, flexibility, and completeness of a software implementation—Waverunner includes a fully operational implementation of Raft without hardware acceleration and can smoothly transition between hardware-accelerated and software-only modes. In a failure test, Waverunner can recover from a leader crash within 1 second. While the implementation of Raft is by no means simple, leaving it in software is appealing for debugging, upgrades, and maintainability. Moreover, the 220-line HLS-based C++ implementation of the hardware, made possible by limiting the hardware only to a simple core routine, greatly simplifies modification and maintainability of the hardware components compared to a full-hardware implementation.

2 Background & Motivation

In this section, we briefly introduce the concepts of state machine replication and FPGAs, as well as the key idea and motivation of our Waverunner approach.

2.1 State Machine Replication

State machine replication (SMR) is the standard approach to build highly consistent and available systems [6, 13, 26]. It aims to provide a consistent view among replicas while tolerating replica failures in a practical environment where messages can be arbitrarily delayed and there are no perfect failure detectors. In the most common model, SMR replicates a sequence of log entries that contain the operations to be executed by each replica. The application is modeled as a deterministic state machine so that all replicas will have identical states after executing the same sequence of operations.

At the center of an SMR system is a consensus protocol, which is known to be complex and delicate. Most consensus protocols share a common leader-follower model, from early academic ones (e.g., Viewstamped Replication [51] and Paxos [40]) to more recent ones that are widely adopted in industry (e.g., Zookeeper Atomic Broadcast [26], and Raft [52]). Despite their differences, these protocols largely follow a two-stage structure: a leader election and recovery stage when the system elects a new leader and synchronize all replicas after possible failures, and a data replication stage when the leader replicates log entries to the followers as new requests arrive.[1]

2.2 Programmable NIC with FPGA

The last three decades have witnessed the emergence of a great mismatch between network speed and CPU performance. The bottleneck of a networked system has gradually moved from the network (NIC and switch) to the CPU and the OS software stack. To mitigate this problem, the use of programmable NICs with Field Programmable Gate Arrays (FPGAs) has emerged. Equipped with on-chip processing units and memory, FPGA-based NICs can process packets at line rate (e.g., 100 Gbps), with stable nanosecond-range latency. In comparison, the traditional software approach can process only several gigabits per second on a CPU core, with latencies measured in milliseconds.

Although capable of high throughput and low latency, FPGA hardware is notoriously difficult to program. Programming FPGAs is particularly challenging because it requires hardware development skills and knowledge of hardware description languages. Although high-level synthesis (HLS) tools make the FPGA programming process more accessible by translating functions from C++ to hardware, these tools are difficult to use effectively when the logic being implemented is complex. As a result, one of our goals in the design of Waverunner is to make sure that all functions that we implement in FPGA hardware are straightforward and simple, so they can be easily implemented with HLS.

2.3 Motivation

As the critical building block of large-scale systems, the performance of SMR has been a focus in many recent studies. High-performance SMR implementations use a wide range of advanced hardware, including programmable NICs with FPGAs [30], RDMA [2], and programmable switches [16,32,42]. In this work, we propose a unique hybrid approach to accelerating SMR with FPGAs: only accelerating the data plane and leaving the control plane, including election and recovery, to

[1] (Multi-)Paxos does not have a universally agreed algorithm, whether to implement it as a two-stage structure depends on the implementation [2, 10].

	Usage Ratio (%)		
	Control plane	Data plane	Application
Our Raft	~0 (1e-8)	88	12
NuRaft	~0 (1e-4)	92	8
etcd	~0 (1e-4)	72	28

(a) CPU cycle breakdown of Raft implementations.

	LOC (approx.)		
	Control plane	Data plane	Application
Our Raft	1500	200	5400
NuRaft	3600	1100	6000
etcd	1700	180	8900

(b) LOC estimates of Raft implementations.

	Usage ratio (%)
Network descriptor read	6.1
Network descriptor write	16.8
Other RPC cost	5.9
Memory allocation	9.6
Reference counting and memory free	13.3

(c) Breakdown of the data plane usage on system-related (non-logic) code in our Raft implementation.

Table 1: Raft Implementation Analysis

be processed by software. Our approach leverages the observation that the data plane dominates the system performance when the system is stable (most of the time), while the recovery part is only used in rare cases and is usually fast (seconds or less), so there is no need to accelerate it.

Table 1a and 1b show a CPU cycle analysis of various Raft implementations, including our own implementation in C++. In our experiments, we set the failure rate to once a year, and observe that the data plane consumes the majority of the CPU cycles while the control plane consumes almost none. Moreover, the majority of data plane usage is on common utilities such as networking, data serialization, memory allocation, etc. (Table 1c). Implementing the data plane in FPGA can avoid these costs. Implementing the control plane with software can minimize the programming effort needed in the hardware; it also allows us to rapidly iterate on the software implementation, as required when developing complex modules.

In this paper, we choose Raft as our acceleration target. This is another advantage of our approach: we can accelerate existing protocols and systems rather than develop entirely new ones. The hardware acceleration in deployment can then be optional. That is, the system can run either with or without the programmable NIC. This approach adds great flexibility in practice. Though we use Raft for our prototype, we believe our approach similarly applies to other common consensus protocols, as they have similar structure [64, 67].

3 Waverunner System Overview

Waverunner takes the standard state machine replication model: it replicates a sequence of operation log entries (messages) identically onto each replica. The target of replication is referred to as the application (e.g., a key-value store or a lock service). The application must be deterministic; after all replicas process the messages in the order they are recorded in the log, the replicas will all reach the same state. A client for SMR can be either the replicated application itself or an independent client that sends requests to the application. A replica server is either a leader or a follower. Only the leader accepts client requests and replicates these requests to the followers. Followers reject client requests, causing the clients to resend requests to the leader.

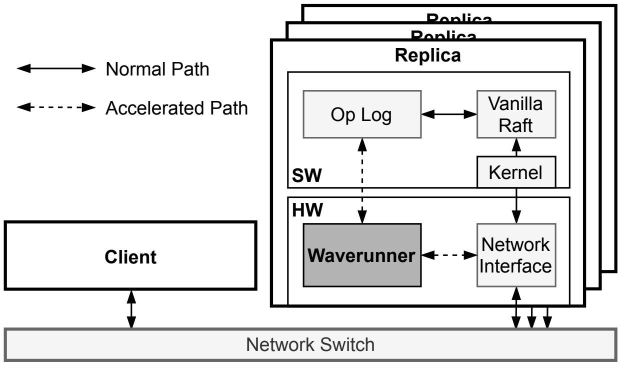

Figure 1: System Design Overview. The solid black lines represent the normal path when Waverunner is disabled, where network packets travel between the application (Vanilla Raft) and NIC via the POSIX interface and Kernel device driver; the dashed lines represent the accelerated path, where the NIC diverts incoming packets to Waverunner, which can send messages into the network via the NIC and/or deliver them directly into a buffer allocated by the application.

System Architecture. The system architecture, including its hardware and software components, is shown in Figure 1. The software includes a complete vanilla Raft implementation that uses conventional TCP sockets provided by the Linux kernel. Additionally, the software can access hardware configuration registers to control the Waverunner hardware. The software can disable Waverunner hardware acceleration, causing all received network packets to be delivered via the conventional network stack. However, if the software enables hardware acceleration, the network interface examines all received packets to identify those carrying Raft messages, and directs packets containing the most common Raft messages (client requests and data replication messages) to the Waverunner hardware protocol handler instead of the kernel network stack. To communicate with the software, the Waverunner hardware writes messages into a pre-allocated user-space log buffer, bypassing the kernel. Uncommon Raft messages and all other network traffic (e.g., ARP requests and ssh connections) are handled by the kernel like with a conventional NIC, regardless of whether Waverunner hardware acceleration is enabled or not.

Typical Waverunner Workflow. Waverunner takes advantage of the Raft leader election protocol to coordinate enabling and disabling hardware acceleration (Figure 2). When the system is first initialized, hardware acceleration is disabled by

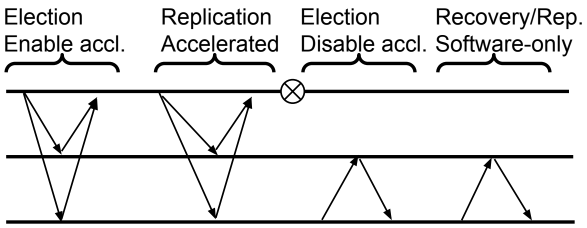

Figure 2: Waverunner Workflow

default. The Raft software triggers a leader election, selecting one of the replicas as the leader, and enables hardware acceleration. The system then replicates data with the assistance of hardware acceleration. The hardware takes care of the data replication operations and deposits the committed log messages into the user-space log buffer for the software application to handle. If a leader failure occurs, the system disables acceleration, conducts another leader election, and performs the requisite Raft protocol actions to resolve the problem before re-enabling acceleration. Other uncommon and complex operations are handled in a similar way. For example, after a series of failure events, the replicas may have diverged log sequences. The leader software will synchronize replicas by re-sending the missing log entries and potentially overwriting existing entries in the followers if necessary. After the complex conditions are addressed and all replicas are in the same state, the system can conduct another leader election and re-enable hardware acceleration.

The safety and consistency properties of SMR guarantee that 1) all replicas will commit the same log entry at the same position in the buffer, and 2) if an operation starts after another one commits, then the two operations will appear in the same order in the logs of all replicas.

4 Waverunner Hardware

In this section, we describe the Waverunner hardware (§ 4.1), explain its hardware data replication operation (§ 4.2), and detail the design of the communication mechanism necessary for Waverunner to achieve high performance (§ 4.3).

4.1 Hardware Architecture

Our Waverunner prototype is based on a traditional PCIe NIC architecture, where a NIC DMA engine, controlled by the host kernel network stack, streams data between the network interface and the host using the PCIe bus. When receiving packets from the network interface, the NIC transfers the packet contents into host memory and raises an interrupt to alert the CPU that the transfer is completed. To transmit packets, the NIC uses the PCIe bus to traverse a queue of packet contents populated in host memory by the software, transferring packets to the network interface.

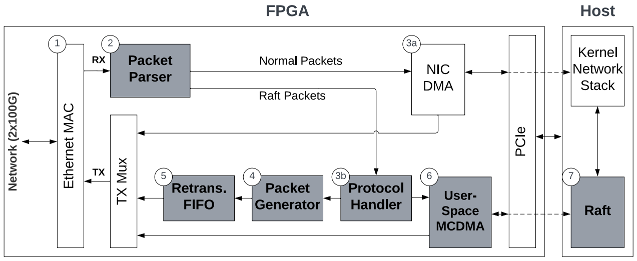

Figure 3: Waverunner Hardware Overview

Waverunner uses a bump-in-the-wire organization to extend the traditional NIC functionality with SMR hardware acceleration, as shown in Figure 3.

When packets arrive over the network ①, they are first streamed through a packet parser ② module to identify packets containing SMR protocol messages. Packets that do not contain SMR messages are streamed to the NIC DMA engine (3a) and are handled by the host kernel. If hardware acceleration is enabled and the packet parser detects a supported Raft message, the message is streamed to the Waverunner hardware protocol handler (3b) instead of the NIC DMA. The protocol handler performs internal bookkeeping on the protocol state, tracking Raft messages forwarded to the followers and their acknowledgements. If the received messages require one or more packets to be sent out (e.g., client requests must be replicated to the follower nodes), the protocol handler streams the messages to a packet generator module ④, which generates the packets and transmits them into the network. Outgoing packets are buffered in a circular buffer ⑤ that includes retransmission logic; when packet loss or reordering is detected, the protocol handler can signal the buffer to retransmit its contents. Notably, in this case, Raft packets are received and transmitted with minimum latency, entirely without host CPU involvement. Finally, the protocol handler determines if the received message must also be sent via User-Space MCDMA ⑥ to the Raft software for further processing ⑦. For each data replication Raft message, Waverunner will write two operation messages into the user-space log buffer in the host memory: 1) when the client request is received and forwarded to the follower replicas, the request is also written into the log buffer, and 2) when a sufficient number of acknowledgements are received from the followers, a commit operation is written into the log buffer. Application software then processes the log in order and performs the committed operations.

Messages to be delivered to the software are streamed to a descriptor-based, high-performance Multi-Channel DMA (MCDMA) engine that transfers the message contents directly to the user-space Raft software, bypassing the kernel network stack in a similar way to DPDK [21]. Each channel of MCDMA has a ring buffer of descriptors. Each descriptor consists of metadata such as the size and address of the data buffer, complete bit, and the pointer to the next descriptor.

Once the MCDMA completes a transaction, it updates the metadata, allowing the CPU software to consume the data. In the case that Raft traffic arrives too fast to transfer to the host and the MCDMA ring buffer becomes full, the Waverunner hardware will drop the packets before the packet parser, preventing the scenario where a message is processed by the Raft protocol handler but not transferred to the software.

To facilitate coordination between the host software and the hardware acceleration modules, the Waverunner hardware exposes a set of control and status registers (CSRs) accessible from the host software. Some registers, such as follower MAC and IP address, are used to configure the Waverunner system before the operation. An ACC_ENABLE register can be used to enable or disable the accelerated packet processing. Finally, the Waverunner hardware and software use several CSRs to maintain the Raft protocol state. After disabling hardware acceleration, the software can read the latest values of these registers from the hardware. The complete set of Raft protocol registers is listed in Appendix (§ C).

4.2 Raft Leader and Follower Operation

Although Raft is complex, Waverunner implements only the most common operations in hardware (pseudocode shown in Figure 4) and relies on the software to handle uncommon and complex interactions. Uncommon hardware-generated messages such as `AppendEntryReject` are sent to the software via the user-space log without additional processing, while complex messages such as leader election are not identified by the packet parser at all, allowing these messages to be delivered via the traditional NIC DMA (also passing through the OS network stack, and therefore allowing these messages to naturally leverage features such as reliable TCP transport).

When configured to act in a leader role, the hardware accelerator includes protocol handling logic for only two message types: client requests and follower acknowledgements. Upon receiving a client request (Figure 4, lines 2-12), the accelerator updates the Raft protocol state (e.g., `lastLogIndex`, `lastLogTerm`), streams the message contents to the packet generator (to transmit `AppendEntry` messages to the followers), and also sends the message to software. Upon receiving a follower acknowledgement (Figure 4, lines 33-43), the accelerator updates the protocol state and, if the operation is ready for commit (half of the followers have acknowledged), the acknowledgement message is sent to the user-space log. Only one acknowledgement is delivered to the software, subsequent acknowledgements for the same request are ignored. All other protocol messages identified by the packet parser are delivered to the software without updating protocol state and without response packet generation by the accelerator.

The follower role is even simpler, as it handles only `AppendEntry` messages. Upon receiving a message (Figure 4, lines 15-30), the follower first conducts several safety checks, including the `is_leader` check, checking if the previous log

```
1  // FPGA receives client request
2  function FPGA-AppendEntry(fs, op):
3    if fs.isLeader
4      prevLogIndex = fs.lastLogIndex
5      prevLogTerm = fs.lastLogTerm
6      logEntry = makePair(op, fs.currentTerm)
7      push(fs.host.log, logEntry)
8      fs.lastLogIndex++
9      fs.lastLogTerm = fs.currentTerm
10     send <'FPGA-append', op, prevLogIndex,
           prevLogTerm, fs.currentTerm, fs.commitIndex>
           to all except self
11   else
12     reject
13

14 // FPGA receives <'FPGA-append', op,
       prevLogIndex, prevLogTerm, term,
       commitIndex>
15 function FPGA-ReceiveAppend(fs, op, prevLogIndex,
       prevLogTerm, term, commitIndex):
16   if not fs.isLeader
17      and fs.currentTerm == term
18      and fs.lastLogTerm == prevLogTerm
19     if prevLogIndex > fs.lastLogIndex
20       reply with retransmission request
21     else if prevLogIndex < fs.lastLogIndex
22       ignore and return
23     fs.host.commitIndex = commitIndex
24     logEntry = makePair(op, fs.currentTerm)
25     push(fs.host.log, logEntry)
26     fs.lastLogIndex++
27     fs.lastLogTerm = fs.currentTerm
28     reply <'FPGA-appendOK', term, fs.id, fs.
         lastLogIndex>
29   else
30     reply <'FPGA-appendReject', fs.id>
31

32 // FPGA receives <'FPGA-appendOK', term, id,
       lastLogIndex>
33 function FPGA-ReceiveAppendAck(fs, term, id,
       lastLogIndex):
34   if fs.isLeader and fs.currentTerm == term
        and fs.matchIndex[id] < lastLogIndex
35     fs.matchIndex[id] = lastLogIndex
36     if fs.commitIndex < lastLogIndex
37       if a (majority-1) elements in matchIndex
            >= lastLogIndex
38         fs.commitIndex = lastLogIndex
39         fs.host.commitIndex = fs.commitIndex
40         if the request does not read the
               system state (e.g., a blind write
               in a key-value store)
41           notify the client of commit and skip
                 the reply from the host
                 software (in ApplyLog)
42   else
43     halt and notify host to handle failures
```

Figure 4: Pseudocode of the hardware accelerator.

index and term match (ruling out the case of lost or duplicated packets), and confirming that the message `term` value matches the `currentTerm` variable. Then the follower updates the protocol state, streams the message into the packet generator to produce an acknowledgement (`AppendEntryAck`) or rejection (`AppendEntryReject`) message, and updates its local `commitIndex` according to the message `commitIndex`.

It is worth noting that the protocol-specific Waverunner hardware accelerator actions and logic are intentionally primitive. For an FPGA implementation, the logic of these operations is automatically translated from their C++ description.

4.3 User-Space Log Considerations

The behavior of communication between the hardware and software is critical for high Waverunner performance. Raft messages must be delivered by MCDMA (shown in Figure 3) to the log accessible in user-space software without being a bottleneck in the system. We describe three critical aspects of the design of the User-Space MCDMA block.

First, the MCDMA component requires *fast write response*. In our platform, MCDMA is connected to the PCIe bridge using AXI Memory Mapped (AXI-MM) interfaces, where the MCDMA's AXI master interface writes data to the PCIe bridge's AXI slave interface to transfer data to the user-space buffer. Under the AXI-MM protocol, MCDMA first issues the write address to the PCIe bridge, followed by the write data. After the data transfer is complete, the PCIe bridge sends a write acknowledgement response to the MCDMA. In our experiments, we observed a very high latency (300 to 400 cycles) for the PCIe bridge to send the write acknowledgement response after the data transfer. During this period, MCDMA stops processing descriptors and accepting packets from the protocol handler, negatively affecting the system performance. To solve this problem, we insert a small custom FIFO between the MCDMA and the PCIe bridge. The custom functionality of this FIFO is to send write acknowledgement responses immediately after the write operations are completed on the MCDMA side, thereby hiding the high acknowledgement latency introduced by the PCIe bridge and allowing MCDMA to process descriptors for the other channels.

Second, we introduce *batched MCDMA operation* in the hardware. For each MCDMA operation, in addition to the data transfer, there are also descriptor read and write operations across PCIe. The descriptor reads and writes are overheads, which significantly reduce the effective bandwidth for the actual PCIe data transfers. To minimize the descriptor overhead, we designed a hardware module to batch multiple consecutive message writes into a single transfer to amortize the descriptor overheads, solving the performance bottleneck and improving throughput over PCIe. The batching hardware collects messages until one of two conditions is met: either a preconfigured batch size is reached or a pre-configured timeout is reached without new messages arriving on the given channel. With the second condition, the latency increase caused by batching is negligible.

Finally, although Waverunner hardware acceleration significantly reduces the work that must be done by the CPU of the leader replica, the application code that executes committed operations still consumes CPU resources and can become the bottleneck in the system. One design option is to use a software dispatcher to handle incoming log messages from the hardware and coordinate spreading the handling of the log messages across software threads running on different cores. However, at our target throughput, a software dispatcher would itself become the system bottleneck. Instead, Waverunner shards log messages in hardware, using separate DMA channels to write log messages destined for processing by different application software instances. The leader replica runs multiple application processes, one per core, with each process having its own user-space log buffer into which the hardware deposits Raft messages belonging to the corresponding shard. This approach mirrors the operation of high-performance NICs that allow the software (e.g., DPDK) to install rules into the NIC hardware to steer incoming packets to different descriptor rings or queues to be handled by different cores.

4.4 Transmission with UDP

Our implementation uses UDP to transmit packets between FPGAs. UDP is unreliable for transmission and suffers from packet loss, duplication, and reordering with traditional hardware and software stacks. However, using UDP in the Waverunner hardware greatly reduces the hardware complexity compared to a TCP implementation. To handle the cases of UDP packet loss and reordering, our hardware implements a small retransmission buffer. The buffer, placed between our packet generator and the TX Mux (shown in Figure 3), holds all recently sent packets. In the event of packet loss or reordering being detected in `AppendEntry`, the protocol handler requests the packet generator to create a retransmission request packet (Figure 4, lines 19-20). When a retransmission request arrives at the retransmission buffer, instead of writing it into the buffer, the control logic triggers a retransmission of all packets currently in the buffer. The retransmission is finished when all the packets in the buffer have been transmitted. During the retransmission, incoming packets continue to be written to the tail of the buffer. The buffer is 256 KB, ensuring that in the worst-case scenario in our system (192 byte packets at 26 Mpps) packets will remain in the buffer, eligible for retransmission, for 52 μs. This time is sufficient to tolerate 28 consecutive retransmission requests in our testbed, and is well beyond the round-trip latency of modern datacenter networks. Notably, the retransmission buffer does not affect system correctness; it simply avoids triggering software failure recovery in case of UDP packet loss or reordering in the network. In the extremely unlikely case that persistent retransmissions repeat until a lost packet is no longer present in the retransmission buffer, the hardware triggers a conventional Raft failure recovery in software.

5 Waverunner Control Plane

In this section, we describe our software components. The software is primarily in charge of the uncommon routines of the system, such as bootstrapping and recovering from an abnormal system state. This functionality includes electing a new leader, synchronizing data between a new leader and the replicas, and controlling hardware acceleration. We begin with a vanilla Raft software implementation and add support for interaction with the accelerator hardware. For completeness, we include a discussion of the full Raft implementation, and we explicitly highlight the Waverunner-specific design decisions and additions for software-hardware interaction.

5.1 Switching to Software via Leader Election

Leader failures are handled by re-electing a new leader. Leader election can be initiated by any replica. Each replica keeps a timer in software, starting an election if a timeout is triggered due to a lack of new messages received from the leader. Each replica uses a randomized timeout value, thereby reducing the probability of competition. When the system is idle, a timed loop in the leader's software sends empty requests to the hardware to avoid unnecessary elections. This does not cause extra overhead compared to a software-only Raft implementation, as it would have a similar timed loop to send empty `AppendEntry` heartbeats.

When a follower triggers an election and requests to become leader (referred to as a candidate), its software will first disable the hardware acceleration and wait for the hardware to complete processing of the packets in the hardware accelerator pipeline and MCDMA batch queues.

The candidate software will increment its own `term` and stop responding to replication requests with lower terms, and send a `RequestVote` message to the other replicas. Even if the replicas that receive the message are using hardware acceleration, the message will pass through the hardware transparently and be directly handled by the software. Each replica will confirm that its `term` is smaller than that of the candidate, and then disable hardware acceleration and check if the candidate has a more up-to-date log by comparing the `lastLogIndex` and `lastLogTerm` of the candidate with its own. If the candidate is more up-to-date (or the same), the receiver grants the vote and sets the leader `id` to the candidate. If the candidate receives enough votes (including itself for a majority), it transitions into the leader role.

When the new leader software takes over, the system is a fully capable software Raft. It can perform any traditional system maintenance operations, such as view change. The software handles all Raft routines not implemented in hardware, such as synchronizing logs on the replicas.

5.2 Synchronizing Missing Logs

When a leader crashes or communication with it fails, the replicas may be left unsynchronized, such that some replicas may have longer logs. In more complex cases, such as election competition or consecutive leader crashes, replicas may even have different *uncommitted* logs at the same log position. Raft's (or any consensus protocol's) logic for handling these situations is complex. To ensure a simple hardware implementation, Waverunner keeps the implementation of replica log synchronization entirely in software.

After a new leader is elected, it initiates synchronization of the replica logs by sending an `AppendEntry` message containing a special `noop` operation to all replicas. If a follower has fallen behind or has non-matching logs, it will reply with an `AppendEntryReject` message, indicating a log mismatch. The leader will then send earlier log entries until the follower acknowledges accepting these logs, ensuring that the follower is synchronized with the leader. The `noop` commit entry is necessitated by the Raft protocol, as simply counting existing log entries in all replicas may fail due to a corner case in the Raft algorithm. (This is a documented idiosyncrasy of the Raft protocol (§5.4.2) [52].)

Note that there is a limit on the maximum number of entries that a replica can hold in its in-memory log. If a replica is down for an extensive amount of time, or a new replica is added to the system, that replica cannot catch up via the aforementioned approach because the leader will have discarded its older logs from memory. In this case, the leader should send a snapshot of the application dataset to the failed replica. The snapshot will contain the system state up to a particular log position which is still in the leader memory, allowing the leader to catch up the replica by sending it the entries starting from the snapshot log position. Similar to the original Raft work [52] and other recent works on speedy SMR [2, 49], the creation of the snapshot is application-specific and is orthogonal to the scope of this paper. For example, a standard approach is available in [65].

5.3 Enabling Hardware Acceleration

After the logs of all reachable replicas are synchronized, the leader will increment the `term` and send out `RequestVoteFPGA` messages to the synchronized followers. This message is identical to a normal leader election, with the additional side-effect of causing the followers to enable hardware acceleration. Once the leader receives acknowledgements from half of the followers (reaching a majority when including itself), the leader enables its own hardware acceleration. All future log entries are replicated by the hardware, without involving the CPU.

Separating hardware and software into different terms provides several benefits. First, it keeps the Raft algorithm intact, eliminating the need to prove correctness of our changes. Second, this approach is easy to implement, debug, and maintain, because the term of a log entry indicates whether it was initially replicated by software or hardware.

If a replica does not receive RequestVoteFPGA, but later receives an AppendEntry (e.g., a crashed replica rejoins), it will see a higher term in the message. Whenever a replica sees a higher term in AppendEntry than its currentTerm, or if it cannot find an entry at prevLogIndex in its log, it will disable hardware acceleration (if it is enabled) and reply with AppendEntryReject. When the leader receives an AppendEntryReject, it will disable hardware acceleration, trigger an election to go into a new term, and synchronize with the straggling replica using software. Afterwards, the system will transition back to running with hardware acceleration using the previously described steps. To summarize, Waverunner uses a unified approach to address all possible cases that deviate from the normal replication routines, including the possible loss of messages, message delays, temporary server anomalies, etc.

6 Evaluation and Results

We evaluate Waverunner with real-world applications and off-the-shelf hardware. The major questions we answer are:

- What is Waverunner's replication performance?
- Can Waverunner efficiently recover from a failure?
- How well does the Waverunner approach perform with real-world applications?
- How does Waverunner compare to other hardware-accelerated SMR approaches?

6.1 Setup

We conduct our evaluation on a 3-replica Waverunner cluster, with several additional client machines to issue requests in an open-loop manner. Each replica has two Intel E5-2695v4 CPUs, 1TB DDR4 memory, and a Xilinx U280 FPGA connected via PCIe gen 3 x16. Each FPGA has two 100 Gbps QSFP28 ports. Our replicas are connected to one switch and clients to another, and there is a 100 Gbps fiber connecting the two switches. On the FPGAs, we implemented the Waverunner hardware accelerator using Vivado HLS. For the control plane, we modified our C++ Raft implementation to coordinate with the Waverunner FPGA hardware.

In addition to Waverunner, we also evaluate the replication performance of two SMR systems for comparison:

- Mu [2]: An RDMA-based SMR implementation, which aims to provide microsecond level latency for application replication. It has a custom leader-follower consensus protocol. In the Mu implementation, all requests originate from the leader. This gives it an advantage over Waverunner, where requests are sent over the network from clients.
- DPDK-Raft: An in-house DPDK-based Raft implementation. We built our own DPDK-Raft implementation because the state of the art Raft implementation (in eRPC [34]) is equipped to perform latency tests and we are interested in both latency and throughput experiments. Our DPDK-Raft achieves similar latency as eRPC Raft.

Both Mu and DPDK-Raft use a 100 Gbps Mellanox ConnectX-4 NIC included in the replicas with the same connection specifications as the FPGA (i.e., PCIe gen 3 x16 and 100 Gbps QSFP28).

6.2 Methodology

We present two key metrics: throughput and latency. We report throughput in millions of request packets per second (Mpps) and total network bandwidth used (Gbps). For latency, we report the time in microseconds (μs) and present the median (50th percentile) and tail (90th and 99th percentile) measurements. To improve accuracy, whenever possible, we collect the results using internal hardware performance counters on the FPGAs, NICs, and switches.

We implement our client using DPDK to achieve high performance and accurate measurement. Precise control of the offered load at the client is difficult to achieve, so we set approximate targets and plot all results by using the actual measured request rates. As a result, experiments that vary the request rate may not have results with round throughput values (e.g., our plots may show 4.1 and 5.2 Mpps, rather than precisely 5 Mpps). For measuring end-to-end latency from the client and to avoid subjecting latency measurements to client-side queueing, we use a sampling approach by concurrently running two DPDK client configurations: one to apply the target load and a second lightly-loaded client (using a separate NIC) to precisely measure the latency.

6.3 Replication Performance Results

We first focus on evaluating the SMR replication performance without a specific application (where the "application" simply discards committed operations) to better understand the capability of Waverunner in a clean environment. The clients send requests to the replica cluster using small random packets (50 bytes for Waverunner and DPDK-Raft, 64 bytes for Mu due to its implementation restriction). We sweep the request rate in steps of approximately 1 Mpps and measure the replication throughput and latency at each step. We described Waverunner's MCDMA batching with timeout mechanism in Section 4.3. An adaptive batching strategy can minimize latency under all load scenarios, however, we found that for our evaluation, a constant batch size was sufficient to limit PCIe transfer overheads.

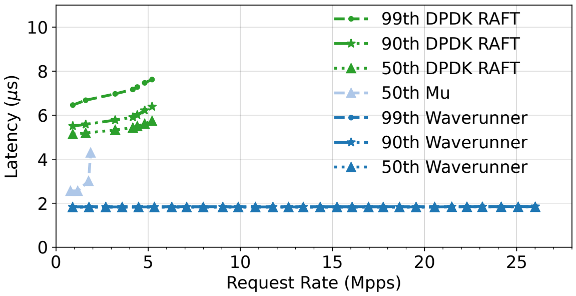

Figure 5: Performance of Packet Replication

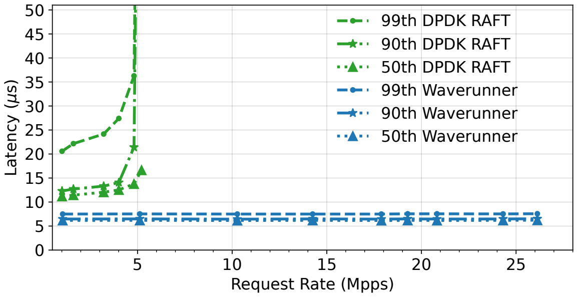

Figure 6: End-to-end Latency

Throughput. Figure 5 shows that the maximum request rate Waverunner can achieve is 26 Mpps, which is bounded by the leader's network receive bandwidth. Under this request rate, the bandwidth utilization of the leader FPGA approaches 85.5 Gbps, the maximum theoretical bandwidth achievable by the network transceivers and switch.[2]

Beyond this throughput, the FPGA transmit FIFOs experience back pressure from the MAC, which cascades to the Waverunner protocol handler and causes packet loss of incoming client requests and follower acknowledgements. This result indicates that Waverunner can fully utilize the available network bandwidth and achieve the maximum request rate.

On the contrary, Mu and DPDK-Raft cannot saturate the network bandwidth with one request per packet, resulting in only ∼2 Mpps and ∼5 Mpps peak request rate, respectively. Mu has to rely on client-side batching (aggregating multiple requests into one packet) to increase the request rate and utilize the available network bandwidth. However, doing so also drastically increases the latency. For DPDK-Raft, the throughput is bottlenecked by descriptor ring handling via PCIe. At peak throughput, DPDK-Raft starts to drop packets because the CPU cannot process and release RX descriptors at the same rate as the incoming packet stream, a situation we overcome in Waverunner by transferring multiple requests using each descriptor (§ 4.3).

Latency. Figure 5 also shows the replication latency for Waverunner, Mu, and DPDK-Raft at various request rates. Replication latency is measured from when the leader receives a client request, until the leader receives the corresponding acknowledgements from half of the followers.

The Waverunner replication latency is effectively constant at 1.8 μs, only marginally higher than the RTT of a minimum-sized packet in our network (1.68 μs). There are two characteristics of Waverunner's latency that are notable: the median, 90th-, and 99th-percentile latencies are all nearly identical, and as the request rate increases, the latency does not increase, all the way until network bandwidth is exhausted. These latency characteristics are a unique advantage of an FPGA implementation [30], as most of the components in the FPGA hardware have low and constant latency that is immune to queuing effects, allowing the replication latency to remain stable. In contrast, both Mu and DPDK-Raft exhibit substantially higher 90th- and 99th-percentile latency compared to the median latency, and the latency grows as the request rate increases and the CPUs become busier, amplifying interference and system queuing effects. As a result, the replication latency of Waverunner is significantly lower than Mu and DPDK-Raft. The worst Waverunner 99th-percentile tail latency is approximately 1/3 (36%) of the best median latency of DPDK-Raft (5 μs) and 40–80% of Mu (2.5–4.3 μs).

For completeness, we also measure the end-to-end latency on the client for Waverunner and DPDK-Raft, as shown in Figure 6. The end-to-end latency on the client includes the replication latency, the RTT between the client and the leader (with 1 switch placed between the two), and the time for the client to process the packets. For Waverunner, this adds another 4–6 μs for the median and tail latencies. For DPDK-Raft, the additional time is much larger because DPDK-Raft relies on batching and buffering to achieve maximum throughput, which add extra cost to overall latency.

Performance with Different Packet Sizes. In addition to minimum sized packets, we investigate the effect of larger requests on Waverunner, shown in Figure 7. We maintained a constant throughput of 1 Mpps and varied the payload size accordingly. For minimum-sized packets, the replication latency is 1.79 μs. As the payload grows, the latency increases slowly to a maximum of 2.13 μs. Importantly, the 99th-percentile latency remains approximately the same as the median.

CPU Utilization. Compared to the RDMA and DPDK approaches, Waverunner has an important advantage, especially at high request rates: it places far less pressure on the host CPU cores. For example, to achieve peak performance in our tests, DPDK-Raft saturates 18 CPU cores. In contrast, Waverunner consumes negligible host CPU resources because it only needs to manage the MCDMA descriptors for the operation log. This leaves CPU resources almost entirely free, allowing them to be used by the target application.

[2] Each Ethernet frame includes a 7-byte preamble, a 1-byte start of line delimiter, and a 12-byte inter-packet gap, which together account for the approximately 14.5 Gbps gap to the advertised 100 Gbps line rate.

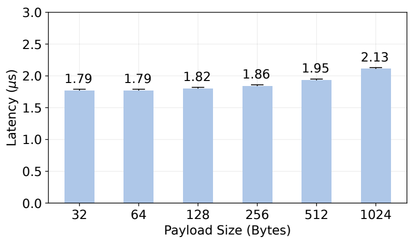

Figure 7: Latency across payload sizes. Bars show 99th percentile.

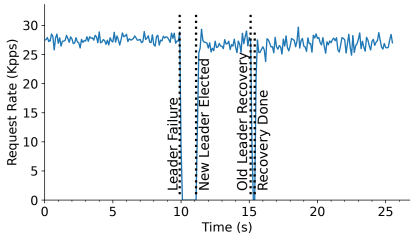

Figure 8: Request rate during failure recovery.

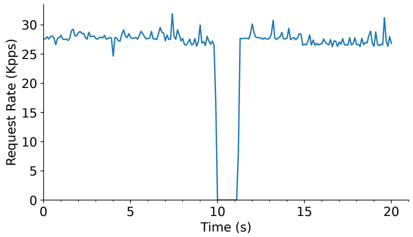

Figure 9: Request rate during view change.

6.4 Fault Tolerance and View Change

We evaluate Waverunner fault tolerance by injecting a leader failure on a healthy cluster. As expected, the leader failure triggers a leader election. At a later point, we resume the old leader, allowing it to rejoin the SMR cluster. Figure 8 presents the behavior by showing the request rate measured at the client as these failure-related events take place.

In this experiment, the system first runs normally for ten seconds, then we halt the leader to simulate a failure. At this point, requests from the clients fail to proceed because the system has no available working leader, and the throughput of the system drops to 0.

We configured the followers with a one second timeout. After timeout, a new leader election begins. This can be seen on the graph slightly more than one second after the failure, where the system resumes processing requests after the clients discover the new leader and resume sending requests.

At 15 seconds, we resume the old leader; this is recognized by the new leader when the old leader rejects the replication requests that it receives from the new leader. Because the old leader is missing log entries from its down time, the new leader starts recovery by catching up the old leader's replica. After approximately 200 ms, the log recovery completes and the system re-enables hardware acceleration, showing that the hybrid architecture of Waverunner can correctly and efficiently recover from failure.

Similar to the failover test, we performed a view change test; results are shown in Figure 9. Initially, the system runs with three replicas. After ten seconds, we send a view change command to the leader to reconfigure the system down to two replicas (removing one follower). The leader disables hardware acceleration, initiates a leader election to advance to a new term, completes the view change, and then re-enables hardware acceleration.

6.5 Real-world Applications

To understand how Waverunner performs with real-world applications, we evaluate three key-value stores: an in-memory hash table, Memcached, and Redis. We modified the applications to receive requests from Waverunner instead of the conventional network sockets. This enables low latency and high throughput operation as Waverunner bypasses the kernel to send and receive packets. Scalability across cores is achieved through sharding; the number of replication groups is the same as the number of threads. Unless otherwise indicated, we use 8-byte keys and values, which makes the packets (including network header) 135 bytes for the hash table, 150 bytes for Memcached, and 156 bytes for Redis. We use open-loop clients that perform operations on uniformly distributed keys. Although the applications were not originally designed with SMR in mind, using them with Waverunner transforms them into consistent high-availability systems.

Throughput. To evaluate applications throughput, our client can send a mixture of PUT and GET requests. Both PUT and GET requests are replicated in Waverunner, but are processed differently. For PUT requests, Waverunner responds to the client when the request is committed in Raft (acknowledged by the majority of replicas), allowing the application to handle the request log in the background, eventually updating the key-value store. For GET requests, Waverunner does not generate a client response, instead relying on the application to execute the operation from the log by retrieving the relevant data and sending them to the client. Figure 10 shows the peak sustained throughput we observed. The peak throughput of the original Memcached and Redis implementations is 1.5 Mpps, while the Waverunner implementations reach 20.7 Mpps and 19.9 Mpps, respectively. Redis has a lower throughput because it dynamically increases the size of its hash table and needs to rehash every entry. However, Memcached has a constant-size hash table that is initialized at the start.

We also examined the effect of different GET/PUT ratios on Memcached, shown in Figure 11. We observed that Memcached needs more CPU cycles for PUT requests than GET requests because, in addition to fetching the query in the key-value store (like a GET does), it also locks the region containing the key to update. As a result, higher GET ratios observe higher throughput. For Redis, we observed that changing the GET/PUT ratio does not affect the throughput.

Latency. We measured the end-to-end latency of GET and PUT operations in a 50% GET/PUT test for Memcached and Redis, as shown in Figure 12. To show the effect of different

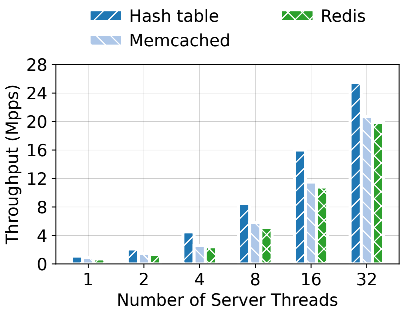

Figure 10: Peak application throughput.

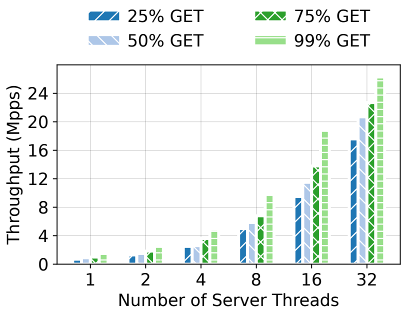

Figure 11: Memcached throughput across request ratios.

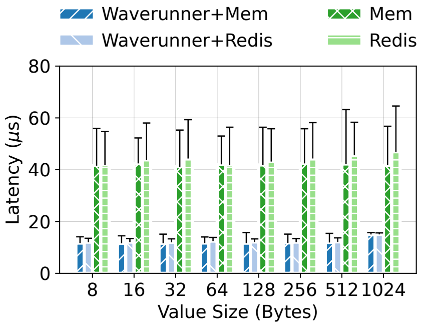

Figure 12: End-to-end latency. The bars indicate 99th percentile latency.

packet sizes, we varied the size of the value from 8 to 1024 bytes. Without Waverunner, Memcached and Redis exhibit a stable end-to-end latency of 41–44 μs, which is much lower than reported in the prior work [2, 17]. We attribute the lower latency to our high-end Mellanox NICs, both in client and server. When run on commodity NICs, the results (not shown) are much higher and closer to the previously published work.

Waverunner has a lower latency of 11.69 μs and 12.07 μs for Memcached and Redis respectively in most cases. With Waverunner, the leader FPGA generates responses for PUT requests without any CPU involvement, resulting in significantly lower end-to-end latency. For incoming GET requests, Waverunner delivers them to the application and transmits the responses to the network without involving the kernel. In summary, applications using our Waverunner framework can achieve performance comparable to kernel bypassing techniques (e.g., DPDK) for processing GET requests, and better performance for PUT requests.

6.6 Comparison to Prior Work

In this section, we discuss a comparison of Waverunner with Consensus in a Box [30] (referred to as ZABFpga below), a recent implementation of the ZooKeeper SMR protocol on an FPGA. We did not find the exact code release corresponding to ZABFpga online, which complicated our ability to study its operation in our environment. Although we did locate a project that appears to include ZABFpga's code [1], we found it challenging to port to our platform and extract from it just the ZABFpga components. A ground-up re-implementation of ZABFpga would constitute a major development effort. Such difficulties highlight the challenges in the development, portability, and maintenance of FPGA-based systems, stressing the benefits of the Waverunner approach in leaving the majority of the SMR protocol in software and implementing the hardware components using relatively portable HLS.

Based on what we can infer from the description of ZABFpga, the system has excellent performance, and would likely exhibit throughput and latency on par with Waverunner if ported to our environment, which has 100 Gbps NICs compared to 10 Gbps in the original paper. However, the ZABFpga system clearly required a drastically more complex development effort and would incur massively higher maintenance and troubleshooting cost. This is because ZABFpga implemented the ZooKeeper protocol completely, including the leader election and failure recovery, in custom FPGA hardware. This approach also required implementing the application (a key-value store) on the FPGA, including the ability to store the replication log in DRAM connected to the FPGA. Based on the description in the paper, the replication latency is 3 μs while Waverunner has 1.8 μs. It would be unfair to compare two designs on the throughput as the ZABFpga uses 10Gbps NICs to communicate with other nodes.

7 Related Work

Hardware Accelerated Networking. Early works on hardware acceleration in NICs offered a range of features, from simple ones such as checksum calculation and receive-side-scaling (RSS), to complex ones such as RDMA and TCP offloading engines [48]. Although earlier network hardware accelerators hard-wired the acceleration functionality, the trend has shifted toward programmability, with modern *SmartNIC* devices comprising programmable CPU cores [47] or programmable FPGA fabrics [50]. Modern advanced accelerators include functionality such as in-line handling of protocol encapsulation, VLAN processing, and encryption and decryption of data streams [18, 43]. Waverunner is a SmartNIC that accelerates replication routines of the Raft protocol. Like other SmartNICs, we utilized a bump-in-the-wire architecture to accelerate the replication routines in the FPGA. AccelNet [20] accelerates network services for virtual machines on SmartNICs in data centers. However, the acceleration is only loosely coupled with the application, such that whenever AccelNet does not have a rule for a packet, it consults the application to install the missing rule. On the other hand, Waverunner is more specialized, as it identifies Raft packets and does not need software support to handle other protocols. P4 [4] is a high-level language for network functions with implementations on FPGAs [3, 7, 9, 27, 45, 66]. hXDP [5], an FPGA-based NIC, uses soft cores to execute eBPF, another high-level language to describe network functions. These systems target system-wide packet processing, whereas Waverunner is specialized and optimized for processing only the Raft replication routines.

Caribou [29] implements a hardware accelerator for high-performance databases computations, including the full functionality of a fault-tolerant key value store inside an FPGA. KV-Direct [41], CliqueMap [61], Xenic [59], and RedN [55] extend RDMA primitives to enable remote key value operations to main memory. Waverunner takes a similar approach which puts requests in the follower's memory through PCIe. This is unlike Caribou, which does not use the host and relies on the FPGA to execute the database application. Floem [53], NICA [19], iPipe [44], FlexTOE [60], and FairNIC [23] provide a framework that can offload network applications such as Memcached on programmable SmartNICs. Although its goals of low latency and high throughput are similar to Floem and NICA, Waverunner targets the Raft distributed protocol, using hardware acceleration for the communication among multiple nodes rather than for the application logic.

State Machine Replication. State machine replication achieves fault tolerant, highly available services by leveraging consensus protocols [26, 40, 52]. From among the popular consensus protocols, Waverunner implements the Raft protocol [52], offloading the replication routines to a hardware accelerator. Several prior studies proposed ways to increase the performance of SMRs. eRPC [34], FaSST [35], and Breakwater [12] use an RPC library on top of the NIC API and RDMA, respectively, to provide low latency communication for applications. PigPaxos [11] relays messages by subgouping followers. These works optimize the network IO bottleneck, increasing the performance considerably, but they still suffer from the CPU bottleneck for implementing all parts of the protocol. Increasing parallelism of SMR [14, 24, 36, 56, 62] can further improve the performance, which Waverunner can benefit from for the application design. HovercRaft [39] moves SMR from the application layer to the transport layer and optimizes Raft to avoid the CPU and network IO bottleneck. Similarly, Waverunner addresses the same bottlenecks by offloading the network communication and replication to the hardware accelerator. Some SMR systems leverage high performance programmable switches [15, 32, 33, 42]. Rather than changing the network infrastructure, Waverunner employs a hardware accelerator in the NIC of each replica to accelerate the replication communication and operations.

There are several studies on low latency SMR through RDMA [17, 31, 37, 38, 54, 65], some of which are based on variants of Paxos. Although these works offer low latency, they are still bounded by the CPU bottleneck, as all of them cannot send packet at line rate with minimum size packets, and have high replication latency. Mu [2] introduces a microsecond latency SMR in which the leader writes requests in the log of each replica in only one round of RDMA transfers, without involvement from the CPUs on the follower nodes. Comparably, Waverunner achieves constant microsecond latency using FPGAs without changing core routines of the Raft protocol, while achieving high throughput on minimum size packets. ZABFpga [30] accelerates the Zookeeper consensus protocol using an FPGA and shows the benefits of hardware accelerator for SMRs in terms of latency and throughput. Waverunner achieves similar performance, but presents a design for the replication routines of Raft protocol while leaving all complex functionality of the Raft protocol (such as leader election and failure recovery) and the application (a key-value store) in traditional software.

8 Conclusions

We presented Waverunner, a hardware-software hybrid approach for implementing state machine replication. Our approach relies on the observation that, despite the complexity of SMR, the most frequently used routines can be easily implemented in hardware, while leaving the complex protocol and application logic in traditional software. Using this approach, we attain the best characteristics of the prior work, achieving the performance of full-hardware implementations while retaining the flexibility of software implementations with hardware-assist mechanisms such as DPDK and RDMA.

Waverunner is a practical realization of our approach. It is elegant and simple, leveraging a complete software implementation of the Raft protocol at its core and demonstrating how the most-frequently used functionality can be offloaded to hardware using only 220 lines of C++ HLS code. Waverunner achieves network line-rate throughput, nearly constant mean and tail (99th percentile) replication latency regardless of throughput, and leaves the majority of the CPU processing power available for the target application.

Acknowledgements

We thank our shepherd Maria Apostolaki and the anonymous reviewers of NSDI '23, OSDI '22, SOSP '21. We thank Yida Wu for implementing the first version of DPDK Raft; we also thank Satya Jain and Sergey Madaminov for their contribution to this work in the early stage. This research was supported in part by NSF CCF 2007362, CNS 1763680, CNS 2130590, and CNS 2214980.

References

[1] FPGA ZooKeeper source code. https://github.com/fpgasystems/caribou.

[2] Marcos K Aguilera, Naama Ben-David, Rachid Guerraoui, Virendra J Marathe, Athanasios Xygkis, and Igor Zablotchi. Microsecond consensus for microsecond applications. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, November 2020.

[3] P. Benáček, V. Puš, J. Kořenek, and M. Kekely. Line rate programmable packet processing in 100gb networks. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, 2017.

[4] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, and David Walker. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review (CCR)*, 44(3), July 2014.

[5] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. hXDP: Efficient software packet processing on FPGA NICs. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, November 2020.

[6] Michael Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, November 2006.

[7] Jakub Cabal, Pavel Benáček, Lukáš Kekely, Michal Kekely, Viktor Puš, and Jan Kořenek. Configurable fpga packet parser for terabit networks with guaranteed wire-speed throughput. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, page 249–258, 2018.

[8] Qizhe Cai, Shubham Chaudhary, Midhul Vuppalapati, Jaehyun Hwang, and Rachit Agarwal. Understanding host network stack overheads. In *Proceedings of ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, August 2021.

[9] Z. Cao, H. Su, Q. Yang, J. Shen, M. Wen, and C. Zhang. P4 to FPGA-a fast approach for generating efficient network processors. *IEEE Access*, 8, 2020.

[10] Tushar Deepak Chandra, Robert Griesemer, and Joshua Redstone. Paxos made live: an engineering perspective. In *Proceedings of ACM Symposium on Principles of Distributed Computing (PODC)*, August 2007.

[11] Aleksey Charapko, Ailidani Ailijiang, and Murat Demirbas. PigPaxos: Devouring the communication bottlenecks in distributed consensus. In *Proceedings of ACM International Conference on Management of Data (SIGMOD)*, June 2021.

[12] Inho Cho, Ahmed Saeed, Joshua Fried, Seo Jin Park, Mohammad Alizadeh, and Adam Belay. Overload control for µs-scale RPCs with breakwater. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, November 2020.

[13] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally distributed database. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, October 2012.

[14] Heming Cui, Rui Gu, Cheng Liu, Tianyu Chen, and Junfeng Yang. Paxos made transparent. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, October 2015.

[15] Huynh Tu Dang, Pietro Bressana, Han Wang, Ki Suh Lee, Noa Zilberman, Hakim Weatherspoon, Marco Canini, Fernando Pedone, and Robert Soulé. P4xos: Consensus as a network service. *IEEE/ACM Transactions on Networking (ToN)*, 28(4), August 2020.

[16] Huynh Tu Dang, Marco Canini, Fernando Pedone, and Robert Soulé. Paxos made switch-y. *ACM SIGCOMM Computer Communication Review (CCR)*, 46(2), April 2016.

[17] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. Farm: Fast remote memory. In *Proceedings of USENIX Conference on Networked Systems Design and Implementation (NSDI)*, April 2014.

[18] Haggai Eran, Maxim Fudim, Gabi Malka, Gal Shalom, Noam Cohen, Amit Hermony, Dotan Levi, Liran Liss, and Mark Silberstein. Flexdriver: A network driver for your accelerator. In *Proceedings of ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2022.

[19] Haggai Eran, Lior Zeno, Maroun Tork, Gabi Malka, and Mark Silberstein. Nica: An infrastructure for inline acceleration of network applications. In *Proceedings of USENIX Conference on Annual Technical Conference (ATC)*, July 2019.

[20] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. Azure accelerated networking: SmartNICs in the public cloud. In *Proceedings of USENIX Conference on Networked Systems Design and Implementation (NSDI)*, April 2018.

[21] Linux Foundation. Data plane development kit (DPDK).

[22] Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. BMC: Accelerating memcached using safe in-kernel caching and pre-stack processing. In *Proceedings of USENIX Conference on Networked Systems Design and Implementation (NSDI)*, April 2021.

[23] Stewart Grant, Anil Yelam, Maxwell Bland, and Alex C. Snoeren. SmartNIC performance isolation with FairNIC: Programmable networking for the cloud. In *Proceedings of ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, July 2020.

[24] Zhenyu Guo, Chuntao Hong, Mao Yang, Dong Zhou, Lidong Zhou, and Li Zhuang. Rex: replication at the speed of multi-core. In *Proceedings of ACM European Conference on Computer Systems (EuroSys)*, April 2014.

[25] Dongxu Huang, Qi Liu, Qiu Cui, Zhuhe Fang, Xiaoyu Ma, Fei Xu, Li Shen, Liu Tang, Yuxing Zhou, Menglong Huang, Wan Wei, Cong Liu, Jian Zhang, Jianjun Li, Xuelian Wu, Lingyu Song, Ruoxi Sun, Shuaipeng Yu, Lei Zhao, Nicholas Cameron, Liquan Pei, and Xin Tang. TiDB: A Raft-based HTAP Database. *The Proceedings of the VLDB Endowment (PVLDB)*, 13(12), 2020.

[26] Patrick Hunt, Mahadev Konar, Flavio P Junqueira, and Benjamin Reed. ZooKeeper: wait-free coordination for internet-scale systems. In *Proceedings of USENIX Conference on Annual Technical Conference (ATC)*, June 2010.

[27] Stephen Ibanez, Gordon Brebner, Nick McKeown, and Noa Zilberman. The P4->NetFPGA workflow for line-rate packet processing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2019.

[28] Stephen Ibanez, Alex Mallery, Serhat Arslan, Theo Jepsen, Muhammad Shahbaz, Changhoon Kim, and Nick McKeown. The nanopu: A nanosecond network stack for datacenters. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, July 2021.

[29] Zsolt István, David Sidler, and Gustavo Alonso. Caribou: Intelligent distributed storage. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, August 2017.

[30] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. Consensus in a box: inexpensive coordination in hardware. In *Proceedings of USENIX Conference on Networked Systems Design and Implementation (NSDI)*, March 2016.

[31] Sagar Jha, Jonathan Behrens, Theo Gkountouvas, Matthew Milano, Weijia Song, Edward Tremel, Robbert Van Renesse, Sydney Zink, and Kenneth P Birman. Derecho: Fast state machine replication for cloud services. *ACM Transactions on Computer Systems (TOCS)*, 36(2), April 2019.

[32] Xin Jin, Xiaozhou Li, Haoyu Zhang, Nate Foster, Jeongkeun Lee, Robert Soulé, Changhoon Kim, and Ion Stoica. Netchain: Scale-free sub-rtt coordination. In *Proceedings of USENIX Conference on Networked Systems Design and Implementation (NSDI)*, April 2018.

[33] Xin Jin, Xiaozhou Li, Haoyu Zhang, Robert Soulé, Jeongkeun Lee, Nate Foster, Changhoon Kim, and Ion Stoica. Netcache: Balancing key-value stores with fast in-network caching. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, October 2017.

[34] Anuj Kalia, Michael Kaminsky, and David Andersen. Datacenter rpcs can be general and fast. In *Proceedings of USENIX Conference on Networked Systems Design and Implementation (NSDI)*, February 2019.

[35] Anuj Kalia, Michael Kaminsky, and David G Andersen. Fasst: Fast, scalable and simple distributed transactions with two-sided (RDMA) datagram RPCs. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, October 2016.

[36] Manos Kapritsos, Yang Wang, Vivien Quema, Allen Clement, Lorenzo Alvisi, and Mike Dahlin. All about eve: Execute-verify replication for multi-core servers. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, October 2012.

[37] Antonios Katsarakis, Vasilis Gavrielatos, M.R. Siavash Katebzadeh, Arpit Joshi, Aleksandar Dragojevic, Boris Grot, and Vijay Nagarajan. Hermes: A fast, fault-tolerant and linearizable replication protocol. In *Proceedings of ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2020.

[38] Mikhail Kazhamiaka, Babar Memon, Chathura Kankanamge, Siddhartha Sahu, Sajjad Rizvi, Bernard Wong, and Khuzaima Daudjee. Sift: resource-efficient consensus with RDMA. In *The International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, December 2019.

[39] Marios Kogias and Edouard Bugnion. Hovercraft: achieving scalability and fault-tolerance for microsecond-scale datacenter services. In *Proceedings of ACM European Conference on Computer Systems (EuroSys)*, April 2020.

[40] Leslie Lamport. Paxos made simple. *ACM SIGACT News*, 32(4), 2001.

[41] Bojie Li, Zhenyuan Ruan, Wencong Xiao, Yuanwei Lu, Yongqiang Xiong, Andrew Putnam, Enhong Chen, and Lintao Zhang. KV-Direct: High-performance in-memory key-value store with programmable nic. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, October 2017.

[42] Jialin Li, Ellis Michael, Naveen Kr Sharma, Adriana Szekeres, and Dan RK Ports. Just say no to paxos overhead: Replacing consensus with network ordering. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, October 2016.

[43] Jiaxin Lin, Kiran Patel, Brent E. Stephens, Anirudh Sivaraman, and Aditya Akella. PANIC: A high-performance programmable NIC for multi-tenant networks. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, November 2020.

[44] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading distributed applications onto SmartNICs using iPipe. In *Proceedings of ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, August 2019.

[45] Thomas Luinaud, Jeferson Santiago da Silva, J.M. Pierre Langlois, and Yvon Savaria. Design principles for packet deparsers on FPGAs. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021.

[46] Michael Marty, Marc de Kruijf, Jacob Adriaens, Christopher Alfeld, Sean Bauer, Carlo Contavalli, Michael Dalton, Nandita Dukkipati, William C. Evans, Steve Gribble, Nicholas Kidd, Roman Kononov, Gautam Kumar, Carl Mauer, Emily Musick, Lena Olson, Erik Rubow, Michael Ryan, Kevin Springborn, Paul Turner, Valas Valancius, Xi Wang, and Amin Vahdat. Snap: A microkernel approach to host networking. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, October 2019.

[47] Marvell. OCTEON TX2 LiquidIO III SmartNIC. https://www.marvell.com/products/data-processing-units.html.

[48] Jeffrey C. Mogul. TCP offload is a dumb idea whose time has come. In *Proceedings of USENIX Workshop on Hot Topics in Operating Systems (HotOS)*, May 2003.

[49] Iulian Moraru, David G Andersen, and Michael Kaminsky. There is more consensus in egalitarian parliaments. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, November 2013.

[50] NVIDIA. NVIDIA Mellanox Innova-2 Flex Open Programmable SmartNIC. https://www.nvidia.com/en-us/networking/ethernet/innova-2-flex/.

[51] Brian M Oki and Barbara H Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of ACM Symposium on Principles of Distributed Computing (PODC)*, June 1988.

[52] Diego Ongaro and John K Ousterhout. In search of an understandable consensus algorithm. In *Proceedings of USENIX Conference on Annual Technical Conference (ATC)*, June 2014.

[53] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. Floem: A programming system for nic-accelerated network applications. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, October 2018.

[54] Marius Poke and Torsten Hoefler. DARE: High-performance state machine replication on rdma networks. In *Proceedings of ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, June 2015.

[55] Waleed Reda, Marco Canini, Dejan Kostić, and Simon Peter. RDMA is turing complete, we just did not know it yet! In *Proceedings of USENIX Conference on Networked Systems Design and Implementation (NSDI)*, April 2022.

[56] Sajjad Rizvi, Bernard Wong, and Srinivasan Keshav. Canopus: A scalable and massively parallel consensus protocol. In *The International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, November 2017.

[57] Luigi Rizzo. netmap: a novel framework for fast packet i/o. In *Proceedings of USENIX Conference on Annual Technical Conference (ATC)*, June 2012.

[58] Timothy Roscoe. Keynote: It's time for operating systems to rediscover hardware. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, July 2021.

[59] Henry N. Schuh, Weihao Liang, Ming Liu, Jacob Nelson, and Arvind Krishnamurthy. Xenic: SmartNIC-accelerated distributed transactions. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, October 2021.

[60] Rajath Shashidhara, Tim Stamler, Antoine Kaufmann, and Simon Peter. FlexTOE: Flexible TCP offload with Fine-Grained parallelism. In *Proceedings of USENIX Conference on Networked Systems Design and Implementation (NSDI)*, April 2022.

[61] Arjun Singhvi, Aditya Akella, Maggie Anderson, Rob Cauble, Harshad Deshmukh, Dan Gibson, Milo MK Martin, Amanda Strominger, Thomas F Wenisch, and Amin Vahdat. CliqueMap: productionizing an RMA-based distributed caching system. In *Proceedings of ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM)*, August 2021.

[62] Adriana Szekeres, Michael Whittaker, Jialin Li, Naveen Kr. Sharma, Arvind Krishnamurthy, Dan R. K. Ports, and Irene Zhang. Meerkat: multicore-scalable replicated transactions following the zero-coordination principle. In *Proceedings of ACM European Conference on Computer Systems (EuroSys)*, April 2020.

[63] Rebecca Taft, Irfan Sharif, Andrei Matei, Nathan VanBenschoten, Jordan Lewis, Tobias Grieger, Kai Niemi, Andy Woods, Anne Birzin, Raphael Poss, et al. CockroachDB: The resilient geo-distributed sql database. In *Proceedings of ACM International Conference on Management of Data (SIGMOD)*, June 2020.

[64] Robbert Van Renesse, Nicolas Schiper, and Fred B Schneider. Vive la différence: Paxos vs. viewstamped replication vs. zab. *IEEE Transactions on Dependable and Secure Computing*, 12(4), 2014.

[65] Cheng Wang, Jianyu Jiang, Xusheng Chen, Ning Yi, and Heming Cui. APUS: Fast and scalable Paxos on RDMA. In *Proceedings of ACM Symposium on Cloud Computing (SoCC)*, September 2017.

[66] Han Wang, Robert Soulé, Huynh Tu Dang, Ki Suh Lee, Vishal Shrivastav, Nate Foster, and Hakim Weatherspoon. P4FPGA: A rapid prototyping framework for p4. In *ACM Symposium on SDN Research (SOSR)*, 2017.

[67] Zhaoguo Wang, Changgeng Zhao, Shuai Mu, Haibo Chen, and Jinyang Li. On the parallels between paxos and raft, and how to port optimizations. In *Proceedings of ACM Symposium on Principles of Distributed Computing (PODC)*, July 2019.

[68] Irene Zhang, Amanda Raybuck, Pratyush Patel, Kirk Olynyk, Jacob Nelson, Omar S Navarro Leija, Ashlie Martinez, Jing Liu, Anna Kornfeld Simpson, Sujay Jayakar, et al. The demikernel datapath os architecture for microsecond-scale datacenter systems. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, October 2021.

[69] Siyuan Zhou and Shuai Mu. Fault-tolerant replication with pull-based consensus in MongoDB. In *Proceedings of USENIX Conference on Networked Systems Design and Implementation (NSDI)*, April 2021.

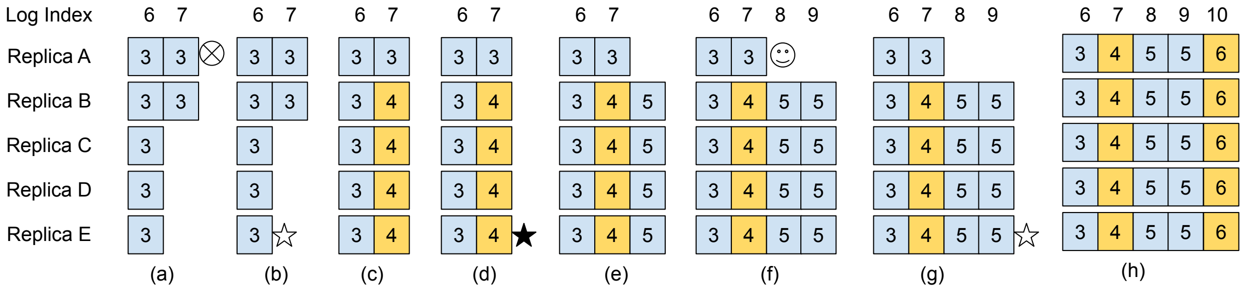

Figure 13: Waverunner Operation. ☆ represents an election that disables hardware acceleration; ★ represents an election that enables the hardware acceleration; ⊗ represents a fail stop and ☺ a recovery. The numbers inside the boxes refer to the `term` numbers in each log entry. The blue boxes refer to regular log entries; the yellow boxes refer to empty `noop` log entries.

A Correctness

Here we discuss the correctness of our approach. One reason we choose to implement Raft instead of inventing a consensus protocol is that Raft is widely used and proved correct.[3] Using Raft can help us avoid having any errors in inventing a new protocol, which is known to be an error-prone process.

We show that with or without the hardware acceleration, including the transition, the system follows Raft protocol.

Fact 1. When the hardware acceleration is off at a replica, the replica follows the Raft protocol.

Therefore, if hardware acceleration at all replicas is off, the system design is a standard Raft and it is correct.

Lemma 2. When the hardware acceleration is on (and during the process it is switched on) at a replica, the replica follows the Raft protocol.

This is the principle throughout the system design. The hardware part is designed to switch back to software whenever it sees a message that it is not expecting. Not responding to that particular message is not a behavior that violates Raft's safety because Raft's original assumption is that the network is asynchronous and messages could be lost. Therefore, the replica as a whole (both hardware and software) is still following the Raft protocol, except that it requests an election. In Raft (and other consensus protocols), doing an election is always safe.

As a replica follows the Raft protocol regardless of whether the hardware acceleration is on or off (or during transition), the system is a Raft and thus correct.

[3] By "correct" we mean the system has both safety and liveness. Because Raft has already proved on these, we will use "correct" to refer to our system is either a standard Raft or is equivalent to it.

B An Example of Waverunner Operations

Figure 13 walks through an example of Waverunner failure recovery with five replicas A,B,C,D, and E. The numbers in the boxes refer to the `term` numbers in each log entry.

(a) Replica A is the leader, using hardware acceleration to replicate log entries to followers until it stops.
(b) Replica E is first to detect a lack of new messages from leader A. E disables hardware acceleration and triggers a leader election, which it wins (becoming the new leader) after receiving votes from C and D.
(c) Replica E commits a `noop`, indicated in yellow, to all replicas except A (which remains unavailable). Note that a log entry in replica B is overwritten because it was ahead of the new leader. This operation is safe because the log entry was not committed.
(d) Replica E starts a round of `RequestVote2FPGA`, enabling hardware acceleration on all replicas.
(e) Replica E operates as the leader, replicating log entries using the hardware accelerator.
(f) Replica A recovers, immediately observing new `AppendEntry` messages arriving from leader E. Replica A reports a mismatch with its existing logs by rejecting the new entries.
(g) Having learned of the mismatch on replica A from the rejection message, replica E disables hardware acceleration with another leader election.
(h) Replica E then commits another `noop` and sends the missing log entries to replica A. In this process, the mismatched logs on replica A are also overwritten.

```
Variables shared by hardware and software:
 Used only by leader:
  matchIndex[] for each follower, index of highest
               log entry known to be replicated,
               initialized to 0, increases
               monotonically
  commitIndex  index of highest log entry known to
               be committed, initialized to 0
 Used by all replicas:
  id           a globally unique integer that
               identifes the server
  isLeader     hint that suggests whether the
               server is leader
  currentTerm  latest term server has seen,
               initialized to 0
  lastLogIndex index of the last log entry, is a
               sequentially increasing counter,
               initialized to 0
  lastLogTerm  term of the last log entry

Variables in host software:
 Used only by leader:
  nextIndex[]  for each server, index of the next
               log entry to send to that server,
               initialized to leader's
               lastLogIndex+1
 Used by all replicas:
  votedFor     candidateId that received vote in
               current term (or null if none)
  log[]        log entries; each entry contains
               command for state machine, and term
               when entry was received by leader
               (first index is 1)
  lastApplied  index of highest log entry applied
               to state machine (initialized to 0,
               increases monotonically)
```

Figure 14: Variables in Hardware and Software.

C Hardware and Software Variables

Figure 14 presents the complete set of Raft protocol variables that are used by hardware and software.