RDMA is Turing complete, we just did not know it yet!

Waleed Reda
Université catholique de Louvain
KTH Royal Institute of Technology

Marco Canini
KAUST

Dejan Kostić
KTH Royal Institute of Technology

Simon Peter
UT Austin

Abstract

It is becoming increasingly popular for distributed systems to exploit network offload to alleviate load on the CPU. Remote Direct Memory Access (RDMA) NICs (RNICs) are one such device, allowing offload of remote memory access. However, RDMA still requires CPU intervention for complex offloads that go beyond simple remote memory access. As such, the offload potential for RNICs is limited and RDMA-based systems usually have to work around such limitations.

We present RedN, a principled, practical approach to implementing complex RNIC offloads, without requiring any hardware modifications. Using *self-modifying* RDMA chains, we lift the existing RDMA verbs interface to a Turing complete set of programming abstractions. We explore what is possible in terms of offload complexity and performance with just a commodity RNIC. Through a key-value store use case study, we show how to integrate complex RNIC offloads into existing applications. RedN can outperform one and two-sided RDMA implementations by up to 3× and 7.8× for key-value get operations and performance isolation, respectively, and provide applications with failure resiliency to OS and process crashes.

1 Introduction

As server CPUs become an increasingly scarce resource, NIC offload is gaining in popularity [25, 30, 32–34, 38]. System operators wish to reserve CPUs for application execution, while common, often-repeated operations may be offloaded to the NIC. NIC offloads, in particular, have the benefit that they reside in the network data path and NICs can carry out operations on in-flight data with low latency [33].

For this reason, remote direct memory access (RDMA) [16] has become ubiquitous [21]. Mellanox ConnectX NICs [5] have pioneered ubiquitous RDMA support and Intel has added RDMA support to the 800 series of Ethernet network adapters [8]. RDMA focuses on the offload of simple message passing (via SEND/RECV verbs) and remote memory access (via READ/WRITE verbs) primitives [16]. Both primitives are widely used in networked applications and their offload is extremely useful. However, RDMA is not designed for more complex offloads that are also common in networked applications. Operations, such as remote data structure traversal and hash table accesses, are not normally deemed realizable with RDMA [41]. This led to many RDMA-based systems to require multiple message round-trips or to reintroduce involvement of the server's CPU to execute requests [19, 24, 28, 29, 37, 39, 43].

To support complex offloads, the networking community has developed a number of SmartNIC architectures [3, 4, 12, 15, 18]. SmartNICs incorporate more powerful compute capabilities via CPUs or FPGAs. They can execute arbitrary programs on the NIC, enabling complex offloads. However, these SmartNICs are not ubiquitous and their smaller volume implies a higher cost. SmartNICs can cost up to 5.7× more than commodity RDMA NICs (RNICs) at the same link speed (§2.1). Due to their custom architecture, they are also a management burden to the system operator, who has to support SmartNICs apart from the rest of the fleet.

We ask whether we can avoid this tradeoff and attempt to use the ubiquitous RNICs to realize complex offloads. To do so, we have to solve a number of challenges. First, we have to answer if and how we can use the RNIC interface, which consists only of simple data movement verbs (READ, WRITE, SEND, RECV, etc.) and no conditional or looping constructs, to realize complex offloads. More than that, our solution has to be general so that offload developers can use it to build *RDMA chains* — sequences of RDMA verbs — that can perform a wide range of offloads. Second, we have to ensure that our solution is efficient and that we understand the performance and performance variability properties of using RNICs for complex offloads. Finally, we have to answer how complex RNIC offloads integrate with existing applications.

In this paper, we show that it is indeed possible to use RNICs to implement complex offloads. To do so, we implement conditional branching via *self-modifying* RDMA verbs. Clever use of the existing compare-and-swap (CAS) verb enables us to dynamically modify the RNIC execution path by editing subsequent verbs in an RDMA chain, using the

CAS operands as a predicate. Just like self-modifying code executing on CPUs, self-modifying verbs require careful control of the execution path to avoid consistency issues due to RNIC verb prefetching. To do so, we rely on special WAIT and ENABLE RDMA verbs [30, 36], which add an execution dependency between operations. WAIT allows us to trigger operations based on the completions of others, providing strict ordering between RDMA verbs. By controlling verb prefetching, ENABLE enforces consistency for verbs modified by preceding operations. WAIT also allows us to create loops by re-triggering earlier, already executed verbs in an RDMA work queue.

Based on these primitives, we present RedN, a principled, practical approach to implementing complex RNIC offloads. Using self-modifying RDMA chains, we develop a number of building blocks, called *gadgets*, that lift the existing RDMA verbs interface to a Turing complete set of programming abstractions. Using these gadgets, we explore what is possible in terms of offload complexity and performance with just a commodity RNIC. Through a use case study, we show how to integrate complex RNIC offloads, developed with RedN principles, into existing networked applications. RedN affords offload developers a practical way to implement complex NIC offloads on commodity RNICs, without the burden of acquiring and maintaining SmartNICs.

We make the following contributions:

- We present RedN, a principled, practical approach to offloading arbitrary computation to RDMA NICs. RedN leverages RDMA ordering and compare-and-swap primitives to build conditionals and loops.
- Using RedN, we present and evaluate the implementation of various offloads that are useful in common server computing scenarios. In particular, we implement hash table lookup with Hopscotch hashing and linked list traversal.
- We evaluate the complexity and performance of offload in a number of use cases with the Memcached key-value store. In particular, we evaluate offload of common-case key-value get operations, performance isolation, and failure resiliency. We demonstrate that RNIC offload with RedN can realize all of these use-cases. It can outperform state-of-the-art one and two-sided RDMA implementations [24, 37] for get operations by 3×, improve performance under contention by 7.8×, and improve availability under different failure scenarios.

2 Background

RDMA was conceived as a fast and reliable interconnect for high-performance computing (HPC) clusters, but has grown out of this niche [21]. It is becoming ever-more popular due to the growth in network bandwidth, with stagnating growth in CPU performance, making CPUs an increasingly scarce resource that is best reserved to running application code. With RNICs now considered commodity, it is opportunistic to explore the different use-cases where their hardware can yield benefits. These efforts, however, have been hampered by the slow evolution of the RDMA API, which has not sufficiently adapted to the new, broader uses of RDMA. Consequently, the networking community has built SmartNICs using FPGAs or CPUs to evaluate new complex offloads.

2.1 SmartNICs

To allow for complex network offloads, SmartNICs have been developed [2, 3, 11, 12]. SmartNICs include dedicated computing units or FPGAs, memory, and several dedicated accelerators, such as cryptography engines. For example, Mellanox BlueField [12] has 8× ARMv8 cores with 16GB of memory and 2×25GbE ports. These SmartNICs are able to run full operating systems, but also ship with lightweight runtime systems that can provide kernel-bypass access to the NIC's IO engines.

Related work on SmartNIC offloads. SmartNICs have been used to offload complex tasks to the NIC. For example, StRoM [41] uses an FPGA NIC to implement RDMA verbs and creates generic kernels (or building blocks) that perform various functions, such as traversing linked lists. KV-Direct [32] uses an FPGA Programmable NIC to accelerate key-value accesses. iPipe [33] and Floem [38] are programming frameworks that simplify complex offload development for primarily CPU-based SmartNICs. E3 [34] transparently offloads microservices to SmartNICs.

The cost of SmartNICs. While SmartNICs provide the capabilities for complex offloads, they come at a cost. For example, a dual-port 25GbE BlueField SmartNIC at $2,340 costs 5.7× more than the same-speed ConnectX-5 RNIC at $410 (cf. [14]). Another cost is the special management required for SmartNICs. SmartNICs are a special piece of complex equipment that system administrators need to understand and maintain. SmartNIC operating systems and runtimes can crash, have security flaws, and need to be kept up-to-date with the latest vendor patches. This is an additional maintenance burden on operators that is not incurred for RNICs.

2.2 RDMA NICs

RDMA NICs (or RNICs) processing power has doubled with each subsequent generation. This allows RNICs to cope with higher packet rates and more complex, hard-coded offloads (e.g., reduction operations, encryption, erasure coding, etc.).

We measure the highest achievable verb throughput over several generations of Mellanox ConnectX NICs, using the Mellanox `ib_write_bw` benchmark. This benchmark performs 64B RDMA writes, and it is not network bandwidth-limited due to the small RDMA write size. We find that the verb processing throughput doubles with each generation, as we can see in Table 1. This is primarily due to a doubling in processing units (PUs) in each generation.[1] As a result, ConnectX-6 NICs can execute up to 110 million RDMA operations per second with a single NIC port. This increased

[1] Private discussions with Mellanox affirmed our findings.

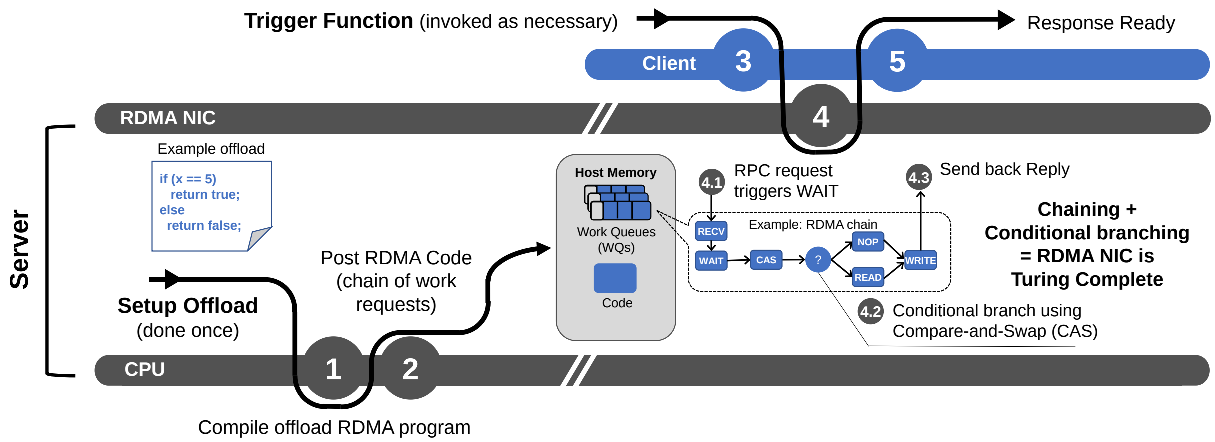


Figure 1: RDMA NICs can be turned into Turing machines, without any hardware modifications, if we simply allow conditional branches to be expressed. Conditional branching can be implemented by using RDMA Compare-and-Swap (CAS) verbs to self-modify subsequent RDMA operations in the chain.

hardware performance further motivates the need for exploiting the computational power of these devices.

Related work on RDMA offload. RDMA has been employed in many different contexts over the years—including accelerating key-value stores and filesystems [20, 24, 28, 37, 46], consensus [19, 29, 39, 43], distributed locking [47], and even more nuanced use-cases such as efficient access in distributed tree-based indexing structures [48]. While these works use RDMA for performance, they do so within the confines of RDMA's intended use as a data movement offload. When complex functionality is required, these systems involve multiple RDMA round-trips and/or rely on host CPUs to carry out these complex operations.

Hyperloop [30] focuses on improving the offload capabilities for RNICs. Their approach relies on combining regular RDMA verbs to implement commonly-used storage operations such as chain replication. However, their framework does not provide a blueprint for offloading arbitrary computations and cannot offload functions that use any type of conditional logic (e.g., walking a remote hashtable). Moreover, the Hyperloop protocol is likely incompatible with next generation Mellanox RDMA NICs, as its implementation relies on modifying WR ownership—a feature that has been deprecated for ConnectX-4 and newer cards.

Unlike this previous body of work, we aim to leverage a combination of existing RDMA verbs in novel ways to unlock the computational power of latest generation RNICs and provide an unprecedented level of programmability for complex offloads (§3).

RNIC	PUs	Throughput
ConnectX-3 (2014)	2	15 Mops/s
ConnectX-5 (2016)	8	63 Mops/s
ConnectX-6 (2017)	16	112 Mops/s

Table 1: Number of Processing Units (PUs) and performance of various ConnectX generations.

3 The RedN Computational Framework

We develop a principled framework that enables complex offloads, called RedN. RedN's key idea is to combine widely available capabilities of RNICs to enable a restricted form of self-modifying RDMA programs. These programs — chains of RDMA operations — are capable of executing dynamic control flows with conditional and looping constructs.

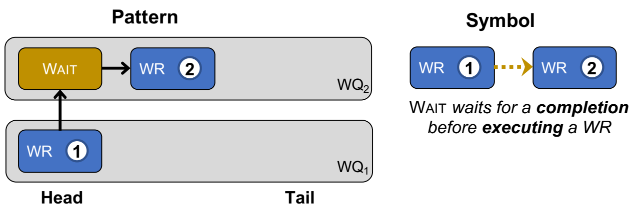
Fig. 1 depicts an illustration of the envisioned usage of RedN. The setup phase involves creating and posting the RDMA code of the offload. Clients can then use the offload by invoking a trigger that sets RNIC execution to follow the posted RDMA chain.

To understand our framework, we first look into the execution models offered by RDMA NICs, and the ordering guarantees they provide for RDMA operations. We then look into the expressivity of traditional RDMA verbs and explore parallels with modern CPU instruction sets. We use these insights to describe strategies for expressing complex logic using traditional RDMA verbs, *without requiring any hardware modifications.*

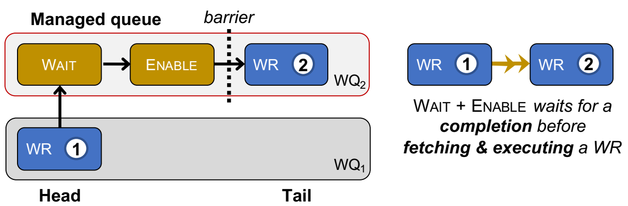
3.1 RDMA execution model

The RDMA interface specifies a number of data movement verbs (READ, WRITE, SEND, RECV, etc.) that are *posted* as *work requests* (WRs) by offload developers into *work queues* (WQs) in server memory. The RNIC starts execution of a sequence of WRs in a WQ once the offload developer triggers a *doorbell*—a special register in RNIC memory that informs the RNIC that a WQ has been updated and should be executed.

Work request consistency. Ordering rules for RDMA WRs differentiate between write WRs and operations that return a value. Within each category of operations, RDMA guarantees in-order execution of all these WRs within a single WQ. In particular, write WRs (i.e., SEND, WRITE, WRITEIMM) are totally ordered with each other. However, the RDMA standard allows a write to be reordered before prior non-write WRs, i.e., writes-follows-reads consistency is not guaranteed.



(a) Completion consistency.



(b) Doorbell consistency.

Figure 2: Work request consistency levels that guarantee a total order of operations (a,b) and coherence with the host-memory-resident WR (b). The symbols on the right will be used as notation for these WR chains in the examples of §3.

We call the default RDMA ordering model *work queue (WQ) consistency*. To support sophisticated offload logic, we require stronger consistency models, which we construct with the help of two RDMA verbs. Fig. 2 shows two stricter consistency models that we introduce and how to achieve them.

The WAIT verb stops WR execution until the completion of a specified WR from another WQ or the preceding WR in the same WQ. We call this *completion consistency* (Fig. 2a). It achieves total ordering of WRs along the execution chain (which potentially involves multiple WQs). It can be used to enforce data consistency, similar to data memory barriers in CPU instruction sets—to wait for data to be available before executing the WRs operating on the data. Moreover, WAIT allows developers to *pre-post* chains of RDMA verbs to the RNIC, without immediately executing them.

The RNIC is free to prefetch into its cache the WRs within a WQ. Thus, the execution outcome reflects the WRs at the time they were fetched, which can be incoherent with the versions that reside in host memory in case these were later modified. To avoid this issue, the RNIC allows placing a WQ into *managed* mode, in which WR prefetch is disabled. The ENABLE verb is used to explicitly start the prefetching of WRs. This allows for further WRs to be posted or existing WRs to be modified within the WQ, as long as they occur after the ENABLE in the WQ chain , and as long as this is done before completion of the posted ENABLE—similar to an instruction barrier . We achieve a full (data and instruction) barrier, by using WAIT and ENABLE in sequence. We call this *doorbell consistency* (Fig. 2b). Doorbell consistency allows developers to modify WR chains in-place. In particular, it allows for *data-dependent, self-modifying* WRs.

Thus, we can control WR fetch and execution via special verbs and WQ modes, resulting in different code and data consistency levels. These verbs are widely available in commodity RNICs but are not standard (e.g., Mellanox terms them cross-channel communication [36]).

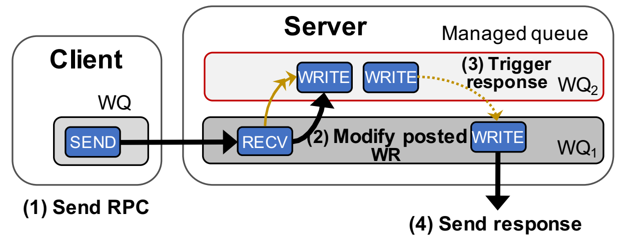


Figure 3: Clients can trigger posted operations. Thick solid lines represent (meta)data movements.

3.2 Dynamic RDMA Programs

While a static sequence of RDMA WRs is already a rudimentary RDMA program, complex offloads require *data-dependent execution* — where the logic of the offload is influenced directly by input arguments. To do so, we construct *self-modifying RDMA code*.

Self-modifying RDMA code. Doorbell ordering enables a restricted form of self-modifying code capable of data-dependent execution. To illustrate this concept, we use the example of a server that offloads an RPC handler to its RNIC as shown in Fig. 3. Importantly, the RPC response depends on some argument set by the client and thus the RDMA offload is data-dependent. The server posts the RDMA program that consists of a set of WRs spanning two WQs. The client invokes the offload by issuing a SEND operation. At the RNIC, the SEND triggers the posted RECV operation.

Observe that RECV specifies where the SEND data is placed. We configure RECV to inject the received data into the posted WR chain to modify their attributes. We achieve this by leveraging doorbell ordering, to ensure that posted WRs are not prefetched by the RNIC and can be altered by preceding WRs.

This is an instance of self-modifying code. As such, clients can pass arguments to the offloaded RPC handler and the RNIC will dynamically alter the executed code accordingly. To be precise, this requires that clients use a pre-established format for the argument; we envision that the low-level details of the mechanisms throughout our framework are automatically handled by the offload library through traditional compilation techniques. However, this, by itself is not sufficient to provide a Turing complete offload framework.

Turing-completeness of RDMA. Turing-completeness is a universally accepted benchmark that determines the expressive power of any programming language or instruction set. For a programming language to be Turing-complete, we need to satisfy requirements [27]:
R1: Ability to read/write arbitrary amounts of memory.
R2: Conditional branching (e.g. if/else statements).

The former can be mostly satisfied with regular RDMA

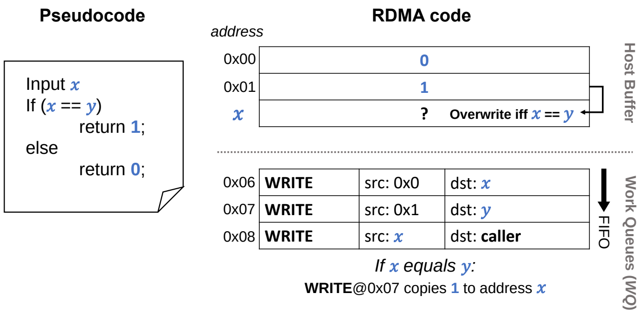


Figure 4: RDMA copy verbs can be used to emulate a simple *if* statement by copies to addresses x and y, we can use overwrites to detect equalities.

verbs, whereas the latter has never been demonstrated with RDMA NICs. Moreover, to be able to truly access an *arbitrary* amount of memory, we essentially need a way of realizing loops. This is also practical from an offload perspective, as it opens up potential use-cases and lowers the number of constraints that programmers have to consider for offloads. As such, to make it more explicit, we add a third requirement: **R3:** The ability to execute code repeatedly (loops).

In the next sub-sections, we show how dynamic execution can be used to satisfy all the aforementioned requirements. A proof sketch of Turing completeness is given in Appendix A.

3.3 Basic Conditionals

Conditional execution branching—choosing which computation to perform based on a runtime condition—is typically assembled using jump instructions, which are not readily available in RDMA. Inspired by Dolan [22], we show that simple data movements are sufficient to do comparisons. In particular, all it takes is a chain of appropriately posted WRITEs.

Consider testing for equality of two values. Figure 4 illustrates the example of checking if $x = y$. We issue two successive WRITEs, treating the values of the operands as two destination addresses, one for each write. What is written is the effect of the two branches. Interestingly, if operands x and y are equal, the second write overwrites the first. Thus, by arranging to associate the effect of when the condition is false with the first write and the effect of when it is true with the second write, we ensure that address x holds the outcome of the evaluated conditional after the two writes. The example uses a third WRITE to return the outcome to the client.

The above constructs achieve a primitive kind of conditional execution in which both branches are "executed." This is immediately applicable to simple conditional logic but is inefficient as it executes operations in both branches. To efficiently branch execution, it is imperative to have the capability of modifying the program's control flow and choosing the "right" branch. We provide a novel way to implement conditionals using compare-and-swap CAS RDMA verbs. We introduce this in the next section, using a *while* loop example, to elaborate on the usefulness of branching, or being able to modify a program's control flow.

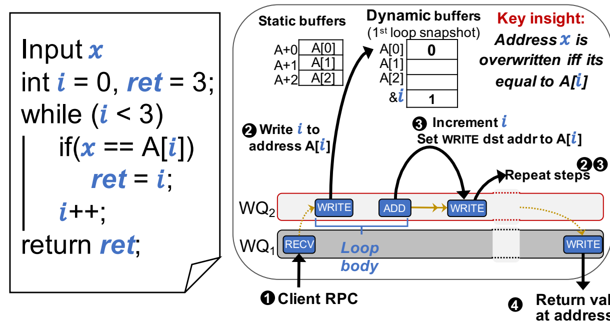


Figure 5: *while* loop toy example.

3.4 Loops

Using a static loop, we highlight that supporting loop constructs requires (1) conditional branching (to test the loop condition) and (2) WR re-execution (to repeat the loop body).
Static loop. In the Figure 5 example, offload searches for x in an array A and returns the corresponding index. The loop is static because A has finite size (3), known a priori. To simplify presentation, assume the contrived case with $A[i] = i, \forall i$. Without this simplification, the example would include an additional WRITE to fetch the value at $A[i]$. The loop body uses the ADD verb to increment i and the WRITE verb to test for equality (as in §3.3). Ideally, the loop could terminate as soon as $x = A[i]$. However, without a conditional branching, the example must repeat the loop body 3 times, as many iterations as the size of A. This is possible by posting to the NIC as many copies of the loop body. However, to repeat the loop body for a dynamic number of iterations, we require both a way to perform conditional branching and a loop re-execution mechanism. We develop each, in turn, below.

3.4.1 Conditional branching using CAS.

Figure 6 modifies the previous example with a break that exits the loop if the element is found. The loop remains static and its body is posted to the NIC in 3 copies. The role of break is to *jump* out of the loop. To do so, we develop another way to do conditional branching, using the compare-and-swap CAS RDMA verb. Recall that CAS atomically compares the value at a memory address with an *old* value and if they are equal, stores a *new* value at that address.

In more detail, for the conditional test, we inline x (the first if operand) in CAS's *old* attribute. Fetched via a preceding WRITE, we inline $A[i]$ (the second if operand) as the *ID* attribute of NOOP. CAS addresses the location of the NOOP verb and *new* is the READ opcode. When the condition $x = A[i]$ holds, CAS modifies the *opcode* attribute since it is contiguous to the *ID* attribute. Thus, this requires a single CAS. To invalidate the remaining WRs, the posted NOOP is formatted such that once it is transformed into a READ, it addresses (as a scatter read) the opcode field of subsequent WRs and moves a zero (the NOOP opcode) into those opcodes. This is how we perform a break and modify the control flow of the program and jump out of the loop.

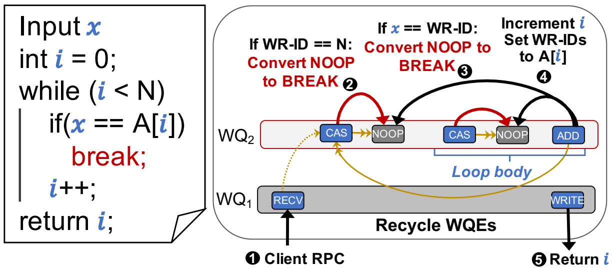


Figure 6: ***while*** **loops with (*N*) iterations.**

To continue execution after break, a WAIT verb in another WQ monitors the WQ where the loop executes. Thus, as desired, the execution jumps to the other WQ once either the loop terminates or the loop breaks out. Using CAS to implement conditionals allows us to modify the control flow of the program and branch. As opposed to the approach described in Figure 5, it also does not require us to treat the value of operands as addresses and, instead, compares the values and performs a branch using the CAS verb.

3.4.2 Dynamic loops via WQ Recycling.

Next, we modify Figure 6 to have the loop iterate N times, where N is not known a priori. To support this, we make use of a novel technique which we denote as *WQ Recycling*. RNICs execute code differently from CPUs: the RNIC iterates over WQs, which are circular buffers, and executes the WRs therein. Thus, by design each WR is meant to only execute once and there is no native functionality to treat a chain of WRs as a function that may be invoked multiple times. However, there is no fundamental reason why WRs cannot be reused. The RNIC does not erase them from the WQ. To enable recycling of a WR chain, we use doorbell consistency within the same WQ so that the WQ's head wraps around the tail and the RNIC refetches the WR chain.

The WAIT and ENABLE sequence is inserted at the end of the loop body. To stop the loop, we rewrite the loop as a while true loop with a break construct conditioned by $i = N$. This is handled analogously to the case in the previous example.

3.5 Putting it all together

With conditional branching, we can dynamically alter the control flow of any function on an RNIC. Loops allow us to traverse arbitrary data structures. Together, we have transformed an RNIC into a general processing unit. In this section, we discuss the usability aspects from overhead, security, programmability, and expressiveness perspectives.

Gadget API. We abstract and parameterize the RDMA chains required for conditional branching and looping into if and while *gadgets*. The overhead in terms of RDMA WR chains of our gadgets is shown in Table 2. We can see a breakdown of the minimum number of operations required for each.

We introduce two different variations of our gadgets — namely, *Copy* and *CAS*. The former uses only regular copy

Offload gadgets		Number of WRs		Operand limit [bits]	
		Copy	CAS	Copy	CAS
if	$==, \neq$	$3+1E$	$2+1A+2E$	$\log_2(M)$	24
	$>, \geq, <, \leq$	$4+1E$	$3+1A+2E$		
while	$==, \neq$	$3+1E$	$2+1A+2E$		
	$>, \geq, <, \leq$	$4+1E$	$3+1A+2E$		

Table 2: Breakdown of the overhead of our gadgets with different offload strategies. *A* refers to atomic RDMA operations. *E* refers to WAIT +ENABLE operations. *M* refers to the size of the registered memory region (in bytes).

verbs to implement conditionals (as introduced in Fig. 4) and the latter uses *CAS* to modify the execution path of an RDMA chain (Fig. 6).

We support arithmetic comparison operations, such as $<$ or $>$, by combining equality checks with MAX or MIN, as seen later in Table 3. However, their availability is vendor-specific, but supported in ConnectX RNICs.

Operand limits. Table 2 also shows the operand size limits, noting that Copy-based approaches require the user to register larger areas of memory, as the operands are encoded in the address fields. To support operand sizes up to 32 bits, 4 GBs of memory need to be registered. One caveat is that, in modern Linux systems, the starting address of a memory region cannot necessarily be controlled by the client and/or server [1]. As such, to avoid restricting the values to be greater than the starting address, clients can simply treat the operands as offsets and add them to the memory region's starting address.

For CAS-based approaches, the limit is based on the supported size for the CAS verb, which is 64 bits. The operand is provided as a 24-bit value, encoded in the ID and opmod fields of the WR. The remainder is used for modifying the opcode of the WR depending on the result of the comparison. We note that our advertised limits only signify what is possible with the the number of operations we allocate for our gadgets. For instance, despite the 24-bit operand limit for CAS-based gadgets, we can chain together multiple CAS operations to handle different segments of a larger operand. As such, there is no fundamental limitation — only a performance penalty.

Offload setup. To offload an RDMA program, clients first create an RDMA connection to the target server and send an RPC to initiate the offload. We envision that the server already has the offload code; however, other ways of deploying the offload can be imagined.

Upon receiving a connection request, the server creates one or more managed local WQs to post the offloaded code. Next, it registers two main types of memory regions for RDMA access: (a) a code region, and (b) a data region. The code region is the set of remote RDMA WQs created on the server, which are unique to each client and need to be accessible via RDMA to allow self-modifying code. The data region holds any data elements used by the offload (e.g., a hash table). Data regions can be shared or private, depending on the use-case.

Security aspects. RedN is a generic offload framework and, as such, does not set out to solve security challenges in existing RDMA or Infiniband implementations [42]. However, we posit the claim that, if anything, RedN can help RDMA systems become more secure.[2] For such systems, *one-sided* RDMA operations (e.g., RDMA READ and WRITE) are frequently used [24, 30, 35, 37, 44, 45] as they avoid CPU overheads at the responder. However, doing so requires clients to have direct read and/or write memory access. This can compromise security if clients are buggy and/or malicious. To give an example, FaRM allows clients to write messages directly to shared RPC buffers. This requires clients to behave correctly, as they could otherwise overwrite or modify other clients' RPCs. RedN allows applications to use *two-sided* RDMA operations (e.g., SEND and RECV), which do not require direct memory access, while *still* fully bypassing server CPUs. As we demonstrate in all our use-cases in §5, SEND operations can be used to trigger offload programs without any CPU involvement.

RedN treats client remote WQs (i.e., its code region) similar to its local WQs. They cannot be accessed by other clients and are protected by memory keys — special tokens required for RDMA access — upon registration (at connection time).

Correctness. For Copy-based gadgets, performing comparisons requires using operands as memory addresses and can potentially clobber application data. To mitigate this type of memory corruption, we restrict comparison operations to a *scratch* memory region (separate from data and code regions, each of which are protected by memory keys). If more than one gadget uses the scratch region, a barrier is inserted between them to execute the gadgets in sequence (and avoid cross-gadget interferences). The overheads of using barriers are evaluated in §5.1.2.

In addition, if operands used as RDMA addresses point outside their registered memory regions (e.g., due to a buggy client), no response will be given since the RNIC simply fails WRs with invalid addresses. The NIC will also produce a completion failure message for the failed WR and add it to a completion queue. These can be queried by the client via an RPC or by directly reading these events via one-sided RDMA verbs (if the completion queue is registered for RDMA access). The completion messages encode details of the failure event.

Fairness & isolation. Given that RedN implements dynamic loops on RDMA, clients can (ab)use such gadgets to consume more than their fair share of resources. Luckily, popular RNIC models like ConnectX provide WQ rate-limiters [7] for performance isolation. As such, even if clients trigger non-terminating offload code, they still have to adhere to their assigned rates. Moreover, offloaded code can be configured by the servers to be auditable through completion events — created automatically after a WR is executed. These events can be periodically monitored and servers can terminate connections to clients running misbehaving code.

[2] A full consideration of secure RDMA offloads, including how to formally verify provably secure offloads, is out of scope.

4 Implementation

Our offload framework is implemented in C with ∼2,300 lines of code — this includes our gadgets (∼1400), and convenience wrappers for RDMA verbs (*libibverbs*) API (∼900).

Our approach does not require modifying any RDMA libraries or drivers. RedN uses low-level functions provided by Mellanox's ConnectX driver *libmlx5* to expose in-memory WQ buffers and register them to the RDMA NIC, allowing WRs to be manipulated using other RDMA operations. We configure the ConnectX-5 firmware by modifying configurations registers [13] to allow reuse of WR IDs — which makes it possible to recycle WRs. We disable WR signatures—used for verifying WR integrity— to allow the NIC to process self-modifying WRs. WR signatures can be used to detect data corruption due to unintended stray writes from the RDMA userspace driver. However, memory protection keys can be used to help mitigate data corruption [23]. RedN is compatible with any ConnectX NICs that support WAIT and ENABLE primitives – including ConnectX-3 cards and later.

5 Evaluation

We first characterize the underlying RDMA performance (§5.1) to understand how it affects our implemented *gadgets*. Then, in our evaluation against state-of-the-art RDMA and SmartNIC offloads, we show that RedN:

1. Speeds up remote data structure traversals, such as hash tables (§5.2) and linked lists (§5.3) compared to the state-of-the-art;
2. Accelerates (§5.4) and provides performance isolation (§5.5) for the Memcached key-value store;
3. Provides improved availability for applications (§5.6) — allowing them to run in spite of OS & process crashes;
4. Exposes *gadgets* that are generic enough to enable a wide-variety of use-cases (§5.2–§5.6);

Our experimental setup is as follows.

Testbed. Our experimental testbed consists of 3x dual-socket Haswell servers running at 3.20GHz, with a total of 16 cores and 128 GB of DRAM, and 100 Gbps Dual-Port ConnectX-5 Mellanox Infiniband NICs. All nodes are running on Ubuntu 18.04 with Linux Kernel version 4.15 and are connected via back-to-back Infiniband links.

NIC setup. For all our experiments, we use Reliable Connection (RC) RDMA transport — which supports the RDMA synchronization features we use. All WQs that enforce Doorbell order are initialized with a special "managed" flag — to disable the driver from issuing doorbells after a WR is posted. The WQ size — which is configurable — is set to match the offloaded program.

5.1 Microbenchmarks

We run microbenchmarks to breakdown the RDMA latency components, understand the overheads of our different consistency models, and throughput limits of RDMA verbs.

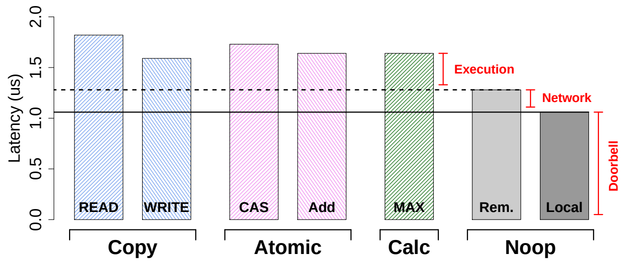


Figure 7: Latencies of different types of RDMA verbs. The solid line marks the latency of ringing the doorbell via MMIO. Difference between dashed and solid lines estimates network latency.

5.1.1 RDMA latency

We break down the performance of RDMA verbs, configured to perform 64B IO. As seen in Fig. 7, WRITE has a low latency of 1.6 µs, as it uses posted PCIe transactions, which are one-way. Comparatively, non-posted verbs such as READ or atomics[3] need to wait for a PCIe completion and take ∼1.8 µs. The execution time differences is overall small among verbs, even for more advanced, vendor-specific *Calc* verbs that perform logical and arithmetic computations (e.g.,MAX).

To break down the different latency components for RDMA, we first estimate the latency of issuing a doorbell and copying the WR to the RNIC. This can be done by measuring the execution time of a NOOP WR. This time can be subtracted from the latencies of other WRs to give an estimate of their execution time once the WR is available in the RNIC's cache. We also quantify the network cost by executing remote and loopback WRs and measuring the difference—roughly 0.25 µs for our back-to-back connected nodes.

Overall these results show low execution latency, which justifies building more sophisticated functions atop. We next measure the implications of ordering for offloads.

5.1.2 Consistency overheads

We show the latency of executing chains of RDMA operations using different consistency levels. All the posted WRs within a chain are NOOP, to allow us to isolate the performance impact of consistency. We start by measuring the latency of executing a chain of operations posted to the same queue but absent any constraints (WQ consistency), and compare it to the consistency levels that we introduced in Fig. 2 — namely, completion consistency (CMP) and doorbell consistency (DB). WQ consistency only mandates in-order updates to memory, which allows for increased parallelism. Operations that are not modifying the same memory address can be parallelized and the RNIC is also free to prefetch multiple WRs simultaneously with a single DMA.[4] We can see in Fig. 8 that completing a single operation takes around 1.63 µs and the overhead of adding subsequent operations is roughly

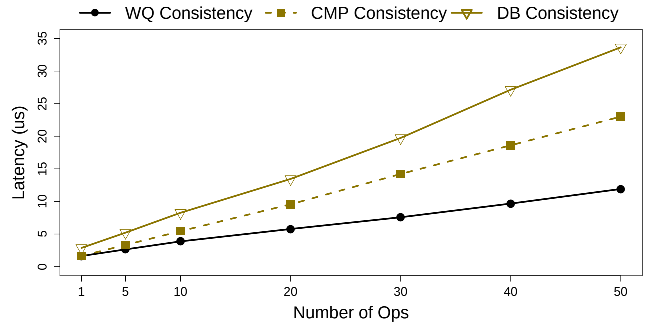


Figure 8: Execution latency for chains of RDMA operations. Chaining adds more overheads, because it requires the NIC to fetch work requests sequentially.

0.24 µs per operation. The first operation is slower since it requires an initial doorbell to tell the NIC that there is outstanding work. For CMP order, less parallelization is possible since WRs await the completions of their predecessors, and the overhead of adding subsequent operations increases to 0.45 µs. For DB consistency, no latency-hiding is possible, as the NIC has to fetch WRs from memory – in this case one-by-one – which results in an overhead of 1.49 µs per additional WR. These results signify that, DB consistency should be used conservatively, as there is more than 1 µs latency increase for every instance of its use, compared to more relaxed consistency levels.

Predictability of RNICs. Based on the results in Fig. 8, we also note that we can accurately predict the performance of any RDMA chain based on its size and consistency level. For all three lines, performance scales linearly with the number of operations in the chain, with a coefficient of determination (which indicates how well a linear regressor performs) greater than 0.99, allowing the latency of an offload to be estimated with high accuracy. Being able to accurately predict performance allows developers to pick offload targets based on their predicted latency and desired service level objectives.

5.1.3 RDMA verb throughput

We show the throughput of the common RDMA verbs in Table 3 for a single ConnectX-5 port. ConnectX cards assign compute resources on a per port basis. For ConnectX-5, each port has 8 PUs. Atomic verbs, such as CAS, offer a comparatively limited throughput (8× lower than regular verbs) due to memory synchronization across PCIe.

In addition, we measure the performance limits of RedN gadgets. Using 24-bit operands, a ConnectX-5 NIC can execute 900K/s `if` statements when using copy verbs and 700K/s using CAS. This is due to the need for CAS to ensure DB consistency between CAS and the subsequent WR it modifies. The throughput for `if` in both cases is bound by NIC processing limits. `while` loops require the same number of operations per iteration as for an `if` statement and their throughput is identical.

[3] Older-generation NICs (e.g., ConnectX-4) use internal concurrency control to implement atomics, resulting in higher latencies.

[4] The number of operations fetched by the RNIC can change dynamically. Prefetcher operation in ConnectX RNICs is unpublished.

Operation			Throughput (M ops/s)	Support
Atomic	CAS		8.4	Native
	ADD			
Copy	READ		65	
	WRITE		63	
Gadgets	if	Copy	0.9	RedN
		CAS	0.7	
	while	Copy	0.9	
		CAS	0.7	

Table 3: List of all available RDMA operations and their performance in a ConnectX-5 NIC. if and while have similar performance because the conditional branching to check the break condition is what dominates the loop iteration execution.

5.2 Hash Lookups

Key-value stores often use hash tables to manage their stored objects. To perform a simple *get* operation, clients first have to lookup the desired key-value entry in the hash table. The entry can either have the value directly inlined or a pointer to its memory address. The value is then fetched and returned back to the client. Hopscotch hashing is a popular hashing scheme that resolves collisions by using H hashes for each entry and storing them in 1 out of H buckets. Each bucket has a *neighborhood* that can probabilistically hold a given key. A lookup might require searching more than one bucket before the matching key-value entry is found.

For distributed key-value stores built over RDMA, *get* operations are usually implemented in one of two ways:

One-sided approaches first retrieve the key's location using a one-sided RDMA READ operation and then issue a second READ to fetch the value. These approaches typically require two roundtrips at a minimum — which greatly increases latency but does not require involvement of the server's CPU. Many systems utilize this approach to implement their lookups — including FaRM [24] and Pilaf [37].

Two-sided approaches require the client to send a request using an RDMA SEND or WRITE. The server intercepts the request, locates the value and then returns it using one of the aforementioned verbs. This widely used [20,28] approach follows traditional RPC implementations and avoids the need for

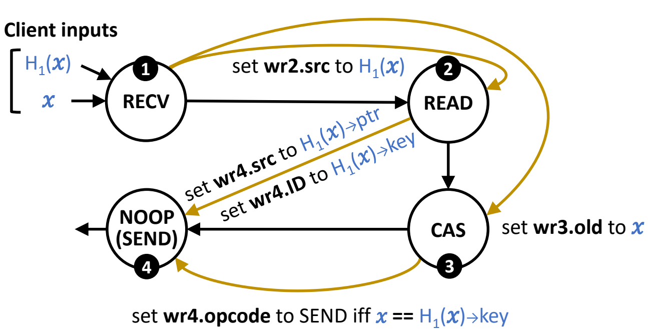


Figure 9: Hash lookup offload schema using CAS. Black arrows indicate order of execution of WRs in their WQs. Brown arrows indicate self-modifying code dependencies and require doorbell consistency. x is the requested key and $H_1(x)$ is its first hash.

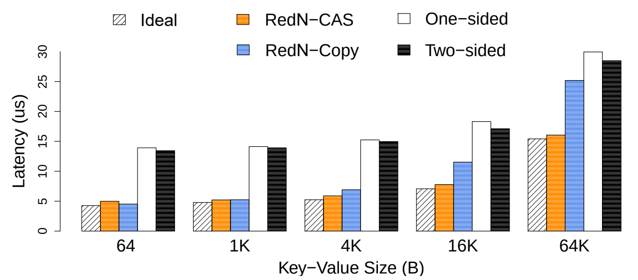


Figure 10: Average latency of hash lookups. *Ideal* shows the latency of a single network round-trip READ.

several roundtrips. However, this comes at the cost of server CPU cycles in order and can introduce scalability concerns with larger numbers of clients.

5.2.1 Our approach

To offload key-value *get* operations, we leverage the offload schemes we introduced earlier in §3.3 and §3.4 — namely those that use CAS to perform conditional logic and other schemes that rely only on WRITE verbs. For the remainder of this evaluation, we will refer to them as *RedN-CAS* and *RedN-Copy*, respectively.

To *get* a value corresponding to a key, the client first computes the hashes for its key. Fig. 9 describes the RDMA operations involved for a single-hash lookup using CAS. A similar schema is used for Copy-based approaches which, for brevity, we leave out. For this use-case, we set the number of hashes to two — which is commonly used in practice [26]. The client then performs a SEND with the value of the key and hashes, which are then captured via a RECV WR posted on the server. The RECV request first modifies the addresses of a posted READ WR to allow it to retrieve the bucket(s) containing the key-value pairs. The value may be either inlined in the bucket or require a layer of indirection. For our purposes, we assume the value is not inlined and is referenced via a pointer (*ptr*), since inlining limits general usability (and typically requires value sizes to be fixed). We then perform an if statement to check if the key matches that retrieved from the bucket. For RedN-Copy, all buckets are checked regardless of where the key-value is — whereas for RedN-CAS, buckets are checked one-by-one until the required key-value is returned.

5.2.2 Results

We evaluate our approach against both one-sided and two-sided implementations of key-value *get* operations. We use a FaRM-like approach [24] to perform one-sided lookups. FaRM uses Hopscotch hashing to locate the key using approximately two RDMA READs — one for fetching the buckets in a neighborhood that hold the key-value pairs and another for reading the actual value. The neighborhood size is set to 6 by default, implying a 6× overhead for RDMA metadata operations. For two-sided lookups, we use a traditional RPC, which involves a client-initiated RDMA WRITE to transmit the *get* request, and another RDMA Write initiated by the

Hash lookup ($H = 2$)	CAS		Copy	
	≤ 1 KB	64 KB	≤ 1 KB	16 KB
Rate (ops)	430K	180K	550K	200K
Bottleneck	NIC PU	IB bw	NIC PU	PCIe bw

Table 4: Throughput of hash lookups and bottlenecks for RedN-CAS and RedN-Copy. *H* is the number of hashes.

server to return the value after performing the lookup.

Latency. Fig. 10 shows a latency comparison of KV *get* operations of our offload engine against one-sided and two-sided baselines. We use 24-bit keys and vary the value size. The total size of the key-value pair is given on the x-axis. Our offload strategies provide the best performance — fetching a 64-byte key-value pair in as low as 4.52 µs, which is within 6% of a single network round-trip READ (Ideal). Our offloads are able to deliver close-to-ideal performance because they bypass the server's CPU completely ***and*** can fetch the value in a single network RTT. RedN-Copy offers slightly lower latency (∼0.5 µs) than RedN-CAS given that it does not use CAS, which adds a doorbell constraint to WR fetch. Compared to our offloads, one-sided operations incur up to 3x higher latencies, as they require two RTTs to fetch a value. Two-sided performs slightly better than one-sided approaches since they do not incur any extra RTT. However, they require server CPU intervention and do not scale nearly as well with a higher number of clients. Interestingly enough, we see that, as IO size increases RedN-Copy starts performing progressively worse — and RedN-CAS is universally better at 4K IO sizes or higher. This is because, despite have more relaxed consistency, RedN-Copy has to carry out additional copies. Since we use two hash functions (and have two possible buckets), RedN-Copy first writes the value pointed to by each bucket to a special memory location (the address of the key), and then performs another WRITE to return it back to the requester. This means that, as the IO sizes increases, the copy overhead eclipses the reduced consistency constraints.

Throughput. We describe the system throughput in Table 4. We see similar trends as in the latency results where, at lower IO sizes, RedN-Copy can perform better than RedN-CAS— and both schemas are bottlenecked by the NIC's processing units. At 64 KB key-value sizes, CAS reaches the IB bandwidth limit — and reaches 92 Gbps. Whereas, for RedN-Copy, at 16 KB, it starts becoming bottlenecked by the PCIe bandwidth due to the extra copies — reaching 25 Gbps and never exceeding that bandwidth, even at higher IO sizes.

SmartNIC comparison. We compare our performance for hashtable *gets* against StRoM [41], a programmable FPGA-based SmartNIC. Since we do not have access to a programmable FPGA, we extract the results from [41] for comparison, and report them in Table 5. Our hashtable configuration is functionally identical to StRoM's and our client and server nodes are also connected via back-to-back links. We can see that RedN provides lower latencies than StRoM. StRoM uses a low-end Xilinx Virtex 7 FPGA, which runs at 156.25 MHz and also incurs at least two PCIe roundtrips to retrive the key and value. Newer generation FPGAs with higher clock rates may be able to provide better performance than RNICs given that we have to traverse PCIe for every RDMA operation. However, our evaluation shows that, once programmable, RNICs can provide latency that is in-line with more expensive SmartNICs.

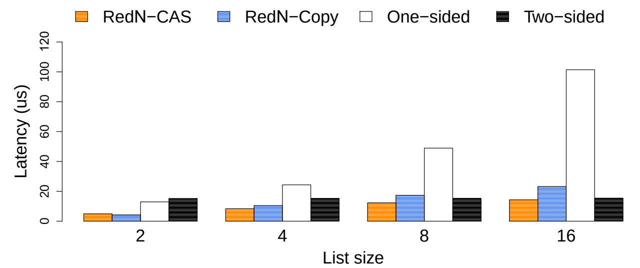


Figure 11: Average latency of walking linked lists.

5.3 List Traversal

Next, we explore another data structure also popularly used in storage systems. We focus on linked lists that store key-value pairs, and evaluate the overhead of traversing them remotely using our offloads. Similar to the previous use-case, we focus on One-sided approaches as used by FaRM and Pilaf [24, 37].

For linked lists, our computations can be decomposed into a *while* loop for traversing the list and an *if* condition for finding an returning the key. Similar to our previous use-case, we evaluate the performance of RedN-Copy and RedN-CAS.

5.3.1 Results

Fig.11 shows the latency of one-sided and two-sided variants against our offload strategies at various linked list sizes. We setup the linked list to use a constant key-value size of 64 Bytes and perform 100k traversal operations for each system. The requested key is chosen at random for each RPC. At lower list sizes, RedN-Copy outperforms RedN-CAS as it does not use the more expensive atomic operations. However, as we increase the list size, RedN-CAS's latency is up to 30% lower, since RedN-Copy lacks any break capability and has to traverse the entire list even if the key if found earlier. RedN-CAS breaks and immediately returns once the key is found and is more efficient at larger list sizes. Both approaches are markedly better than the baselines, up to a factor of 5 and 10 over two-sided and one-sided, respectively. We note that two-sided is largely unaffected by list length, since this only requires additional CPU cycles, with no added roundtrips. One-sided, on the other hand, has to fetch the linked list ele-

IO Size	**System**	**Median**	**99^{th}ile**
64 B	RedN	4.4 µs	5.6 µs
	StRoM	7.1 µs	7.3 µs
4 KB	RedN	7.7 µs	8.6 µs
	StRoM	12.6 µs	12.9 µs

Table 5: Latency comparison of hash *gets*. Results for StRoM obtained from [41].

ments one-by-one, each incurring an RTT, and its latency is up to 100 µs for linked lists containing 16 elements.

5.4 Use Case: Accelerating Memcached

Based on our earlier experience offloading hash lookups and linked list traversals, we set out to see how effective our aforementioned techniques in a real system, and the challenges in deploying it in such settings. Memcached is a key-value store that is often used as a caching service for large-scale storage services. It uses chain-hashing to store and index its data objects — which, for hash collisions, uses a linked list to store objects that hash to the same bucket. As such, we see this as a combination of our offload use-cases in §5.2 and §5.3. Since Memcached does not natively support RDMA, we modify it with ∼700 LoC to integrate RDMA capabilities and allow the RNIC to register the hash table and storage object memory areas. We then used RedN to provide a full-offload for Memcached's *get* requests to allow them to be serviced directly by the NIC without CPU involvement, and compare our results to various configurations of Memcached.

To benchmark Memcached, we use the Memtier benchmark, configure it to use UDP (to reduce TCP overheads for the baselines), and issue 1 million *get* operations using different key-value sizes. To create a competitive baseline for two-sided approaches, we use Mellanox's VMA [10]—a kernel-bypass userspace TCP/IP stack that boosts the performance of sockets-based applications by intercepting their socket calls and using RDMA to send/receive data (RPC-VMA). Figure 12 shows that RPC-VMA still takes 19.8 µs to fetch a 64 byte key-value pair. This is because VMA relies on a network stack to process packets. In addition, to adhere to the socket API, VMA has to memcpy data from send and receive buffers, further inflating latencies. We also compare to a version of Memcached using our RDMA patch to support an offloaded two-sided RPC (RPC-OPT) and a one-sided variation that performs two or more READ operations to find the value without invoking any Memcached code at the server (one-sided). We consider this approach naive for Memcached's hash function, since hash collisions can result in an unbounded number of linked list traversals (as opposed to cuckoo hashing).

The results show that, RedN's offload for hash *gets* is up to 3x and 2.7x faster than RPC-OPT and one-sided, respectively. RPC-OPT is slower than one-sided as it incurs a couple of µs of processing delay at the Memcached server application.

5.5 Use Case: Performance Isolation

One of the benefits of exposing the latent turing power of RNICs is to enforce isolation between applications. CPU contention in multi-tenant and cloud settings, can lead to arbitrary context switches, which can, in turn, inflate average and tail latencies. We explore such a scenario by sending background traffic to Memcached using one or more writers (clients). These clients generate *set* RPCs in a closed-loop

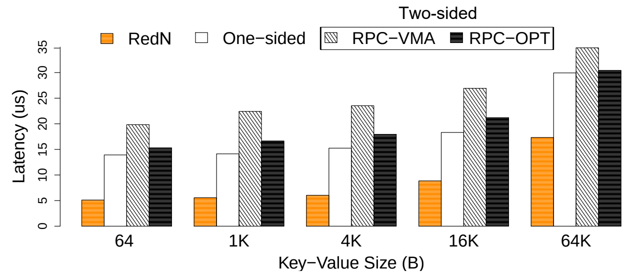


Figure 12: Memcached *get* latencies with different IO sizes.

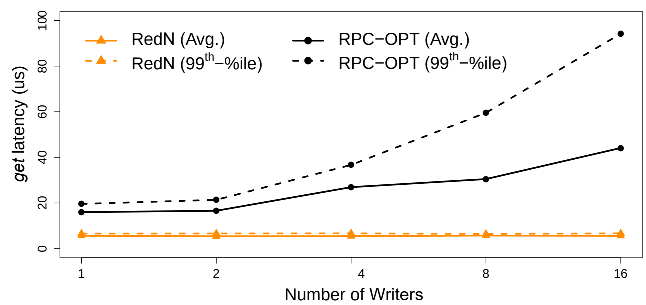


Figure 13: Memcached *get* latencies under hardware contention with varying numbers of writer-clients.

fashion to load the Memcached service. At the same time, we perform *get* requests on a separate partition (to eliminate software contention) and observe the latency impact on our *gets*. We can see that, as we increase the # of writers, both the average and 99^{th} percentile latencies for RPC-OPT increase dramatically. At 16 writers, the 99^{th} percentile is 5× higher than the average. For RedN, CPU contention has no impact on the performance of the RNIC, and both the average and 99^{th} percentiles are unaffected and stay at below 7 µs.

This indicates that RNIC offloads can also have other useful side effects. Service providers may opt to offload high priority traffic for more predictable performance or allocate server resources to tenants to reduce contention.

5.6 Use Case: Failure Resiliency

Lastly, we consider server failure models and how it is affected by the Turing-complete RNICs. Table 6 shows failure rates of server software and hardware components. NICs are much less likely to fail than software components and their Average Failure Rate (AFR) is an order of magnitude lower. Even more importantly, RNICs are partially decoupled and can still access memory (or NVM) in the presence of an OS failure. This means that RNICs are capable of offloading key system functionality can allow servers to continue operating despite OS failures (albeit in a degraded state). To put this to the test, we conduct a fail-over experiment to explore how RedN can enhance a service's failure resiliency.

Process crashes. We look into how we can allow an RNIC to continue serving RPCs after a Memcached instance crashes. We find that this is not simple in practice. RNICs access many resources in application memory (e.g., queues, doorbell

Component	AFR	MTTF	Reliability
OS	41.9%	20,906	99%
DRAM	39.5%	22,177	99%
NIC	1.00%	876,000	99.99%
NVM	$< 1.00\%$	2 million	99.99%

Table 6: Failure rates of different server components. RNICs can still access memory even in the presence of an OS failure. Numbers are obtained from [9, 39].

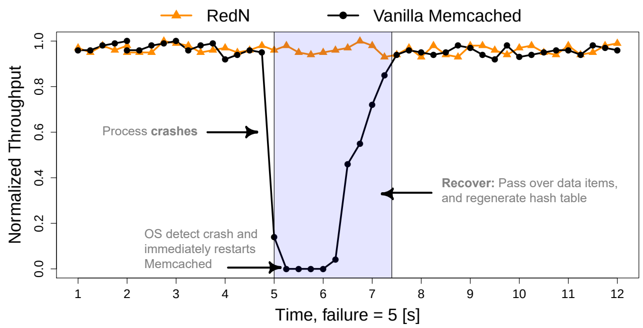


Figure 14: RedN can survive process crashes and continue serving RPCs via the RNIC without interruption.

records, etc.) that are required for it to keep functioning. If the process hosting these resources crashes, the memory belonging to these components will be automatically freed by the operating system resulting in termination of the RDMA program. To counteract this, we use [40] forks to create an empty hull parent for hosting RDMA resources and then allow Memcached to run as a child process. Linux systems do not free the resources of a crashed child until the parent also terminates. As such, keeping the RDMA resources tied to an empty process allows us to continue operating in spite of application failures. We run an experiment (timeline shown in Fig. 14) where we send *get* queries to a single instance of Memcached and then simply kill it during the run. The OS then detects the application's termination and immediately restarts it. Despite this, we can see that a Vanilla Memcached instance will take at least 1 second to bootstrap, and 1.25 additional seconds to build its metadata and hashtables. With RedN, no service disruption is experienced and *get* queries continue to be issued with zero recovery time.

OS failure. We also programmatically induce a kernel panic using sysctl — freezing the system. This can be seen as a more trivial case of process crashes, since we no longer have to worry about the OS freeing up RDMA resources. For brevity, we do not show these results, but we experimentally verified that we can continue our offloads in the presence of an OS crash.

6 Discussion

How can developers pick between Copy-based and CAS-based approaches? Each approach has its own set of trade-offs. RedN-Copy has more relaxed ordering constraints (and is thus faster) compared to RedN-CAS but it is only suited to offloads with limited branching (i.e., hash gets) as it is unable to modify the control flow of a program. RedN-Copy also requires dedicated memory regions for comparisons. RedN-CAS is more general and is capable of modifying the control flow. However, this comes at the cost of stricter ordering constraints — requiring an additional barrier, which adds overhead (evaluated in §5.1.2). Ultimately, the appropriate approach is dependent on the target use-case.

Can you offload remote lock acquisition? Is it possible to pair this with storage systems to provide strong consistency? Yes. Locking is generally less complex than *get* or *set* operations. They can be acquired through atomic verbs, and paired with an if gadget to check if lock has been acquired. RedN offers a distinct advantage against the state-of-the-art [47] as it can chain together locking with storage operations — and not incur multiple network RTTs. We leave this as future work.

Does your approach scale to a large number of clients? RedN requires servers to manage only two WQs per client, which is not higher than other RDMA systems. This can still introduce scalability challenges with thousands of clients since NIC SRAM cache is limited, however, Mellanox's dynamically-connected (DC) transport service [6] — which allows unused connections to be recycled — can circumvent such scalability limits. Scratch regions for Copy-based gadgets can also impose scalability limits. To support C clients using an operand size of N bits, $C \times 2^N$ bytes of scratch memory needs to be registered. With $C = 1000$ clients and $N = 24$ bits, $C \times 2^N \approx 16$ GB of memory is required. For larger operands, where memory cost can become prohibitive, CAS-based approaches can be used instead.

Can your approach be used with applications that do not support RDMA natively? Protocols such as rsocket [17] can be used to transparently translate socket operations to RDMA verbs — making such applications possible targets for offload. Although rsocket does not support popular syscalls such as epoll, other extensions have been proposed [31] that support a more comprehensive list of syscalls and were shown to work with applications like Memcached and Redis.

7 Conclusion

We show that, in spite of appearances, commodity RDMA NICs are Turing-complete and capable of performing complex offloads without *any* hardware modifications. We take this insight and explore the feasibility and performance of these offloads. We find that, using a commodity RNIC, we can achieve up to 3× and 7.8× speed-up versus state-of-the-art RDMA approaches, for key-value get operations and under performance isolation, respectively, while allowing applications to gain failure resiliency to OS and process crashes. We believe that this work opens the door for a wide variety of innovations in RNIC offloading which, in turn, can help guide the evolution of the RDMA standard.

References

[1] Address space randomization. https://lwn.net/Articles/121845/.

[2] Agilio CX SmartNICs. https://www.netronome.com/products/agilio-cx/.

[3] Catapult. https://www.microsoft.com/en-us/research/project/project-catapult/.

[4] Cavium-Xpliant. https://www.openswitch.net/cavium/.

[5] ConnectX series. https://www.mellanox.com/products/ethernet/connectx-smartnic.

[6] Dynamically Connected (DC) QPs. https://docs.mellanox.com/display/rdmacore50/DynamicallyConnected(DC)QPs.

[7] ibv_modify_qp_rate_limit(3) - Linux man page. https://man7.org/linux/man-pages/man3/ibv_modify_qp_rate_limit.3.html.

[8] Intel Ethernet 800 Series Network Adapters. https://www.intel.com/content/www/us/en/products/docs/network-io/ethernet/network-adapters/ethernet-800-series-network-adapters/e810-cqda1-100gbe-brief.html.

[9] Intel Optane DC Persistent Memory - Product Brief. https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/optane-dc-persistent-memory-brief.pdf.

[10] LibVMA. https://github.com/Mellanox/libvma/wiki/Architecture.

[11] LiquidIO II SmartNICs. https://www.marvell.com/products/ethernet-adapters-and-controllers/liquidio-smart-nics/liquidio-ii-smart-nics.html.

[12] Mellanox BlueField. https://www.mellanox.com/products/bluefield-overview.

[13] Mellanox PCX. https://github.com/Mellanox/pcx/tree/master/config.

[14] Mellanox store. http://store.mellanox.com/.

[15] NetFPGA platform. https://netfpga.org/.

[16] RDMA RFC. https://tools.ietf.org/html/rfc5040.

[17] rsocket(7) - Linux man page. https://linux.die.net/man/7/rsocket.

[18] Stingray. https://www.broadcom.com/products/ethernet-connectivity/smartnic.

[19] M. K. Aguilera, N. Ben-David, R. Guerraoui, V. Marathe, and I. Zablotchi. The Impact of RDMA on Agreement. *arXiv preprint arXiv:1905.12143*, 2019.

[20] T. E. Anderson, M. Canini, J. Kim, D. Kostić, Y. Kwon, S. Peter, W. Reda, H. N. Schuh, and E. Witchel. Assise: Performance and Availability via NVM Colocation in a Distributed File System. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020.

[21] O. Cardona. Towards Hyperscale High Performance Computing with RDMA, 2019. https://pc.nanog.org/static/published/meetings/NANOG76/1999/20190612_Cardona_Towards_Hyperscale_High_v1.pdf.

[22] S. Dolan. mov is Turing-complete. *Cl. Cam. Ac. Uk*, pages 1–4, 2013.

[23] M. Dong, H. Bu, J. Yi, B. Dong, and H. Chen. Performance and protection in the ZoFS user-space NVM file system. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 478–493, 2019.

[24] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson. FaRM: Fast remote memory. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 401–414, 2014.

[25] H. Eran, L. Zeno, M. Tork, G. Malka, and M. Silberstein. NICA: An Infrastructure for Inline Acceleration of Network Applications. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 345–362, 2019.

[26] B. Fan, D. G. Andersen, and M. Kaminsky. Memc3: Compact and concurrent memcache with dumber caching and smarter hashing. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 371–384, 2013.

[27] M. Gabbrielli and S. Martini. *Programming Languages: Principles and Paradigms*, page 145. Undergraduate Topics in Computer Science. Springer London, 2010.

[28] A. Kalia, M. Kaminsky, and D. G. Andersen. Using RDMA efficiently for key-value services. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 295–306. ACM, 2014.

[29] M. Kazhamiaka, B. Memon, C. Kankanamge, S. Sahu, S. Rizvi, B. Wong, and K. Daudjee. Sift: resource-efficient consensus with RDMA. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, pages 260–271, 2019.

[30] D. Kim, A. Memaripour, A. Badam, Y. Zhu, H. H. Liu, J. Padhye, S. Raindel, S. Swanson, V. Sekar, and S. Seshan. Hyperloop: group-based NIC-offloading to accelerate replicated transactions in multi-tenant storage systems. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 297–312, 2018.

[31] B. Li, T. Cui, Z. Wang, W. Bai, and L. Zhang. Socks-Direct: Datacenter sockets can be fast and compatible. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 90–103. 2019.

[32] B. Li, Z. Ruan, W. Xiao, Y. Lu, Y. Xiong, A. Putnam, E. Chen, and L. Zhang. KV-Direct: High-Performance In-Memory Key-Value Store with Programmable NIC. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 137–152, 2017.

[33] M. Liu, T. Cui, H. Schuh, A. Krishnamurthy, S. Peter, and K. Gupta. Offloading distributed applications onto smartNICs using iPipe. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 318–333. 2019.

[34] M. Liu, S. Peter, A. Krishnamurthy, and P. M. Phothilimthana. E3: Energy-Efficient Microservices on SmartNIC-Accelerated Servers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 363–378, 2019.

[35] Y. Lu, J. Shu, Y. Chen, and T. Li. Octopus: An RDMA-enabled Distributed Persistent memory file system. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 773–785, 2017.

[36] Mellanox RDMA Aware Networks Programming User Manual. https://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf.

[37] C. Mitchell, Y. Geng, and J. Li. Using One-Sided RDMA Reads to Build a Fast, CPU-Efficient Key-Value Store. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 103–114, 2013.

[38] P. M. Phothilimthana, M. Liu, A. Kaufmann, S. Peter, R. Bodik, and T. Anderson. Floem: A Programming System for NIC-Accelerated Network Applications. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 663–679, 2018.

[39] M. Poke and T. Hoefler. Dare: High-performance State Machine Replication on RDMA Networks. In *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, pages 107–118. ACM, 2015.

[40] A. Rosenbaum. Multiprocess Sharing of RDMA Resources, 2018. https://openfabrics.org/images/2018workshop/presentations/103_ARosenbaum_Multi-ProcessSharing.pdf.

[41] D. Sidler, Z. Wang, M. Chiosa, A. Kulkarni, and G. Alonso. StRoM: Smart Remote Memory. *Proc. of EuroSys. ACM*, 2020.

[42] A. K. Simpson, A. Szekeres, J. Nelson, and I. Zhang. Securing RDMA for High-Performance Datacenter Storage Systems. In *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*, 2020.

[43] C. Wang, J. Jiang, X. Chen, N. Yi, and H. Cui. APUS: Fast and Scalable Paxos on RDMA. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 94–107. ACM, 2017.

[44] X. Wei, Z. Dong, R. Chen, and H. Chen. Deconstructing RDMA-enabled distributed transactions: Hybrid is better! In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 233–251, 2018.

[45] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast in-memory transaction processing using RDMA and HTM. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 87–104, 2015.

[46] J. Yang, J. Izraelevitz, and S. Swanson. Orion: A distributed file system for non-volatile main memory and RDMA-capable networks. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 221–234, 2019.

[47] D. Y. Yoon, M. Chowdhury, and B. Mozafari. Distributed Lock management with RDMA: Decentralization without Starvation. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1571–1586. ACM, 2018.

[48] T. Ziegler, S. Tumkur Vani, C. Binnig, R. Fonseca, and T. Kraska. Designing Distributed Tree-based Index Structures for Fast RDMA-capable Networks. In *Proceedings of the 2019 International Conference on Management of Data*, pages 741–758. ACM, 2019.

Appendix A Turing completeness sketch

To showcase that RDMA is turing complete, we need to establish that RDMA has the following three properties:

1. Can read/write arbitrary amounts of memory.
2. Has conditional branching (e.g., if & else statements).
3. Allows nontermination.

Our paper already demonstrates that these properties can be satisfied using our gadgets but, for completeness, we also analogize our system with `x86` assembly instructions that have been proven to be capable of simulating a Turing machine. Dolan [22] demonstrated that this is in fact possible using just the `x86 mov` instruction. As such, we need to prove that RDMA has sufficient expressive power to emulate a `mov` instruction.

A.1 Emulating the `x86 mov` instruction

To provide an RDMA implementation for `mov`, we first need to consider the different addressing modes used by Dolan [22] to simulate a Turing machine. The addressing mode describes how a memory location is specified in the `mov` operands.

Table 7 shows a list of all required addressing modes, their `x86` syntax, and one possible implementation for each with RDMA. *R* operands denote registers but, since RDMA operations can only perform memory-to-memory transfers, we assume these registers are stored in memory. For simplicity, we only focus on `mov` instructions used to perform *loads* but note that *stores* can be implemented in a similar manner.

For *immediate* addressing, the operand is part of the instruction and is passed directly to register R_{dst}. This can be implemented simply using an WRITEIMM which takes a constant in its *immediate* parameter and writes it to a specified memory location (register R_{dst} in this case). To perform more complex operations, *indirect* allows `mov` to use the value of the operand as a memory address. This enables the dynamic modification of the address at runtime, since it depends on the contents of the register when the instruction is executed. To implement this, we use two write operations with doorbell consistency (refer to §3.1 for a discussion of our consistency modes). The first WRITE changes the *source address* attribute of the second WRITE operation to the value in register R_{src}. This allows the second WRITE operation to write to register R_{dst} using the value at the memory address pointed to by R_{src}. Lastly, *indexed* addressing allows us to add an offset (R_{off}) to the address in register R_{src}. This can be done by simply performing an RDMA ADD operation between the two writes with doorbell consistency, in order to add the offset register value R_{off} to R_{src}. This allows us to finally write the value $[R_{src} + R_{off}]$ to R_{dst}. With these three implementations, we showcase that RDMA can in fact emulate all the required `mov` instruction variants.

A.2 Allowing nontermination

To simulate a real Turing machine, we need to also satisfy the code nontermination requirement. In the `x86` architecture, this can be achieved via an unconditional jump [22] that loops back to the start of the program. For RDMA, this can also be achieved by having the CPU re-post the WRs after they are executed. While this is sufficient for Turing completeness it, nevertheless, wastes additional CPU cycles and can also impact latency if CPU cores are busy or unable to keep up with WR execution. As an alternative, RedN provides a way to loop back without any CPU interaction by relying on the WAIT and ENABLE instructions to recycle RDMA WRs (as described in §3.4.2). Regardless of which approach is employed, RDMA is capable of performing an unconditional jump to the beginning of the program. This means that we can emulate all `x86` instructions used by Dolan [22] for simulating a Turing machine.

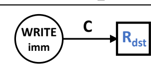
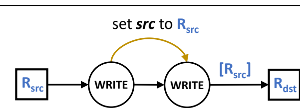
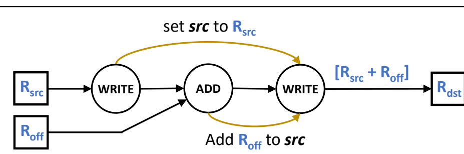
Addressing mode	x86 syntax	RedN equivalent
Immediate	`mov` R_{dst}, C	
Indirect	`mov` R_{dst}, $[R_{src}]$	
Indexed	`mov` R_{dst}, $[R_{src} + R_{off}]$	

Table 7: Addressing modes for the x86 mov instruction and their RDMA implementation in RedN.