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
University of Washington

Abstract

It is becoming increasingly popular for distributed systems to exploit offload to reduce load on the CPU. Remote Direct Memory Access (RDMA) offload, in particular, has become popular. However, RDMA still requires CPU intervention for complex offloads that go beyond simple remote memory access. As such, the offload potential is limited and RDMA-based systems usually have to work around such limitations.

We present RedN, a principled, practical approach to implementing complex RDMA 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 a commodity RDMA NIC. We show how to integrate these RDMA chains into applications, such as the Memcached key-value store, allowing us to offload complex tasks such as key lookups. RedN can reduce the latency of key-value get operations by up to 2.6× compared to state-of-the-art KV designs that use one-sided RDMA primitives (e.g., FaRM-KV), as well as traditional RPC-over-RDMA approaches. Moreover, compared to these baselines, RedN provides performance isolation and, in the presence of contention, can reduce latency by up to 35× while providing applications with failure resiliency to OS and process crashes.

1 Introduction

As server CPU cycles become an increasingly scarce resource, offload is gaining in popularity [23, 28, 30–32, 36]. System operators wish to reserve CPU cycles for application execution, while common, oft-repeated operations may be offloaded. 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 [31].

For this reason, remote direct memory access (RDMA) [15] has become ubiquitous [20]. Mellanox ConnectX NICs [4] have pioneered ubiquitous RDMA support and Intel has added RDMA support to their 800 series of Ethernet network adapters [7]. RDMA focuses on the offload of simple message passing (via SEND/RECV verbs) and remote memory access (via READ/WRITE verbs) [15]. 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. For example, remote data structure traversal and hash table access are not normally deemed realizable with RDMA [39]. This led to many RDMA-based systems requiring multiple network round-trips or to reintroduce involvement of the server's CPU to execute such requests [18, 22, 26, 27, 35, 37, 41].

To support complex offloads, the networking community has developed a number of SmartNIC architectures [2, 3, 11, 14, 17]. SmartNICs incorporate more powerful compute capabilities via CPUs or FPGAs. They can execute arbitrary programs on the NIC, including 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 conditionals or loops, to realize complex offloads. Our solution has to be general so that offload developers can use it to build complex *RDMA programs* that can perform a wide range of functionality. 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 RDMA is *Turing complete*, making it 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 program, 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 the WAIT and ENABLE RDMA verbs [28, 34] that provide execution dependencies. WAIT allows us to halt execution of new verbs until past verbs have completed, providing strict ordering among RDMA verbs. By controlling verb prefetching, ENABLE enforces consistency for verbs modified by preceding verbs. ENABLE also allows us to create loops by re-triggering earlier, already-executed verbs in an RDMA work queue—allowing the NIC to operate autonomously without CPU intervention.

Based on these primitives, we present RedN, a principled, practical approach to implementing complex RNIC offloads. Using self-modifying RDMA programs, we develop a number of building blocks that lift the existing RDMA verbs interface to a Turing complete set of programming abstractions. Using these abstractions, we explore what is possible in terms of offload complexity and performance with just a commodity RNIC. 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. Our code is available at: https://redn.io.

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. We show that these primitives are sufficient to make RDMA Turing complete.
- 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 key-value get operations, as well as performance isolation and failure resiliency benefits. We demonstrate that RNIC offload with RedN can realize all of these benefits. It can reduce average latency of get operations by up to 2.6× compared to state-of-the-art one-sided RDMA key-value stores (e.g., FaRM-KV [22]), as well as traditional two-sided RPC-over-RDMA implementations. Moreover, RedN provides superior performance isolation, improving latency by up to 35× under contention, while also providing higher availability under host-side failures.

2 Background

RDMA was conceived for high-performance computing (HPC) clusters, but it has grown out of this niche [20]. It is becoming ever-more popular due to the growth in network bandwidth, with stagnating growth in CPU performance, making CPU cycles an increasingly scarce resource that is best reserved to running application code. With RNICs now considered commodity, it is opportunistic to explore the use-cases where their hardware can yield benefits. These efforts, however, have been limited by the RDMA API, which constrains the expression of many complex offloads. Consequently, the networking community has built SmartNICs using FPGAs and CPUs to investigate new complex offloads.

2.1 SmartNICs

To enable complex network offloads, SmartNICs have been developed [1, 2, 10, 11]. SmartNICs include dedicated computing units or FPGAs, memory, and several dedicated accelerators, such as cryptography engines. For example, Mellanox BlueField [11] has 8×ARMv8 cores with 16GB of memory and 2×25GbE ports. These SmartNICs are capable of running full-fledged 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 offload. SmartNICs have been used to offload complex tasks from server CPUs. For example, StRoM [39] 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 [30] uses an FPGA NIC to accelerate key-value accesses. iPipe [31] and Floem [36] are programming frameworks that simplify complex offload development for primarily CPU-based SmartNICs. E3 [32] 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. [13]). Another cost is the additional 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 by RNICs.

2.2 RDMA NICs

The processing power of RDMA NICs (RNICs) 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).

We measure the verb processing bandwidth of several generations of Mellanox ConnectX NICs, using the Mellanox ib_write_bw benchmark. This benchmark performs 64B RDMA writes and, as such, it is not network bandwidth limited due to the small RDMA write size. We find that the verb processing bandwidth doubles with each generation, as we can

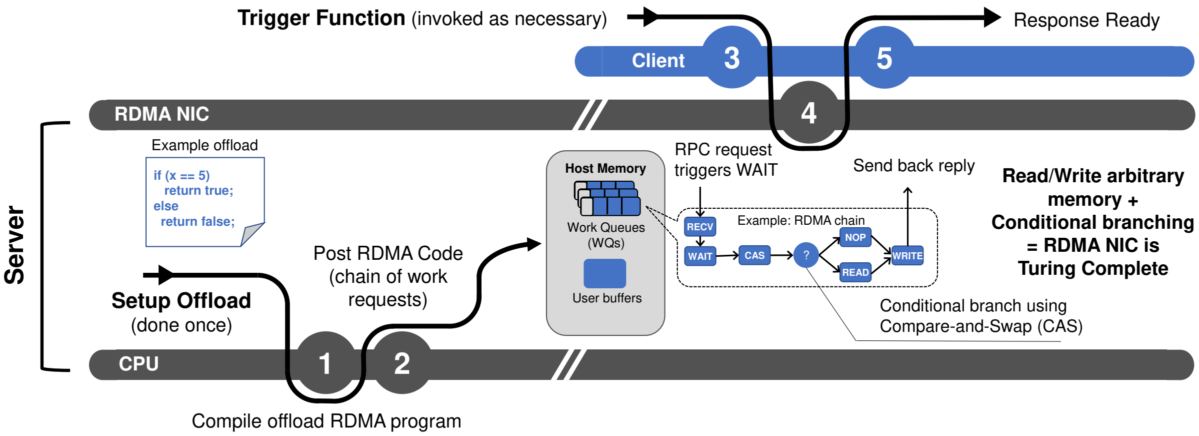

Trigger Function (invoked as necessary)
Response Ready
Client
3
5
RDMA NIC
4
Example offload
if (x == 5)
return true;
else
return false;
Host Memory
Work Queues (WQs)
User buffers
RPC request triggers WAIT
Send back reply
Example: RDMA chain
RECV
WAIT
CAS
?
NOP
READ
WRITE
Read/Write arbitrary memory + Conditional branching = RDMA NIC is Turing Complete
Server
Setup Offload
(done once)
Post RDMA Code (chain of work requests)
Conditional branch using Compare-and-Swap (CAS)
CPU
1
2
Compile offload RDMA program

Figure 1: RDMA NICs can implement complex offloads if we allow conditional branches to be expressed. Conditional branching can be implemented by using CAS verbs to modify subsequent verbs in the chain, without any hardware modification.

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 verbs per second using a single NIC port. This increased 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, including accelerating key-value stores and filesystems [19, 22, 26, 35, 44], consensus [18, 27, 37, 41], distributed locking [45], and even nuanced use-cases such as efficient access in distributed tree-based indexing structures [46]. These systems operate within the confines of RDMA's intended use as a *data movement* offload (via remote memory access and message passing). When complex functionality is required, these systems involve multiple RDMA round-trips and/or rely on host CPUs to carry out the complex operations.

Within the storage context, Hyperloop [28] demonstrated that pushing the RNIC offload capabilities is possible. Hyperloop combines RDMA verbs to implement complex storage operations, such as chain replication, without CPU involvement. However, it does not provide a blueprint for offloading arbitrary processing and cannot offload functionality that uses any type of conditional logic (e.g., walking a remote data structure). Moreover, the Hyperloop protocol is likely incompatible with next-generation RNICs, as its implementation relies on changing work request ownership—a feature that is deprecated for ConnectX-4 and newer cards.

Unlike this body of previous work, we aim to unlock the general-purpose processing power of RNICs and provide an unprecedented level of programmability for complex offloads, by using novel combinations of existing RDMA verbs (§3).

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

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

3 The RedN Computational Framework

To achieve our aforementioned goals, we develop a framework that enables complex offloads, called RedN. RedN's key idea is to combine widely available capabilities of RNICs to enable self-modifying RDMA programs. These programs—chains of RDMA operations—are capable of executing dynamic control flows with conditionals and loops. Fig. 1 illustrates the usage of RedN. The setup phase involves (❶) preparing/compiling the RDMA code required for the service and (❷) posting the output chain(s) of RDMA WRs to the RNIC. Clients can then use the offload by invoking a trigger (❸) that causes the server's RNIC to (❹) execute the posted RDMA program, which returns a response (❺) to the client upon completion.

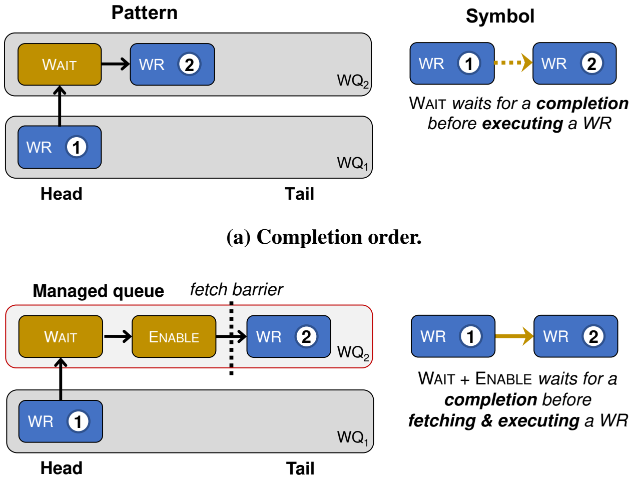
To further understand this proposed framework, we first look into the execution models offered by RNICs, and the ordering guarantees they provide for RDMA verbs. We then look into the expressivity of traditional RDMA verbs and explore parallels with CPU instruction sets. We use these insights to describe 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 host 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 ordering. Ordering rules for RDMA WRs distinguish between write WRs and non-write WRs that return a value. Within each category of operations, RDMA guarantees in-order execution of WRs within a single WQ. In particular, write WRs (i.e., Send, Write, WriteImm) are totally or-

[1] Discussions with Mellanox affirmed our findings.



(a) Completion order.

(b) Doorbell order.

Figure 2: Work request ordering modes that guarantee a total order of operations 2a and, a more restrictive "doorbell" order 2b, where operations are fetched by the NIC one-by-one. The symbols on the right will be used as notation for these WR chains in the examples of §3.

dered with regard to each other, but writes may be reordered before prior non-write WRs.

We call the default RDMA ordering mode *work queue (WQ) ordering*. Sophisticated offload logic often requires stronger ordering constraints, which we construct with the help of two RDMA verbs. Fig. 2 shows two stricter ordering modes 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 ordering* (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.

In all the aforementioned ordering modes, 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 then used to explicitly start the prefetching of WRs. This allows for existing WRs to be modified within the WQ, 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 ordering* (Fig. 2b). Doorbell ordering allows developers to modify WR chains in-place. In particular, it allows for *data-dependent, self-modifying* WRs.

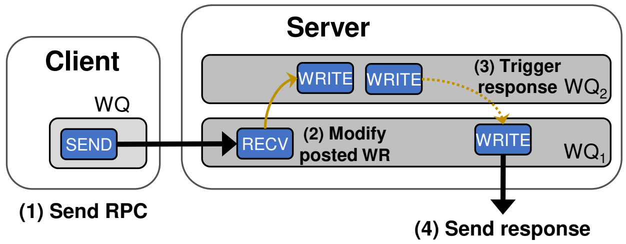


Figure 3: Clients can trigger posted operations. Thick solid lines represent (meta)data movements.

Thus, we have shown that we can control WR fetch and execution via special verbs, which we will exploit in the next section to develop full-fledged RDMA programs. These verbs are widely available in commodity RNICs (e.g., Mellanox terms them cross-channel communication [34]).

3.2 Dynamic RDMA Programs

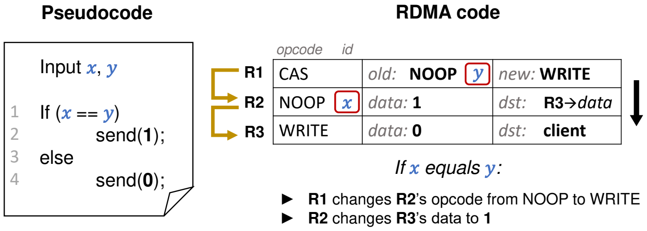
While a static sequence of RDMA WRs is already a rudimentary RDMA program, complex offloads require *data-dependent execution*, where the logic of the offload is dependent on input arguments. To realize data-dependent execution, we construct *self-modifying RDMA code*.

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 host that offloads an RPC handler to its RNIC as shown in Fig. 3. The RPC response depends on the 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 in WQ$_2$ to modify its 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. However, this by itself is not sufficient to provide a Turing complete offload framework.

Turing completeness of RDMA. Turing completeness implies that a system of data-manipulation rules, such as RDMA, are computationally universal. For RDMA to be Turing complete, we need to satisfy two requirements [25]:
T1: Ability to read/write arbitrary amounts of memory.
T2: Conditional branching (e.g. if/else statements).

T1 can be satisfied for limited amounts of memory with regular RDMA verbs, whereas T2 has not been demonstrated with RDMA NICs. However, to truly be capable of accessing an *arbitrary* amount of memory, we need a way of realizing loops. Loops open up a range of sophisticated use-cases and



```
Input x, y
If (x == y)
    send(1);
else
    send(0);
```

R1	CAS	old: NOOP y	new: WRITE
R2	NOOP x	data: 1	dst: R3→data
R3	WRITE	data: 0	dst: client

If x equals y:

- ► R1 changes R2's opcode from NOOP to WRITE
- ► R2 changes R3's data to 1

Figure 4: Simple *if* example and equivalent RDMA code. Conditional execution relies on self-modifying code using CAS to enable/disable WRs based on the operand values.

lower the number of constraints that programmers have to consider for offloads. To highlight their importance, we add them as a third requirement, necessary to fulfill the first:
T3: 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 Conditionals

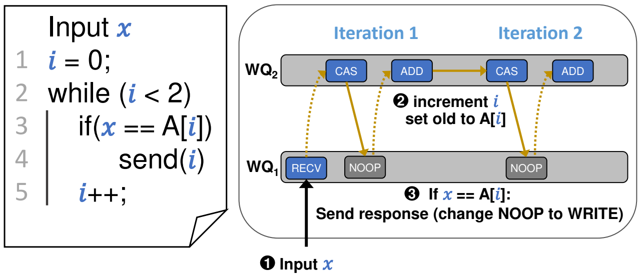
Conditional execution—choosing what computation to perform based on a runtime condition—is typically realized using conditional branches, which are not readily available in RDMA. To this end, we introduce a novel approach that uses self-modifying CAS verbs. The main insight is that this verb can be used to check a condition (i.e., equality of x and y) and then perform a swap to modify the attributes of a WR. We describe how this is done in Fig. 4. We insert a CAS that compares the 64-bit value at the address of R2's *opcode* attribute (initially Noop) with its *old* parameter (also initially Noop). We then set the *id* field of R2 to x. This field can be manipulated freely without changing the behavior of the WR, allowing us to use it to store x. Operand y is stored in the corresponding position in the *old* field of R1. This means that if x and y are equal, the CAS operation will succeed and the value in R1's *new* field—which we set to Write—will replace R2's opcode. Hence, in the case $x = y$, R2 will change from a Noop into a Write operation. This Write is set to modify the *data* value of the return operation (R3) to 1. If x and y are not equal, the default value 0 is returned.

Now that we have established the utility of this technique for basic conditionals, we next look into how to can be used to support loop constructs.

3.4 Loops

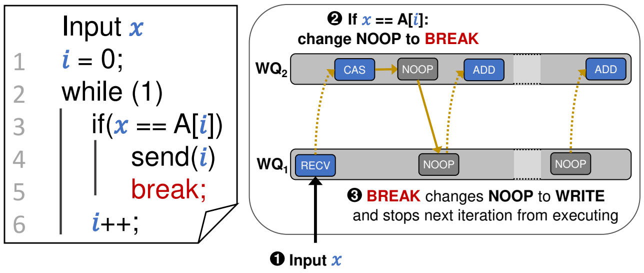
To support loop constructs efficiently, we require (1) conditional branching to test the loop condition and break if necessary, and (2) WR re-execution, to repeat the loop body. We develop each, in turn, below.

Consider the while loop example in Figure 5. This offload searches for x in an array A and sends the corresponding index. The loop is static because A has finite size (in this case, size =2), known a priori. To simplify presentation, consider the case $A[i] = i, \forall i$. Without this simplification, the example would include an additional Write to fetch the value at $A[i]$.



```
Input x
1 i = 0;
2 while (i < 2)
3     if(x == A[i])
4         send(i)
5     i++;
```

Figure 5: while loop using CAS. Loop is unrolled since loop size is fixed and set to 2.



```
Input x
i = 0;
while (1)
    if(x == A[i])
        send(i)
        break;
    i++;
```

Figure 6: *while* loop with breaks realized using CAS. To implement breaks, we use CAS to change a Noop WR to an RDMA Write, which then stops subsequent iterations from executing.

The loop body uses a CAS verb to implement the if condition (line 3), followed by an ADD verb to increment i (line 6). Given that the loop size is known a priori (size = 2), RedN can unroll the while loop in advance and post the WRs for all iterations. As such, there is no need to check the condition at line 2. For each iteration, if the CAS succeeds, the Noop verb in WQ_1 will be changed to Write—which will send the response back to the client. However, it is clear that, regardless of the comparison result, all subsequent iterations will be executed. This is inefficient since, if the send (line 4) occurs before the loop is finished, a number of WRs will be wastefully executed by the NIC. This is impractical for larger loop sizes or if the number of iterations is not known a priori.

Unbounded loops and termination. Figure 6 modifies the previous example to make it such that the loop is unbounded. For efficiency, we add a break that exits the loop if the element is found.The role of break is to prevent additional iterations from being executed. We use an additional Noop that is formatted such that, once transformed into a Write by the CAS operation, it prevents the execution of subsequent iterations in the loop. This is done by modifying the last WR in the loop such that it does not trigger a completion event. The next iteration in the loop, which Waits on such an event (via completion ordering), will therefore not be executed. Moreover, the Write will also modify the opcode of the WR used to send back the response from Noop to Write.

As such, break allows efficient and unbounded loop execution. However, it still remains necessary for the CPU to post WRs to continue the loop after all its WRs are executed. This consumes CPU cycles and can even increase latency if the CPU is unable to keep up with the speed of WR execution.

RedN Constructs		Number of WRs	Operand limit [bits]
if		1C + 1A + 3E	48
while	Unrolled	1C + 1A + 3E	
	Recycled	3C + 2A + 4E	

Table 2: Breakdown of the overhead of our constructs with different offload strategies. *C* refers to copy verbs, *A* refers to atomic RDMA verbs, and *E* refers to Wait/Enable verbs. while loops that use WQ recycling incur 2 additional Reads, 1 ADD, and 1 Enable WR.

Unbounded loops via WQ recycling. To allow the NIC to recycle WRs without CPU intervention, we make use of a novel technique that we call *WQ recycling*. RNICs iterate over WQs, which are circular buffers, and execute the WRs therein. By design, each WR is meant to execute only once. However, there is no fundamental reason why WRs cannot be reused since the RNIC does not actually erase them from the WQ. To enable recycling of a WR chain, we insert a Wait and Enable sequence at the tail of the WQ. This instructs the RNIC to wrap around the tail and re-execute the WR chain for as many times as needed.

It is important to note that WQ recycling is not a panacea. To allow the tail of the WQ to wrap around, all posted Wait and Enable WRs in the loop need to have their *wqe_count* attribute updated. This attribute is used to determine the index of the WR that these ordering verbs affect. In ConnectX NICs, these indices are maintained internally by the RNIC and their values are monotonically increasing (instead of resetting after the WQ wraps around). As such, the *wqe_count* values need to be incremented to match. This incurs overhead (as seen in Table 2) and requires an additional ADD operation in combination with other verbs. As such, loop unrolling, where each iteration is manually posted by the CPU, is overall less taxing on the RNIC. However, WQ recycling avoids CPU intervention, allowing the offload to remain available even amid host software failures (as we will see later in §5.6).

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.

Building blocks. We abstract and parameterize the RDMA chains required for conditional branching and looping into if and while constructs. The overhead in terms of RDMA WR chains of our constructs is shown in Table 2. We can see a breakdown of the minimum number of operations required for each. Inequality predicates, such as $<$ or $>$, can also be supported by combining equality checks with Max or Min, as seen later in Table 3. However, their availability is vendor-specific and currently only supported by ConnectX NICs.

Operand limits. RedN's limit is based on the supported size for the CAS verb, which is 64 bits. The operand is provided as a 48-bit value, encoded in its *id* and other neighboring fields (which can also be freely modified without affecting execution). The remaining bits are 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 number of operations we allocate for our constructs. For instance, despite the 48-bit operand limit for our constructs, we can chain together multiple CAS operations to handle different segments of a larger operand (we do not rely on the atomicity property of CAS). 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 are possible. 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. Code regions are protected by memory keys—special tokens required for RDMA access—upon registration (at connection time), prohibiting unauthorized access. 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. RedN does not solve security challenges in existing RDMA or Infiniband implementations [40]. However, RedN can help RDMA systems become more secure. For such systems, *one-sided* RDMA operations (e.g., RDMA Read and Write) are frequently used [22,28,33,35,42,43] 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 our use-cases in §5, Send operations can be used to trigger offload programs without any CPU involvement.

Isolation. Given that RedN implements dynamic loops, clients can abuse such constructs to consume more than their fair share of resources. Luckily, popular RNICs, like ConnectX, provide WQ rate-limiters [6] 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 monitored and servers can terminate connections to clients running misbehaving code.

Parallelism. RDMA WR fetch and execution latencies are more costly compared to CPU instructions, as WRs are fetched/executed via PCIe (microseconds vs. nanoseconds). As such, to hide WR latencies, it is important to parallelize logically unrelated operations. Like threads of execution in a CPU, each WQ is allocated a single RNIC PU to ensure in-order execution without inter-PU synchronization. As such, we carefully tune our offloaded code to allow unrelated verbs to execute on independent queues to be able to parallelize execution as much as possible. The benefits of parallelism are evaluated in §5.2.

4 Implementation

Our offload framework is implemented in C with ∼2,300 lines of code—this includes our use cases (∼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 RNIC, allowing WRs to be manipulated via RDMA verbs. We configure the ConnectX-5 firmware to allow the WR *id* field to be manipulated freely, which is required for conditional operations as well as WR recycling. This is done by modifying specific configuration registers on the NIC [12].

RedN is compatible with any ConnectX NICs that support WAIT and ENABLE (e.g., ConnectX-3 and later models).

5 Evaluation

We start by characterizing the underlying RNIC performance (§5.1) to understand how it affects our implemented programming constructs. Then, in our evaluation against state-of-the-art RNIC 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 vanilla RDMA offload;
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 programming constructs generic enough to enable a wide-variety of use-cases (§5.2–§5.6);

Testbed. Our experimental testbed consists of 3× dual-socket Haswell servers running at 3.2 GHz, with a total of 16 cores, 128 GB of DRAM, and 100 Gbps dual-port Mellanox ConnectX-5 Infiniband RNICs. All nodes are running Ubuntu 18.04 with Linux Kernel version 4.15 and are connected via back-to-back Infiniband links.

NIC setup. For all of 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 is set to match that of the offloaded program.

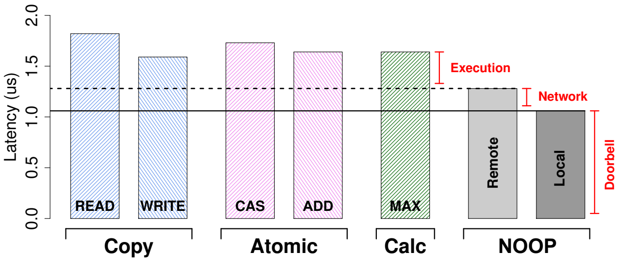


Figure 7: Latencies of different RDMA verbs. The solid line marks the latency of ringing the doorbell via MMIO. The difference between dashed and solid lines estimates network latency.

5.1 Microbenchmarks

We run microbenchmarks to break down RNIC verb execution latency, understand the overheads of our different ordering modes, and determine the processing bandwidth of different RDMA verbs and of our constructs.

5.1.1 RDMA Latency

We break down the performance of RDMA verbs, configured to perform 64B IO, by measuring their average latencies after executing them 100K times. All verbs are executed remotely, unless otherwise stated. As seen in Fig. 7, WRITE has a latency of 1.6 µs. It uses posted PCIe transactions, which are one-way. Comparatively, non-posted verbs such as READ or atomics such as fetch-and-add (ADD) and compare-and-swap (CAS) need to wait for a PCIe completion and take ∼1.8 µs.[2] Overall, the execution time difference is 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 verb execution, 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 local loopback NOOP WRs (shown on the right-hand side) and measuring the difference—roughly 0.25 µs for our back-to-back connected nodes. Overall, these results show low verb execution latency, justifying building more sophisticated functions atop. We next measure the implications of ordering for offloads.

5.1.2 Ordering Overheads

We show the latency of executing chains of RDMA verbs using different ordering modes. All the posted WRs within a chain are NOOP, to simplify isolating the performance impact of ordering. We start by measuring the latency of executing a chain of verbs posted to the same queue but absent any constraints (WQ order), and compare it to the ordering modes

[2] Older-generation NICs (e.g., ConnectX-4) use a proprietary concurrency control mechanism to implement atomics, resulting in higher latencies than later generations that rely on PCIe atomic transactions.

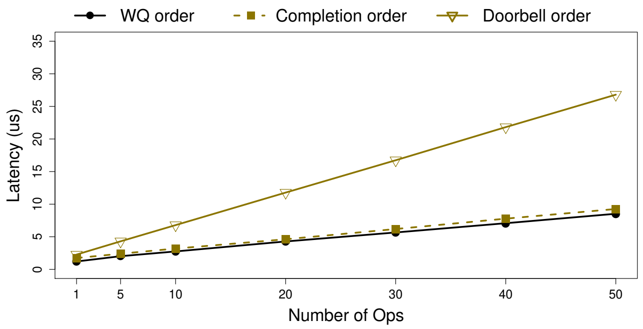


Figure 8: Execution latency of RDMA verbs posted using different ordering modes. More restrictive modes such as Doorbell order add non-negligible overheads as it requires the NIC to fetch WRs sequentially.

that we introduced in Fig. 2—completion order and doorbell order. WQ order only mandates in-order updates to memory, which allows for increased concurrency. Operations that are not modifying the same memory address can execute concurrently and the RNIC is free to prefetch multiple WRs with a single DMA[3]. We can see in Fig. 8 that the latency of a single NOOP is 1.21 µs and the overhead of adding subsequent verbs is roughly 0.17 µs per verb. The first verb is slower since it requires an initial doorbell to tell the NIC that there is outstanding work. For completion ordering, less concurrency is possible since WRs await the completions of their predecessors, and the overhead of increases slightly to 0.19 µs per additional WR. For doorbell order, no latency-hiding is possible, as the NIC has to fetch WRs from memory one-by-one, which results in an overhead of 0.54 µs per additional WR. These results signify that, doorbell ordering should be used conservatively, as there is more than 0.5 µs latency increase for every instance of its use, compared to more relaxed ordering modes.

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 of RedN's if and while constructs. Using 48-bit operands, a ConnectX-5 NIC can execute 700K if constructs per second. This is due to the need for CAS to ensure doorbell ordering between CAS and the subsequent WR it modifies. This causes the throughput to be bound by NIC processing limits. Unrolled while loops require the same number of verbs per iteration as an if statement and their throughput is identical. while loops with WQ recycling have reduced performance due to having to execute more WRs per iteration.

Operation			Throughput (M ops/s)	Support
Atomic	CAS		8.4	Native
	ADD			
Copy	READ		65	
	WRITE		63	
Calc	MAX		63	Mellanox
Constructs	if		0.7	RedN
	while	Unrolled	0.7	
		Recycled	0.3	

Table 3: Throughput of common RDMA verbs and RedN's constructs on a single port of a ConnectX-5. if and unrolled while have identical performance. while loops with WQ recycling require additional WRs and therefore have a lower throughput.

5.2 Offload: Hash Lookup

After evaluating the overheads of RedN's ordering modes and constructs, we next look into the performance of RedN for offloading remote access to popular data structures. We first look into hash tables, given their prominent use in key-value stores for indexing 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. To support dynamic value sizes, we assume the value is not inlined in the bucket and is instead referenced via a pointer.

For distributed key-value stores built with 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 network round-trips at a minimum. This greatly increases latency but does not require involvement of the server's CPU. Many systems utilize this approach to implement lookups, including FaRM [22] and Pilaf [35].

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 [19, 26] approach follows traditional RPC implementations and avoids the need for several roundtrips. However, this comes at the cost of server CPU cycles.

5.2.1 RedN's Approach

To offload key-value *get* operations, we leverage the offload schemes introduced in §3.3 and §3.4.

Fig. 9 describes the RDMA operations involved for a single-hash lookup. To *get* a value corresponding to a key, the client first computes the hashes for its key. For this use-case, we set the number of hashes to two, which is common in practice [24]. The client then performs a SEND with the value of

[3] The number of operations fetched by the RNIC can change dynamically. The Prefetch mechanism in ConnectX RNICs is proprietary.

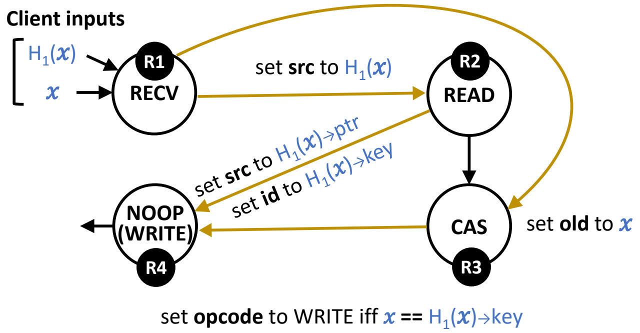

Client inputs
$H_1(x)$
x
R1
RECV
set **src** to $H_1(x)$
R2
READ
set **src** to $H_1(x)$→ptr
set **id** to $H_1(x)$→key
NOOP (WRITE)
R4
CAS
R3
set **old** to x
set **opcode** to WRITE iff x == $H_1(x)$→key

Figure 9: Hash lookup RDMA program. Black arrows indicate order of execution of WRs in their WQs. Brown arrows indicate self-modifying code dependencies and require doorbell ordering. x is the requested key and $H_1(x)$ is its first hash. The acronym *src* indicates the "source address" field of WRs. *old* indicates the "expected value" at the target address of the CAS operation. The *id* field is used for storing conditional operands.

the key x and address of the first bucket $H_1(x)$, which are then captured via a RECV WR posted on the server. The RECV WR (R1) inserts x into the *old* field of the CAS WR (R3) and the bucket address $H_1(x)$ into the READ WR (R2). The READ WR retrieves the bucket and sets the source address (*src*) of the response WR (R4) to the address of the value (*ptr*). It also inserts the bucket's key into the *id* field to prepare it for the conditional check. Finally, CAS (R3) checks whether the expected value *old*, which is set to key x, matches the *id* field in (R4), which is set to the bucket's *key*. If equal, (R4)'s opcode is changed from NOOP to WRITE, which then returns the value from the bucket. Given that each key may be stored in multiple buckets (two in our setup), these lookups may be performed sequentially or in parallel, depending on the offload configuration.

5.2.2 Results

We evaluate our approach against both one-sided and two-sided implementations of key-value *get* operations. We use FaRM's approach [22] 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, our RPC to the host involves a client-initiated RDMA SEND to transmit the *get* request, and an RDMA WRITE initiated by the server to return the value after performing the lookup.

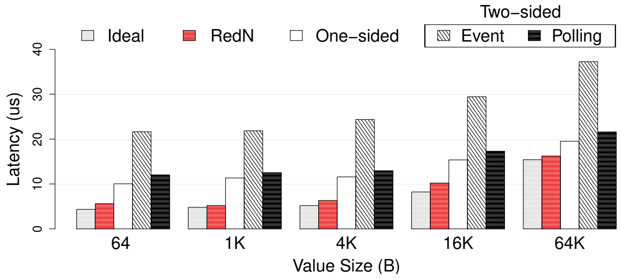


Figure 10: Average latency of hash lookups. *Ideal* shows the latency of a single network round-trip READ.

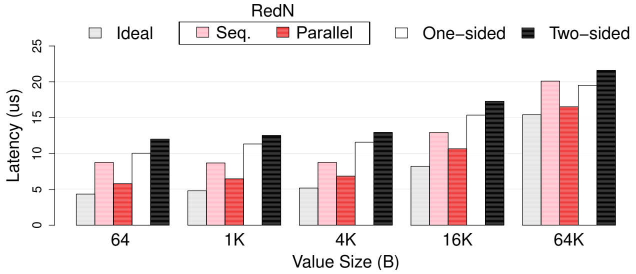


Figure 11: Average latency of hash lookups during collisions. *Ideal* shows the latency of a single network round-trip READ.

Latency. Fig. 10 shows a latency comparison of KV *get* operations of RedN against one-sided and two-sided baselines. We evaluate two distinct variations of two-sided. The *event-based* approach blocks for a completion event to avoid wasting CPU cycles, whereas the *polling-based* approach dedicates one CPU core for polling the completion queue. We use 48-bit keys and vary the value size. The value size is given on the x-axis. In this scenario, we assume no hash collisions and that all keys are found in the first bucket. RedN is able to outperform all baselines — fetching a 64 KB key-value pair in 16.22 µs, which is within 5% of a single network round-trip READ (Ideal). RedN is able to deliver close-to-ideal performance because it bypasses the server's CPU ***and*** fetches the value in a single network RTT. Compared to RedN, one-sided operations incur up to 2× higher latencies, as they require two RTTs to fetch a value. Two-sided implementations do not incur any extra RTT; however, they require server CPU intervention. The polling-based variant consumes an entire CPU core but provides competitive latencies. Event-based approaches block for completion events to avoid wasting CPU cycles and incur much higher latencies as a consequence. RedN is able to outperform polling-based and event-based approaches by up to 2 and 3.8×, respectively. Given the much higher latencies of event-based approaches, for the remainder of this evaluation, we will only focus on polling-based approaches and simply refer to them hereafter as *two-sided*.

Fig. 11 shows the latency in the presence of hash collisions. In this case, we assume a worst case scenario, where the key-value pair is always found in the second bucket. In this scenario, we introduce two offload variants for RedN— RedN-Seq & RedN-Parallel. The former performs bucket lookups sequentially within a single WQ. The latter parallelizes bucket lookups by performing the lookups across two different WQs to allow execution on different NIC PUs. We can see that RedN-Parallel maintains similar latencies to lookups with no hash collisions (i.e., *RedN* in Fig. 10), since bucket lookups are almost completely parallelized. It is worth noting that parallelism in this case does not cause unnecessary data movement, since the value is only returned when the corresponding

Hash lookup	IO Size			
	$\leq$ 1 KB		64 KB	
Port config.	Single	Dual	Single	Dual
Rate (ops/s)	500K	1M	180K	190K
Bottleneck	NIC PU		IB bw	PCIe bw

Table 4: NIC throughput of hash lookups and its bottlenecks.

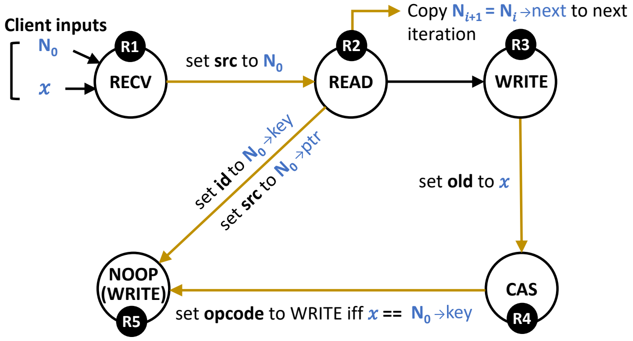


Figure 12: Linked list RDMA program.

key is found. For the other bucket, the WRITE operation (R4 in Fig. 9) is a NOOP. RedN-Seq, on the other hand, incurs at least 3 µs of extra latency as it needs to search the buckets one-by-one. As such, whenever possible, operations with no dependencies should be executed in parallel. The trade-off is having to allocate extra WQs for each level of parallelism.

Throughput. We describe our throughput in Table 4. At lower IO, RedN is bottlenecked by the NIC's processing capacity due to the use of doorbell ordering—reaching 500K ops/s on a single port (1M ops/s with dual ports). At 64 KB, RedN reaches the single-port IB bandwidth limit (~ 92 Gbps). Dual-port configs are limited by ConnectX-5's 16× PCIe 3.0 lanes.

SmartNIC comparison. We compare our performance for hashtable *gets* against StRoM [39], a programmable FPGA-based SmartNIC. Since we do not have access to a programmable FPGA, we extract the results from [39] for comparison, and report them in Table 5. RedN uses the same experimental settings as before. 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 lookup latencies than StRoM. StRoM uses a Xilinx Virtex 7 FPGA, which runs at 156.25 MHz, and incurs at least two PCIe roundtrips to retrieve the key and value. Our evaluation shows that RedN can provide latency that is in-line with more expensive SmartNICs.

IO Size	System	Median	99^{th}ile
64 B	RedN	5.7 µs	6.9 µs
	StRoM	~7 µs	~7 µs
4 KB	RedN	6.7 µs	8.4 µs
	StRoM	~12 µs	~13 µs

Table 5: Latency comparison of hash *gets*. Results for StRoM obtained from [39].

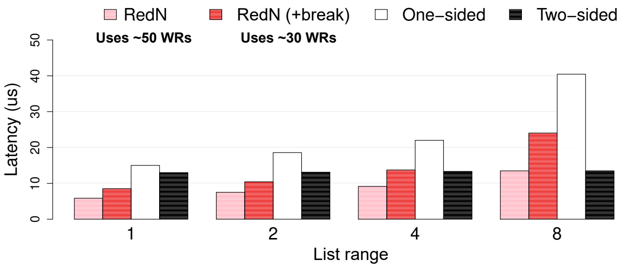


Figure 13: Average latency of walking linked lists.

5.3 Offload: 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 [22, 35].

Linked list processing can be decomposed into a *while* loop for traversing the list and an *if* condition for finding and returning the key. We describe the implementation of our offload in Fig. 12. The client provides the key x and address of the first node in the list N_0. A READ operation (R2) is then performed to read the contents of the first node and update the values for the return operation (R5). We also use a WRITE operation (R3) to prepare the CAS operation (R4) by inserting key x in its *old* field. As an optimization, this WRITE can be removed and, instead, x can be inserted directly by the RECV operation. This, however, will need to be done for every CAS to be executed and, as such, this approach is limited to smaller list sizes, since RECVs can only perform 16 scatters.

For this use-case, we introduce two offload variations. The first, referred to simply as RedN, uses the implementation in Fig. 12. The second uses an additional *break* statement between R4 and R5 to exit the loop in order to avoid executing any additional operations.

5.3.1 Results

Fig. 13 shows the latency of one-sided and two-sided variants against RedN at various linked list ranges — where range represents the highest list element that the key can be randomly placed in. The size of the list itself is set to a constant value of 8. We setup the linked list to use key and value sizes of 48 bits and 64 bytes, respectively, and perform 100k list traversals for each system. The requested key is chosen at random for each RPC. In the variant labelled "RedN", we do not use *breaks* and assume that all 8 elements of the list need to be searched. RedN outperforms all baselines for all list ranges until 8 — providing up to a 2× improvement. *RedN (+break)* executes a break statement with each iteration and performs worse than RedN due to the extra overhead of checking the condition of the *break*. However, using a break statement increases the offload's overall efficiency since no unneeded iterations are executed after the key is found — using an average of 30 WRs across all experiments. Without breaks, RedN will need to execute all subsequent iterations even after the key-value

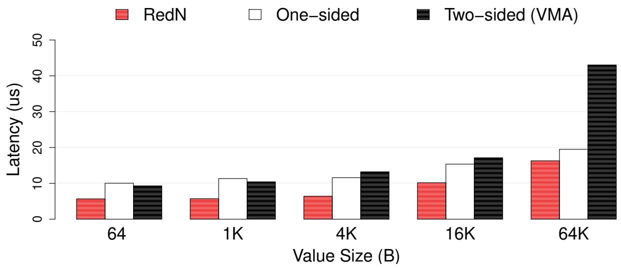


Figure 14: Memcached *get* latencies with different IO sizes.

pair is found/returned and it uses more than 65% more WRs. As such, while RedN is able to provide better latencies, using a break statement is more sensible for longer lists.

5.4 Use Case: Accelerating Memcached

Based on our earlier experience offloading remote data structure traversals, we set out to see: 1) how effective our aforementioned techniques are in a real system, and 2) what are 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. We use a version of Memcached that employs cuckoo hashing [24]. Since Memcached does not natively support RDMA, we modify it with ∼700 LoC to integrate RDMA capabilities, allowing the RNIC to register the hash table and storage object memory areas. We also modify the buckets, so that the addresses to the values are stored in big endian — to match the format used by the WR attributes. We then use RedN to offload Memcached's *get* requests to allow them to be serviced directly by the RNIC without CPU involvement. We 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 [9]—a kernel-bypass userspace TCP/IP stack that boosts the performance of sockets-based applications by intercepting their socket calls and using kernel-bypass to send/receive data. We configure VMA in polling-mode to optimize for latency. In addition, we also implement a one-sided approach, similar to the one introduced in section 5.2.

Fig. 14 shows the latency of *gets*. As we can see, RedN's offload for hash *gets* is up to 1.7× faster than one-sided and 2.6× faster than two-sided. Despite the latter being configured in polling-mode, VMA incurs extra overhead since it relies on a network stack to process packets. In addition, to adhere to the sockets API, VMA has to memcpy data from send and receive buffers, further inflating latencies—which is why it performs comparatively worse at higher value sizes.

5.5 Use Case: Performance Isolation

One of the benefits of exposing the latent turing power of RNICs is to enforce isolation among 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 writer (clients). These writers generate *set* RPCs in a closed loop to load the Memcached service. At the same time, we use a single reader client to generate *get* operations. To stress CPU resources while minimizing lock contention, each reader/writer is assigned a distinct set of 10K keys, which they use to generate their queries. The keys within each set are accessed by the clients sequentially.

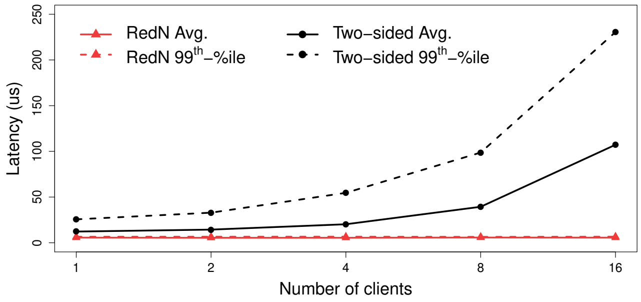


Figure 15: Memcached *get* latencies under hardware contention with varying numbers of writer-clients.

We can see in Fig. 15 that, as we increase the # of writers, both the average and 99^{th} percentile latencies for two-sided increase dramatically. For RedN, CPU contention has no impact on the performance of the RNIC and both the average and 99^{th} percentiles sit below 7 µs. At 16 writers, RedN's 99^{th} percentile latency is 35× lower than the baseline.

This indicates that RNIC offloads can also have other useful 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

We now consider server failures and how failure is affected by RNICs. Table 6 shows failure rates of server software and hardware components. NICs are much less likely to fail than software components—NIC annualized failure rate (AFR) is an order of magnitude lower. Even more importantly, NICs are partially decoupled from their hosts 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 that 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 records, etc.) that are required for functionality. If the process hosting these resources crashes, the memory belonging to

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 [8, 37]. AFR means annualized failure rate, whereas MTTF stands for mean time to failure and is expressed in hours. RNICs can still access memory even in the presence of an OS failure.

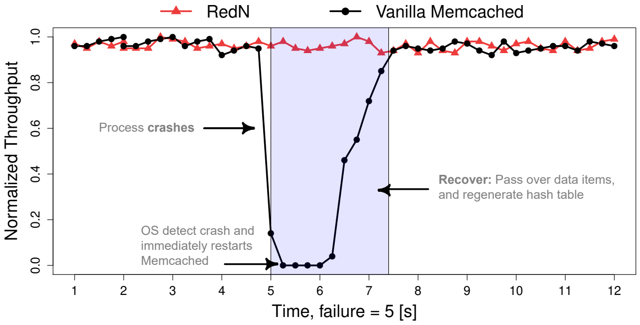


Figure 16: RedN can survive process crashes and continue serving RPCs via the RNIC without interruption.

these components will be automatically freed by the operating system resulting in termination of the RDMA program. To counteract this, we use [38] 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. 16) where we send *get* queries to a single instance of Memcached and then simply kill Memcached during the run. The OS 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 without recovery time.

OS failure. We also programmatically induce a kernel panic using sysctl, freezing the system. This is a simpler case than process crashes, since we no longer have to worry about the OS freeing RDMA resources. For brevity, we do not show these results, but we experimentally verified that RedN offloads continue operating in the presence of an OS crash.

6 Discussion

Client scalability. RedN requires servers to manage at least two WQs per client, which is not higher than other RDMA systems. RedN can still introduce scalability challenges with thousands of clients since RNIC cache is limited. However, Mellanox's dynamically-connected (DC) transport service [5], which allows unused connections to be recycled, can circumvent many such scalability limits.

Offload for sockets-based applications. Protocols such as rsocket [16] can be used to transparently convert sockets-based applications to use RDMA, making them possible targets for RedN. Although rsocket does not support popular system calls, such as epoll, other extensions have been proposed [29] that support a more comprehensive list of system calls and were shown to work with applications like Memcached and Redis.

Intel RNICs. Next-generation Intel RNICs are expected to support atomic verbs, such as CAS—which RedN uses to implement conditionals. To control when WRs can be fetched by the NIC, Intel uses a validity bit in each WR header. This bit can be dynamically modified via an RDMA operation to mimic ENABLE. However, there is no equivalent for the WAIT primitive, meaning that clients cannot trigger a pre-posted chain. One possible workaround for this is to use another PCIe device on the server to issue a doorbell to the RNIC, allowing the WR chain to be triggered. We leave the exploration of such techniques as future work.

Insights for next-generation RNICs. Our experience with RedN has shown that keeping WRs in server memory (to allow them to be modified by other RDMA verbs) is a key bottleneck. If the NIC's cache was made directly accessible via RDMA, WRs can be pre-fetched in advance and unnecessary PCIe round-trips on the critical path can be avoided. We hope future RNICs will support such features.

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 2.6× and 35× speed-up versus state-of-the-art RDMA approaches, for key-value get operations under uncontended and contended settings, 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.
RedN is available at https://redn.io.

Acknowledgements. This work has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 770889), as well as NSF grant 1751231. Waleed Reda was supported by a fellowship from the Erasmus Mundus Joint Doctorate in Distributed Computing (EMJD-DC), funded by the European Commission (EACEA) (FPA 2012-0030). We would like to thank Gerald Q. Maguire Jr. and our anonymous reviewers for their comments and feedback as well as Jasmine Murcia. Thanks also go to our shepherd Ang Chen.

References

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

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

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

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

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

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

[7] 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.

[8] 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.

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

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

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

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

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

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

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

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

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

[18] 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.

[19] 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.

[20] 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.

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

[22] 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.

[23] 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.

[24] 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.

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

[26] 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.

[27] 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.

[28] 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.

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

[30] 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.

[31] 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.

[32] 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.

[33] 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.

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

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

[36] 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.

[37] 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.

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

[39] D. Sidler, Z. Wang, M. Chiosa, A. Kulkarni, and G. Alonso. StRoM: Smart Remote Memory. *Proceedings of the Fifteenth EuroSys Conference*, 2020.

[40] 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.

[41] 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, 2017.

[42] 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.

[43] 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.

[44] 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.

[45] 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, 2018.

[46] 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, 2019.

Appendix A Turing completeness sketch

To show 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 constructs 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 [21] 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 the `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 [21] 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 ordering (refer to §3.1 for a discussion of our ordering 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 ordering, 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 [21] 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 WAIT and ENABLE to recycle RDMA WRs (as described in §3.4). 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 [21] for simulating a Turing machine.

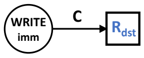
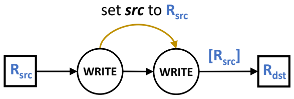
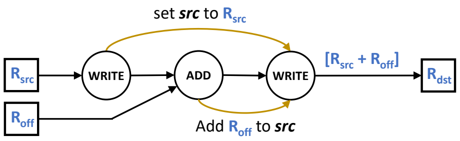
Addressing mode	**x86 syntax**	**RedN equivalent**
Immediate	mov R_{dst}, C	
Indirect	mov R_{dst}, $[R_{src}]$	 set *src* to R_{src} R_{src} WRITE WRITE $[R_{src}]$ R_{dst}
Indexed	mov R_{dst}, $[R_{src} + R_{off}]$	 set *src* to R_{src} R_{src} R_{off} WRITE ADD WRITE $[R_{src} + R_{off}]$ R_{dst} Add R_{off} to *src*

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