HatRPC: Hint-Accelerated Thrift RPC over RDMA

Tianxi Li*
The Ohio State University
Columbus, USA
li.9443@buckeyemail.osu.edu

Haiyang Shi*
The Ohio State University
Columbus, USA
shi.876@buckeyemail.osu.edu

Xiaoyi Lu
University of California, Merced
Merced, USA
xiaoyi.lu@ucmerced.edu

ABSTRACT

In this paper, we propose a novel hint-accelerated Remote Procedure Call (RPC) framework based on Apache Thrift over Remote Direct Memory Access (RDMA) protocols, called *HatRPC*. *HatRPC* proposes a hierarchical hint scheme towards optimizing heterogeneous RPC services and functions. The proposed hint design is composed of service-granularity and function-granularity hints for achieving varied optimization goals and reducing design space for further optimizing the underneath RDMA communication engine. We co-design a key-value store called HatKV with *HatRPC* and LMDB. The effectiveness and efficiency of *HatRPC* are validated and evaluated with our proposed Apache Thrift Benchmarks (ATB), YCSB, and TPC-H workloads. Performance evaluations show that the proposed *HatRPC* approach can deliver up to 55% performance improvement for ATB benchmarks and up to 1.51× speedup for TPC-H queries compared with vanilla Thrift over IPoIB. In addition, the co-designed HatKV can achieve up to 85.5% improvement for YCSB workloads.

CCS CONCEPTS

• **Networks** → **Network protocols**; • **Computer systems organization** → **Architectures**.

KEYWORDS

RDMA, Hint, Code Generation, RPC, Thrift

ACM Reference Format:
Tianxi Li*, Haiyang Shi*, and Xiaoyi Lu. 2021. *HatRPC*: Hint-Accelerated Thrift RPC over RDMA. In *The International Conference for High Performance Computing, Networking, Storage and Analysis (SC '21), November 14–19, 2021, St. Louis, MO, USA*. ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3458817.3476191

1 INTRODUCTION

With the convergence of High-Performance Computing (HPC), Big Data, and Artificial Intelligence, more and more data center applications have started leveraging modern HPC technologies to accelerate the performance of their codes. Along with this direction, high-performance networking technologies, such as Remote Direct Memory Access (RDMA), have been widely adopted in many data center applications recently, such as RPC [40], key-value stores [36, 46, 54, 65], distributed file systems [29, 31, 32], database systems [31], and deep learning frameworks [18, 34, 42, 50], etc.

RDMA has been proven that it can deliver high throughput, low latency, and low CPU utilization to data center applications [32, 36, 50]. However, one of the major burdens of using RDMA is its low productivity for application development. The defacto standard of programming with RDMA is by using a user-space communication library, called *verbs* [45]. *verbs* provides many low-level network primitives, such as *post_send*, *post_recv*, *poll_cq*, etc. These low-level primitives can give a lot of flexibility for the applications to design efficient communication protocols, while it typically requires the developers to spend much effort and time on programming, debugging, and tuning the code for achieving high performance and scalability. For instance, based on our experience, even for a simple communication program with *verbs* (like *hello world on verbs*), we need to write around 600 lines of code (LOC). Unified Communication X (UCX) [7] has been proposed to ease the programming with RDMA. It includes various transports and has more understandable APIs than verbs. However, the hello world example [11] given by the library repository is still more than 500 LOC. Hence, UCX does not ease users' RDMA programming a lot. Such a situation motivates us to rethink a fundamental challenge facing the RDMA community: *Can we propose an approach which can allow applications to use native RDMA-based communication protocols while significantly improve application developers' productivity?* To address this challenge, this paper further investigates the following research problems.

1. Can we design an approach to automatically generate efficient RDMA-based communication substrates for data center applications? Since directly programming with *verbs* is very difficult, this inspires us to explore whether we can generate RDMA-based communication substrates fully or partially for data center applications. If so, applications can achieve both high performance and high productivity.

2. How can the proposed approach satisfy different communication requirements on various RDMA protocols in data center applications? Datacenter applications typically need supports for different performance goals (like latency-sensitive or throughput-sensitive), varied payload sizes, unpredictable concurrences, etc. These are all significant challenges to make such an efficient code generation approach become viable.

3. How can we guarantee the effectiveness and efficiency of the generated RDMA-based communication protocols for heterogeneous data center applications? If such a code generation approach becomes available, we still need to validate and

*Tianxi Li and Haiyang Shi contributed equally to this work.

SC '21, November 14–19, 2021, St. Louis, MO, USA

ACM ISBN 978-1-4503-8442-1/21/11…$15.00
https://doi.org/10.1145/3458817.3476191

evaluate whether the generated RDMA-based communication substrates can effectively and efficiently support end applications.

After we seriously explored the answers for the above-mentioned problems, we have soon realized that it could be impossible or very difficult to design a generic code generation approach for generating and optimizing RDMA communication substrates in all kinds of data center applications. However, through our investigations, we find that if we can leverage some existing widely used code generation frameworks in data center systems and applications, we can significantly reduce the complexity of these research problems.

In this paper, we find that Apache Thrift [58] can become a promising code generation framework for achieving our objectives to a great extent. First of all, Apache Thrift is an open-source Remote Procedure Call (RPC) framework. RPC is the most widely used communication mechanism in distributed data center applications. RPC is very flexible and productive so that it can easily satisfy different communication requirements in data center applications.

Second, Apache Thrift has a well-modularized architecture, which contains a communication transport layer together with a compiler to generate modules in support of connection establishment and message passing. Even though default Apache Thrift does not have the native RDMA support in its transport layer yet, the modularized architecture of Thrift allows us to extend it to enable RDMA-based communication.

Thirdly, Apache Thrift provides an extensible code generation framework based on flex [13] and Bison [2], which can be extended to support customized code generation approaches.

Based on these observations, this paper proposes a novel code generation approach based on Apache Thrift RPC library for automatically generating and optimizing RDMA-based communication substrates via a hierarchical hint scheme. We call it *HatRPC*, which stands for Hint-Accelerated Thrift RPC framework over RDMA.

The key ideas of the *HatRPC* approach include: 1) Proposing a hierarchical hint scheme towards optimizing heterogeneous RPC services and functions. To achieve varied optimization goals and reduced design space for further optimizing the underneath RDMA communication engine, the desired hint scheme is shown in Figure 1. The hierarchical hint scheme is composed of vertical hints (i.e., *Service Level Hints* and *Function Level Hints*) and lateral hints (i.e., *Server Hints* and *Client Hints*). These hint granularities are necessary to support heterogeneous RPC services/functions and optimization isolation. 2) Proposing a new abstraction, named *TRdma*, as a bridge layer for RPC engine and underneath RDMA engine. We intentionally keep the programming model of *TRdma* fully compatible with that of *TSocket*, which is the default programming interface for Socket-based communication in Apache Thrift. With this idea, the code generator can reuse essential codes in Apache Thrift for both *TSocket* and *TRdma*, which can significantly reduce the design and development complexity of *HatRPC*. 3) Extensively examining state-of-the-art RDMA communication protocols for different communication scenarios. Based on our analysis, we are able to create a novel mapping between user-input hints to low-level RDMA protocols.

In general, different kinds of RDMA-based communication engines can be plugged into *HatRPC*. The unique feature of our proposed RDMA engine under *HatRPC* is its capability of supporting hint-aware RDMA protocols.

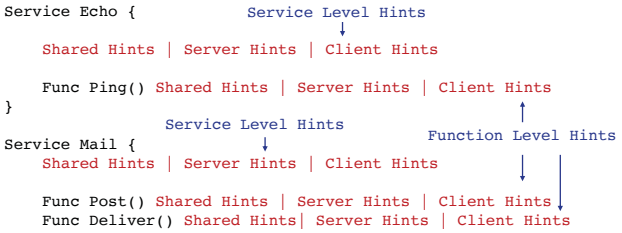


Figure 1: RPC IDL File with Desired Hint Scheme

To demonstrate the usability of *HatRPC*, we further co-design a key-value (KV) store, called HatKV, based on *HatRPC* and LMDB [1]. Then, we extend the YCSB [22] benchmarking framework to support HatKV. The effectiveness and efficiency of *HatRPC* are validated and evaluated with our proposed Apache Thrift Benchmarks (ATB), YCSB, and TPC-H [61] workloads. ATB contains benchmarks for evaluating RPC latency, throughput, and workloads with service- or function-level hints. All ATB benchmarks are developed based on the generated code skeletons by *HatRPC*.

Performance evaluations show that the proposed *HatRPC* approach can deliver up to 55% performance improvement for ATB benchmarks and up to 1.51× speedup for TPC-H queries with 1TB input data size, compared with vanilla Thrift running over a 10-node InfiniBand EDR (100 Gbps) cluster. HatKV can also achieve up to 85.5% performance improvement in the YCSB evaluations.

We present in this paper the following contributions:

- We analyze nine state-of-the-art RDMA protocols and demonstrate the requirements of hints for accelerating RDMA protocols.
- We propose *HatRPC*, enabling Apache Thrift RPC framework over RDMA, which is further accelerated by the proposed hierarchical hint scheme. *HatRPC* supports generating RDMA communication substrates for heterogeneous data center applications, which can significantly ease the RDMA programming and usage.
- We present a co-design example (i.e., HatKV) and detailed evaluations with various workloads to validate the effectiveness and efficiency of *HatRPC*.

To the best of our knowledge, this is the first work to propose a hierarchical hint scheme towards achieving various optimization goals for heterogeneous RPCs over RDMA. Also, this is the first attempt to explore a practical code generation approach for RDMA.

This paper is organized as follows. Section 2 provides background about the Apache Thrift and RDMA for data center applications. Section 3 analyzes state-of-the-art RDMA protocols along with their characteristics. Section 4 elaborates on our *HatRPC*, starting from the hint design, code generation to our RDMA communication engine and a co-designed KV store. Section 5 presents the detailed evaluation methodology and results. Section 6 lists the related work and Section 7 concludes this work.

2 BACKGROUND

This section presents some necessary background information.

Overview of Apache Thrift: Apache Thrift is an open-source Remote Procedure Call (RPC) framework. The framework adopts a

protocol stack together with a compiler to generate modules in support of connection establishment and message passing. It is portable across several platforms and supports multiple languages including: C, C++, Java, Perl, PHP, Python, Ruby, Rust, Swift, etc. Thrift is advantageous in generating templates thus the users can write their logic without concerns about the low-level implementations.

As shown in Figure 2, each layer in the hierarchy has multiple options tuned for different needs. The Protocol layer is for serializing and deserializing the messages across different architectures. Transport Wrapper is an enhanced layer boosting the performance of the communication. The Low-Level Transport layer implements how the data will be sent and received. Because of these many advantages, Thrift has been widely used in data center applications.

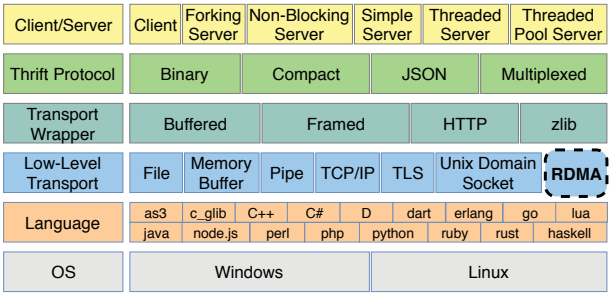


Figure 2: Overview of Thrift Architecture

RDMA for Data Center Applications: RDMA (Remote Direct Memory Access) allows one node to access another node's memory remotely without the involvement of the operating system. It utilizes several techniques that aim to increase the throughput and cut the latency. Zero-copy can eliminate the cost of extra copies of data from userspace to operating systems, which is helpful in improving the performance. In addition, the RDMA communication can bypass the kernel of both the source and the destination, reducing the overhead of the context switch. In contrast to the excellent performance it provides, the RDMA code is hard to write. Even a simple demo will have hundreds of lines in comparison to the ease of programming with POSIX Sockets. Since RDMA and Thrift are both very popular in designing distributed systems, integrating the feature of RDMA into Thrift seems inviting and promising, which motivates us investigating along with this research direction.

3 ANALYSIS OF STATE-OF-THE-ART RDMA PROTOCOLS

In recent years, there have been many research works in the community to take advantage of the ultra-low latency and high bandwidth of RDMA in designing high-performance communication schemes. Many design approaches and guidelines have been proposed by both industry and academia. However, there is no one-size-fits-all approach for various applications. Even small differences in application requirements significantly affect the performance of different design approaches. For instance, [37] shows that a general-purpose RPC design that performs best for a key-value store delivers lower scalability and 16% lower throughput than the best design for a networked sequencer. Therefore, even a well-tuned RPC framework in a complicated distributed system could not deliver optimized performance to all kinds of RPCs. Fortunately, we come up with *HatRPC*, a hint-accelerated RPC over RDMA design, to address this challenging problem. In this section, we first recall some representative design choices for RDMA-accelerated RPC frameworks and then elaborate on the methodology in designing *HatRPC*.

3.1 RDMA Protocols

We evaluate different RDMA protocols with RPC-like workloads, which transfer fix-sized messages between client(s) and a server. For each iteration in the experiments, client(s) send message(s) to the server. Once arrived at the server-side, messages are processed by the server and the results are either sent back to the client by the server or fetched back from the server by the corresponding client. The experimental setup for the evaluations in this section can be found in Section 5.1.

Figure 3 illustrates typical implementations of nine representative RDMA protocols. Eager protocol (i.e., Eager-SendRecv in Figure 3a) is a widely-used and well-studied RDMA protocol [41, 47, 62]. With the Eager protocol, each buffer slot of the pre-registered circular buffers for serving control messages has extra room to carry out a data payload. Such that a payload is able to be sent out together with a control message in the same trip. However, the data payload has to be copied from a user buffer to a slot of the pre-posted circular buffers. Consequently, we usually use Eager protocol for small messages to avoid expensive memory copy overhead and reduce memory footprint. Rendezvous protocols are therefore designed to efficiently transmit large data payloads. To deliver a data payload by the rendezvous protocol, the initiator and the target have to exchange the metadata of the payload. Thus, the target can either prepare a buffer that the initiator can write to or fetch the payload from the initiator. There are mainly two types of rendezvous protocols. RDMA WRITE-based rendezvous protocol (i.e., Write-RNDV in Figure 3d) has been used in MPI for decades [39, 62] because of its capability of reducing memory footprint and improving scalability. RDMA READ-based rendezvous protocol [41, 60, 62] (i.e., Read-RNDV in Figure 3e) is an alternative to Write-RNDV protocol as it also delivers optimized memory utilization and scalability.

Figures 3b, 3c, and 3f show three RDMA protocols that are designed for further optimizing performance. Direct-Write-Send protocol [66] in Figure 3b uses an RDMA WRITE to write data to a pre-known, pre-registered message buffer on the remote side and an RDMA SEND to notify the peer the existence of an available message in the pre-known message buffer. Chained-Write-Send protocol [25, 36, 37] in Figure 3c adopts the same idea as Direct-Write-Send except that it chains the successive RDMA WRITE and SEND as one chained Work Request (WR) to reduce Memory-Mapped I/O (MMIO) over the PCIe bus. Another alternative approach is as shown in Figure 3f, Direct-WriteIMM protocol [25, 43] uses an RDMA WRITE_WITH_IMM to replace the chained RDMA WRITE and SEND in Chained-Write-Send protocol. These protocols are useful and performant in some use cases, but all of them require the remote peer has a pre-known and pre-registered message buffer that is reserved for each connection. It is easy to reason that the reserved message buffer is not feasible to serve all message sizes and needs to protect against being overwritten.

In recent years, there have been many research works focusing on offloading some portion of workloads to clients by leveraging RDMA READ. Figures 3g–3i depict three pre-known designs towards achieving server-bypass. Pilaf [46] (a key-value store) uses ∼3.2 RDMA READs for each GET request on average even with read-heavy workloads [59]. Therefore, we use three RDMA READs (two READs for fetching metadata and one READ for getting payload) to study its performance implications for RPC-like communication. Similarly, FaRM [23] (a key-value store) needs ⩾2 RDMA READs per GET (at least one READ for fetching the index entry, and one for fetching the value) [37], while RFP [59], an RDMA-based RPC paradigm, claims that it usually gets the whole data and metadata with one single RDMA READ. Therefore, we implement FaRM and RFP accordingly as shown in Figures 3h and 3i, respectively.

3.2 Performance Characterization

Our comprehensive evaluations on these representative RDMA protocols with RPC-like workloads present that they have varied characteristics and performance implications. We typically elaborate the evaluation results from three perspectives: performance (i.e., latency and throughput), concurrency, and resource utilization. The goal of the following analyses is to clarify how we can use hints to narrow down design space for designing RDMA-accelerated RPC frameworks and why the proposed hint design is able to meet the desired requirements of RPC frameworks.

As shown in Figures 4 and 5, we vary message sizes, number of clients, and Completion Queue (CQ) polling mechanisms to study their impact on performance, concurrency, and resource utilization. RDMA provides two polling mechanisms to check for the availability of new messages. The first approach is to busily poll NIC to get new completion notifications. While busy polling guarantees low latency it also consumes many CPU cycles. The second mechanism uses events to signal new completion notifications. The NIC raises an interrupt when a new message is available and wakes threads that are waiting for this event. [51] shows that, compared with busy polling, the event polling mechanism reduces the CPU overhead to ∼4% for a full-speed bidirectional transfer with 512KB messages at the cost of a relatively higher latency. In terms of latency performance comparison, we observe similar results when carrying out the RPC-like latency benchmarks (single client to single server communication). As shown in Figure 4, RDMA protocols with busy polling deliver better latency performance than their counterparts with event polling. These results also suggest that, with busy polling, Direct-WriteIMM is the best choice for transferring small messages, and RFP protocol is suitable for message sizes less than 1KB. The reason behind the observations is that, as shown in Figure 3, they use one-sided RDMA operations to complete single-round-trip RPC-like communication. On the other hand, we also use busy polling for large messages to achieve better latency. While Direct-Write-Send and Direct-WriteIMM perform very well, MPI libraries and other applications usually use either Write-RNDV or Read-RNDV to achieve comparable latency performance, reduce memory footprint, and improve scalability.

Figure 5 presents the results of our evaluations with multi-clients to single-server throughput benchmarks. In addition to varying message sizes, polling mechanisms, and number of clients, we bind clients to the NUMA (Non-Uniform Memory Access) node to which the NIC is plugged, when the number of clients is less than or equal to the number of cores of the bound NUMA node on our testbed (i.e., under-subscription). Towards fully leveraging the CPU cores, we do not employ NUMA binding for full-subscription and over-subscription evaluations. As illustrated in the figure, busy polling incurs significant performance degradation with over-subscription setup for message sizes of 512B and 128KB. This is because busy polling mechanism consumes much more CPU cycles than the event polling counterpart and thereby is not as scalable as event polling. Another insight we can get from the figure is that, for small message sizes such as 512B, Direct-WriteIMM with event polling delivers the best performance for under-subscription, full-subscription, and over-subscription setups. However, for large message sizes like 128KB, we observe that the event polling mechanism is more suitable. With event polling, Direct-WriteIMM outperforms other protocols for under-subscription setup, while RFP delivers considerable performance advantage over other protocols. This insight also reflects the observation from the RFP paper [59] that issuing a one-sided RDMA operation (i.e, out-bound RDMA) has much higher overhead than that of serving one (i.e, in-bound RDMA).

3.3 Design Space Analyses

Through the comprehensive performance evaluations in Section 3.2, we know that the design scope for a high-performance RDMA-accelerated RPC framework is too broad, and there is no single design and optimization can beat all others. Prior studies like AR-gRPC [18], HERD [36], FaRM [23], and RFP [59], are optimized towards specific types of applications or adopt relatively generic design approaches to serve some common workloads. However, as an RPC framework, optimizing for specific types of applications or adopting balanced design approaches for generic use cases are far from the desired RPC design. The desired RPC framework should perform well in not only homogeneous services/functions but also heterogeneous ones. For instance, an RPC framework in a distributed file system needs to fetch metadata from metadata servers with low latency and write to (or read from) chunk servers with high throughput. But for existing RPC frameworks, they are not performant in this use case since they are not aware of the heterogeneous functionality requirements. Motivated by the idea of using hints in optimizing database query plans and cache management [19, 20, 38, 48, 52], we come up with a hint-accelerated approach for effectively and efficiently serving heterogeneous services and functions. This hint-accelerated approach enables applications to customize the optimization goal of each service or function, such that our proposed RPC framework can select appropriate and optimal RDMA designs based on the given hint set for these heterogeneous services/functions. Also, the applied optimizations are isolated such that one optimized setup for a service or a function has no side effects on other services and functions.

Figure 6 summarizes the major hint categories and their corresponding design space. Compared with the design space of non-hint-accelerated RPC frameworks, the figure illustrates that hints are able to significantly narrow down the design space. Towards optimizing concurrency performance, we have concurrency hint

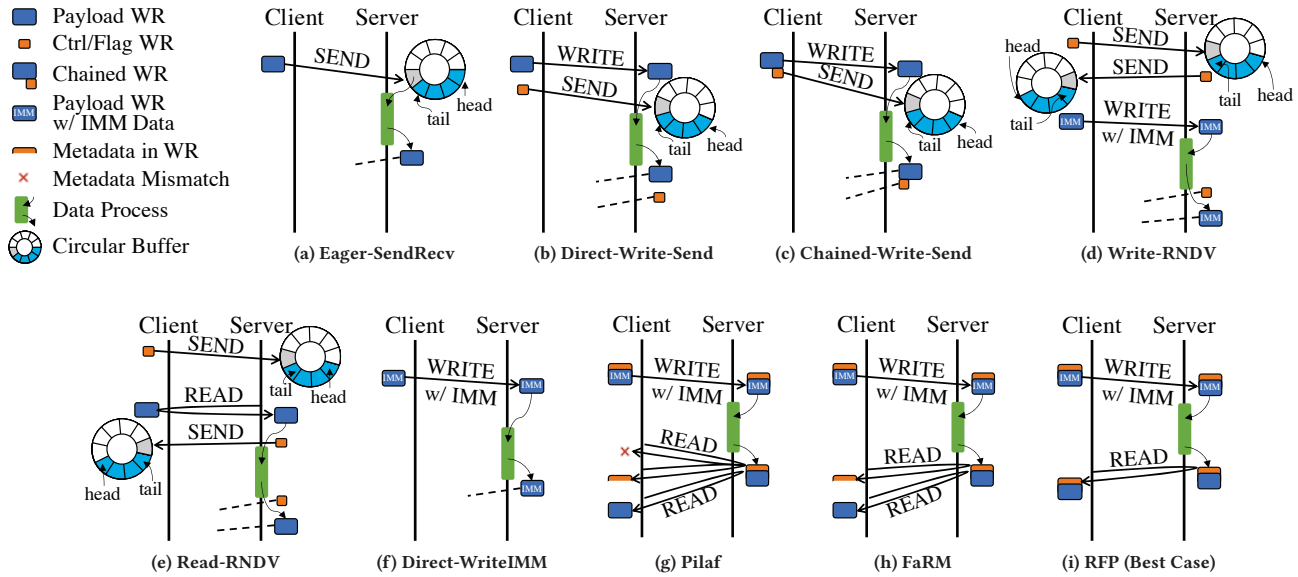


Figure 3: RDMA Protocols. WR: Work Request. Dashed lines indicate that servers repeat the same procedures as what clients have done.

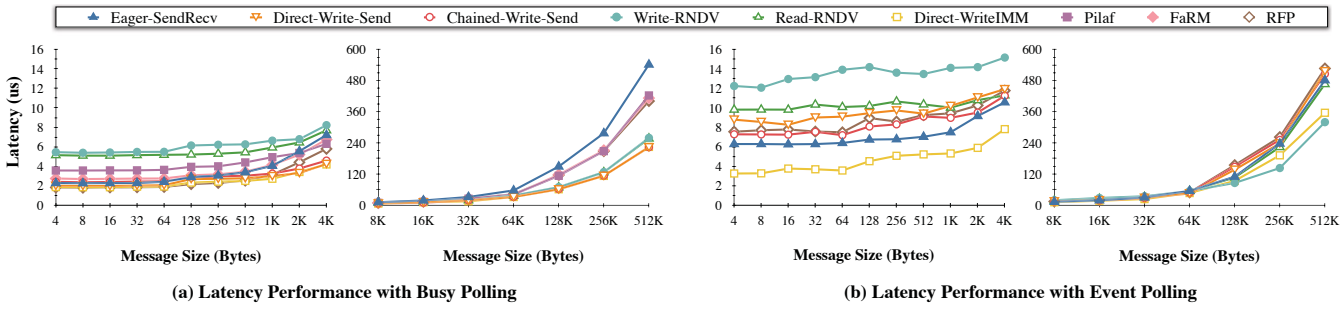


Figure 4: Performance Impact of Different RDMA Protocols on RPC-like Communication Latency

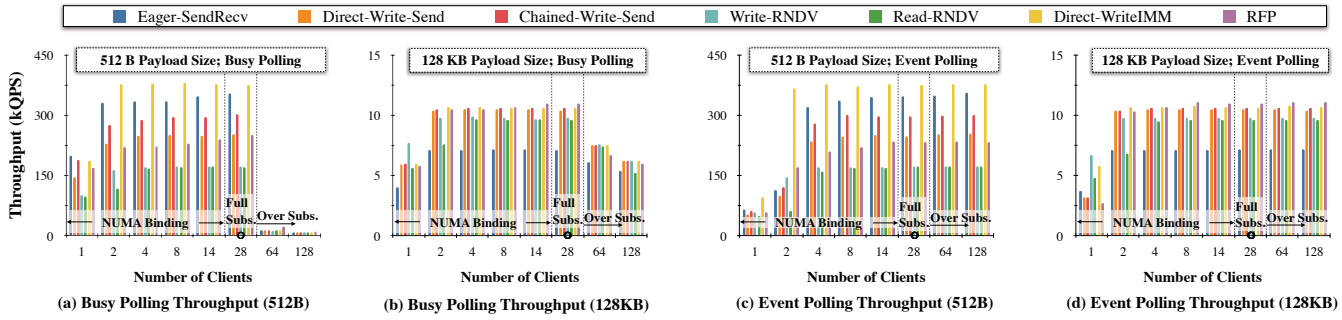


Figure 5: Performance Impact of Different RDMA Protocols on RPC-like Communication Throughput.

(y-axis) that includes three values, i.e., under-subscription, full-subscription, and over-subscription. The x-axis shows the proposed performance goal hint, which is comprised of latency, throughput, and resource utilization. Each service and function in *HatRPC* can also indicate a payload hint to guide our framework for better understanding about the payload size distributions. The design space of each optimization goal is shown in the figure. For instance, if a service or function is marked with over-subscription and throughput

hints, *HatRPC* will use event polling mechanism, and choose RFP for large messages, Direct-WriteIMM for small messages. If applications care about resource utilization, for under-subscription setup, *HatRPC* will use Direct-WriteIMM and Write-RNDV for small and large messages, respectively. While for full-subscription and over-subscription setups, the design space converges to Eager-SendRecv for small messages and Write/Read-RNDV for large messages. This is because that pre-registered buffers used by Direct-WriteIMM and Eager-SendRecv for transferring small messages do not occupy too much memory. However, for large messages, we cannot achieve resource efficiency if we pre-register too much memory. In this case, Write-/Read-RNDV is used for large message communication.

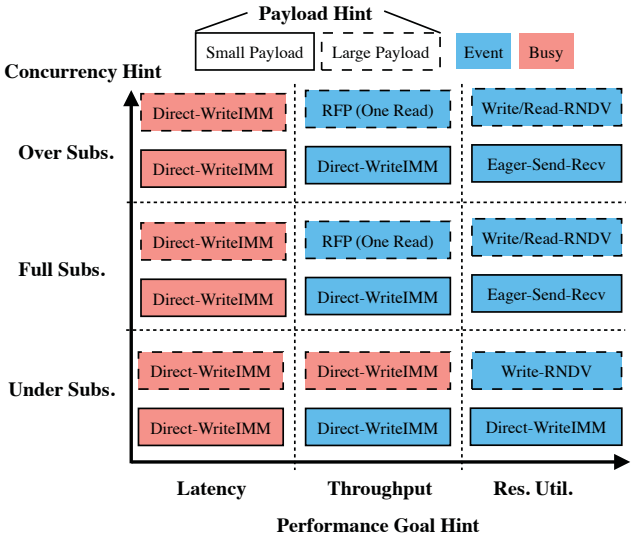


Figure 6: Design Space for Hints and RDMA Protocols

In this paper, we mainly focus on performance-oriented hints, i.e., the hint categories in Figure 6. In addition to these hints, *HatRPC* also supports NUMA binding and hybrid transports (i.e., using TCP for some services/functions and RDMA for other services/-functions), etc. The evaluation in Section 5.5 presents the benefit of using hints to support NUMA binding and hybrid transports.

4 DESIGN

In this section, we first clarify the goals and non-goals of *HatRPC*. Then we illustrate the proposed hints to achieve the goals. Finally, we discuss how we implement *HatRPC* within Apache Thrift and what optimizations are adopted in *HatRPC*.

Non-Goals: a. We do not lay out guidelines for designing the best RDMA-accelerated RPC frameworks. **b.** We do not aim to explore all existing RDMA designs for RPC frameworks.

HatRPC instead strives to gain the following goals:

Hint Design with Minimized Overhead: The end-to-end performance of RPC frameworks is an essential metric to upper-layer applications, thus the proposed hint design should minimize its overhead when we design hint-accelerated RPC frameworks.

Optimized Design Flexibility: Prior studies [17, 21, 23, 24, 36, 46, 65, 68] in the literature have comprehensively explored RDMA's transports, primitives, and optimizations. All of these impressive features of RDMA provide tremendous amount of opportunities for designing high-performance systems. However, the design scope is too broad, and there is no one-size-fits-all design yet. *HatRPC* aims to adopt hints to indicate desired optimization goals for different RPC services or functions, such that the underneath RDMA communication engine can figure out optimized execution plans for them.

Optimization Isolation: We have seen many papers that have proposed designs towards improving performance, scalability, or resource efficiency (e.g., minimized CPU utilization and memory footprint). These designs are globally applied to entire systems. By contrast, this paper attempts to illustrate that hints are able to deliver scalability, reliability, fast connection establishments, and high performance with various granularities. That is, RPC services and functions can simultaneously target different metrics and optimizations by leveraging RPC hints.

4.1 Proposed Hierarchical Hints

Existing RPC systems like Apache Thrift [58] and gRPC [28] receive IDL file from users to generate RPC services. These IDL files typically include user-defined data structures, constant variables, and at least one service with one or several RPC functions. This design partitions the file space into three scopes: global, service and function space. The global scope does not belong to any services but is visible and accessible by all the services. A service/function scope exists for each service/function, respectively. Again, a service scope is viewable by all the function it includes. This vertical partitioning enables us to design the hints in a hierarchical way. Our *HatRPC* supports service-level hints and function-level hints. The service-level hints set tones for all the functions defined in the service while the function-level hints realize a finer grained control over the behavior of some particularly important functions. The hints defined in the function level will override the same type of hints with their own values defined in the enclosing service level, only for that specific RPC function. Since Thrift typically maps one service to one server instance, global-level hints and service-level hints are overlapped. Thus, we target at service-level hints and function-level hints in this work.

The hierarchical hint design is meaningful and crucial in real world applications. Because some applications have heterogeneous communication workloads and often desire different performance goals, especially for RPC functions that need responsiveness, thereby they should be allowed to use more resources than other functions in the same service. On the other hand, it is common for a high priority service to have unimportant functions, e.g., some functions that are called periodically like heartbeats between server and client. These functions neither require a lot of resources, nor have critical performance requirement. With the function-level hints, these functions can be optimized with low priority and give way to other significant RPC functions.

Apart from the hierarchical design, we can also distinguish between the server side and client side, splitting the hints horizontally. In many application scenarios, the performance requirements for server and client are different. Particularly in the literature of RDMA, the mechanism of busy polling which usually brings the

best performance can take up exceeding resources like CPU. This frustrates the server which is often loaded with computation or other intensive I/O tasks while the clients tend to be relatively idle and free to use more resources for network. Another major distinction between server and client lies in the different targets that they have. Servers with high concurrent connections and intensive communications are often optimized towards overall throughput. In the mean time, users or clients typically care more about responsiveness of the communication such that latency is their performance goal. Based on these observations, we further enable the user to give individual hints for server and client.

Figure 7 presents the syntax rules of *HatRPC* IDL file. It is based on Apache Thrift's IDL file [9] and extends the syntax to accommodate the user-defined hints. Hence, *HatRPC* is fully compatible with the original Apache Thrift and existing Thrift applications can be easily altered to leverage *HatRPC*. Service level hints are declared before the functions inside the scope. Function level hints take nearly identical format except that an additional pair of brackets is required as the delimiter after the argument list. Horizontally, 'HintGroup' is separated by three keywords 'hint', 's_hint' (server side), 'c_hint' (client side), and a semicolon at the end. Within each 'HintList' scope, key value hint pairs ('Hint') are separated by comma. Hence, the extended syntax is consistent with that of the original Thrift IDL syntax and made easy to understand and use.

```
Service      ::= 'service' Identifier ( 'extends' Identifier )?
                 '{' HintGroup* Function* '}'
Function     ::= 'oneway'? FunctionType Identifier '(' Field* ')'
                 Throws? ListSeparator? FunctionHint?
FunctionHint ::= '[' HintGroup* ']'
HintGroup    ::= 'hint' ':' HintList ';'
                 | 'c_hint' ':' HintList ';'
                 | 's_hint' ':' HintList ';'
HintList     ::= Hint ',' HintList | Hint
Hint         ::= key '=' value
```

Figure 7: *HatRPC* IDL File Syntax. *HatRPC* Abstract Syntax Tree (AST) nodes are marked red. '*' and '?' are Regex quantifiers.

4.2 Code Generation

In order to transform the user-defined hints to program-recognized and -friendly syntax, we augment the Thrift code generator with hint-related functionalities and propose the *HatRPC* code generator. The *HatRPC* code generator first uses flex [13] to generate a lexical analyzer called flex scanner. *HatRPC* also leverages Bison [2] to generate the parser. The flex and Bison rule files are modified to enable tokenizing and parsing user-defined hints.

As shown in Figure 8, the flex scanner first scans the IDL file and recognizes hint related keywords. These texts are extracted to a sequence of tokens for the Bison parser. The Bison parser defines one-to-one mappings between a sequence of tokens and a grammar node. Each grammar node consists of a matching rule, a function and a return value type that is pre-defined as a class in *HatRPC*. The parsing of the tokenized input starts from a matching process: The parser traverses all grammar nodes' definitions and chooses the first (and only) node whose mapping rule satisfies the sequence of tokens. Then the grammar node is added to an Abstract Syntax Tree (AST) which grows in a top-down manner. The matched tokens are then replaced by the grammar node in the sequence and the matching continues until all the input tokens are transformed to grammar nodes and added to the AST. The final task of the parser is to run the function defined in each grammar node in a reverse direction, in other words, from leaves to the root. A grammar node can proceed its processing only when all its sub-trees have finished executing and returned from their rule functions. This way, a grammar node can use more detailed information collected from its children's return values.

HatRPC code generator prepares several global containers to collect information about hints, RPC-related types and constants during the execution of the rule functions. After the parsing procedure, the code generator will first check the validity of each hint key-value pair, filtering out the hints that have undefined types or unsupported values. Then a merging process will group common hints from the same level for conciseness and readability. When the check, merge, and analysis steps are done, the hints are organized and output as a hierarchical map in the generated files along with the other RPC templates and skeleton for the runtime TRdma.

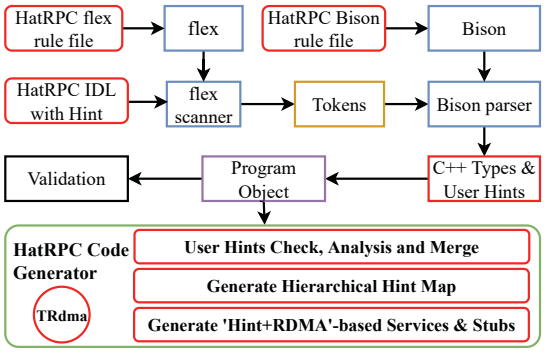


Figure 8: Code Generation of *HatRPC*

4.3 RDMA Communication Engine

The RDMA communication engine in *HatRPC* is enabled by two layers. As shown in Figure 9, TRdma is a layer that bridges the underlying RDMA engine with Thrift library. To be specific, TRdma and TServerRdma are the counterparts of TSocket and TServerSocket in the original Thrift that use TCP/IP. Particular for RDMA, we add TRdmaTransport, a class that is responsible for RDMA handshaking. Upon connection establishment, A TRdmaEndPoint is created that interfaces with the underlying RDMA engine.

When a *HatRPC* service generated from IDL files starts, the static hints will be passed through TRdma layer and eventually to the RDMA engine for RDMA initialization and connection establishment. In some protocols, buffers need to be pre-allocated, registered and exchanged during handshake, and proper parameters need to be set up for creating queue pairs or completion queues. Later during active communication, function-level hints are passed dynamically for RPC calls. We minimize the overhead of the dynamic hints by only passing the pointer and caching the RPC function type at a high level and only pass hints when a new RPC function is invoked.

Inside the RDMA engine, the hints will be translated to RDMA-related parameters and configurations. The configuration affects both receiving (polling) and sending messages. The performance goal hint plays a major role in determining the polling mechanism. Hint 'latency' prioritizes busy polling while hint 'throughput' prefers event-based polling. Other hints like concurrency also affect the decision. Our RDMA engine has implemented several protocols that can be used for the suitable use case, including Eager-SendRecv, Write-RNDV, Direct-Write-Send, Direct-WriteIMM, and RFP. The protocol selection algorithm is based on the results of the preliminary experiments shown in Figure 6.

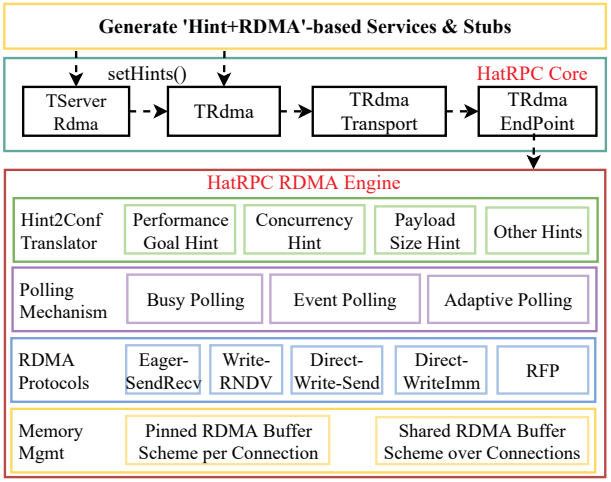


Figure 9: Hint-Accelerated RDMA Communication Engine of *HatRPC*

While performance receives the most attention in previous works about RDMA, it is not the sole focus in our *HatRPC*. Though RDMA WRITE and RDMA READ do not involve remote side's CPU, they require exchanging pre-registered buffers' information beforehand. This typically implies that the buffer is pinned throughout the connection and is exclusive to a pair of peers. When the scale expands, especially at server side, memory could become a big problem. The idle connection also needs to maintain the buffer, which possibly prevents accepting new connections. Protocols including Direct-Write-Send and Direct-Write-Imm are faced with such problems. On the other hand, protocols like Write-RNDV and Eager-SendRecv are more flexible and can improve the memory utilization. For the former, *HatRPC* pre-allocates and registers a buffer pool which makes requesting memories fast during the communication. For the latter, *HatRPC* will allocate a circular buffer. The size of each buffer slot is equal to the Hybrid-EagerRNDV threshold (4KB). These small buffers are posted to receive packets from remote side. In *HatRPC*, if the performance goal is set to res_util (resources utilization), then these two protocols are prioritized.

4.4 Co-design with KV Store

To demonstrate the effectiveness and ease of use of *HatRPC*, we co-design a simple key-value store called HatKV with *HatRPC*. We choose LMDB [1], a B-Tree based embedded database as the storage backend. We later present the experiment results of HatKV with YCSB benchmark in Section 5.4. Figure 10 shows the overview of HatKV with the IDL file for YCSB benchmark [22]. HatKV server and clients communicate by RPCs that are generated by *HatRPC*. Within the server handler, an LMDB instance will serve the requests.

We add MultiPUT and MultiGET operations in YCSB and tailor the 'payload_size' hint for each operation. Our evaluation uses fixed key length of 24 bytes and fixed field length of 100 bytes. The field count for each operation is 10. Furthermore, we set the batch size of MultiPUT and MultiGET to 10. For each of the four RPC functions, we set the function level hints accordingly. For instance, client only transfers 1024 bytes for PUT, including the key and value, while for MultiPUT, the batched key-value pair size can reach up to 10240 bytes for a single RPC. Different payload sizes will result in different optimal choices and *HatRPC* is able to adapt to the best. Different function level hints can be applied to the server and client separately. This again is based on their characteristics and needs. It is especially important for PUT or MultiPUT since there could be a big difference in the message size transferred by client and server. In addition to applying hints to the communication engine, we are inspired by the effective SQL Hints [3, 4, 6, 12, 63] and make further efforts to optimize the LMDB backend through hints. For instance, the number of max readers can be set according to 'concurrency hint'. Varied query, synchronization, and commit strategies are adopted based on different protocols chosen such that the interactions with LMDB will not hinder the critical path in communication.

Apart from the fine-grained control over important RPCs through function-level hints, we can add service-level hints to all the functions of the service for conciseness. As in Figure 10, we add 'concurrency' and the 'perf_goal' hints for the whole service. These two kinds of hints are often the pre-knowledge of the users and can also help *HatRPC* choose the best protocol and configurations.

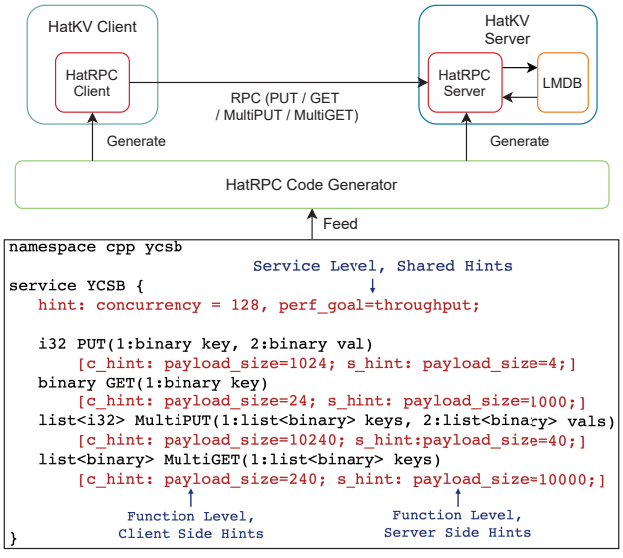


Figure 10: HatKV with IDL File for YCSB. Hints are marked red.

5 EVALUATION

In this section, we conduct multiple experiments to evaluate the effectiveness and efficiency of our proposed *HatRPC*. Our experiments show that *HatRPC* is flexible and achieves high performance and scalability in different scenarios.

5.1 Experimental Setup

Our test environment has 10 nodes in a cluster. Each node is equipped with an Intel Skylake CPU, Xeon Gold 6132. The processor has 28 cores and a frequency of 2.60GHz. The machine has 192GB of RAM and 1TB of HDD for storage. The cluster is connected by Infiniband's ConnectX-5 IB-EDR (100Gbps) and we use MLNX_OFED_LINUX-4.7.

We evaluate *HatRPC* with three different benchmarks: 1) Apache Thrift Benchmarks (ATB) that comprises of three benchmarks, a latency benchmark, a multi-threaded throughput benchmark, and a mix communication benchmark where clients will issue two different RPCs. The two RPCs design is useful for demonstrating function level hints in Section 5.3. 2) Extended YCSB benchmark [22] to study the performance implications of the co-designed HatKV. 3) Standard TPC-H benchmark [61] with a commercial database system applying *HatRPC* approach.

5.2 Evaluation with Service-level Hints

We first test the efficiency of the service-level hints with the latency and throughput benchmarks from the ATB benchmark suite.

For latency evaluation, we mark the service in the *HatRPC* IDL file with the performance goal hint of 'latency', concurrency hint of '1'. With this hint setting, *HatRPC* can automatically select busy polling as the polling mechanism and Direct-WriteIMM as the protocol. We uses varied payload sizes from 4B to 512KB for this benchmark. Figure 11 illustrates the latency of *HatRPC* and four other protocols. Our *HatRPC* can always switch to the appropriate protocol and achieve the best performance. Quantitatively, for small payload sizes ($\leq$ 4KB), *HatRPC* improves the latency by 37% – 54% over Hybrid-EagerRNDV and outperforms Direct-Write-Send by up to 21%. Compared with RFP protocol, the latency performance gain is from 18% – 25%. Since *HatRPC* is configured to use Direct-WriteIMM with busy polling according to Figure 6, the difference between *HatRPC* and Direct-WriteIMM is within 3%. For large payload sizes (> 4KB), *HatRPC* can improve the latency by 20% – 51% over Hybrid-EagerRNDV. The latency improvement is up to 38% and 55% compared with Direct-Write-Send and RFP, respectively. The gap between Direct-WriteIMM and *HatRPC* is negligible.

For the throughput evaluation, our test spans from a single client scenario to a large scale environment of 512 clients. As shown in Figure 12, we partition the x axis into three spaces based on the machine's hardware specifics. Particularly, For the Server Under Subs (number of clients $\leq$ 16), we also apply NUMA bindings to all the protocol testings. In order to optimize the performance towards throughput, we label the IDL file with performance goal hint of 'throughput' and payload_size hint based on the test cases. For small payload size (e.g., 512B), the best protocol is also Direct-WriteIMM which upgrades the aggregated throughput by up to 14% compared with Hybrid-EagerRNDV and up to 20% over Direct-Write-Send. Compared with RFP, *HatRPC* can also achieve up to 12% of performance gain. For large payload sizes (e.g., 128KB), *HatRPC* uses Direct-WriteIMM with busy polling when the number of clients is less than 16 and switches to RFP with event-based polling when the concurrency is above the threshold 16. To compare the performance, *HatRPC* gains up to 56% and 21% of benefits over Hybrid-EagerRNDV and Direct-Write-Send, respectively. *HatRPC* also delivers up to 15% of performance gain over RFP protocol in small scale experiments ($\leq$ 16 clients). When the number of clients exceeds 16, RFP has the advantage and improves over Direct-WriteIMM by 7% – 9%.

5.3 Evaluation with Function-level Hints

To show the flexibility and demonstrate the effectiveness of *HatRPC*, we evaluate the performance of heterogeneous communication workloads and patterns using the Mix Comm Benchmark in ATB. We set up two RPC calls in the service. One function is marked with the performance goal hint 'latency' and the other function with hint 'throughput'. Both functions are correctly labeled with the payload size hint. The clients will randomly issue one of the two RPC functions based on the ratio setting of the two functions. To mimic the server processing in real applications, the service handler at server side will compute a checksum whose overhead increases with the payload size. We adopt a balanced configuration that has 50% of latency function calls and 50% throughput function calls, respectively. We record the latency metrics for latency function calls and throughput for throughput function calls.

As shown in Figure 13, for small message sizes (512B), *HatRPC* improves the latency over Hybrid-EagerRNDV by up to 12% and 18% over Direct-Write-Send. Compared with RFP, *HatRPC* can also improve the latency by up to 9%. For throughput function calls, *HatRPC* can upgrade the throughput performance by up to 11%, 10%, and 8% compared with Hybrid-EagerRNDV, Direct-Write-Send and RFP protocol, respectively. Throughout the test, *HatRPC* sticks to Direct-WriteIMM for both latency and throughput function calls.

Figure 14 shows the Mix Comm Benchmark with large payload size (128KB). In this case, the latency function calls will still go through Direct-WriteIMM, achieving a latency cutdown of up to 12% over Hybrid-EagerRNDV. Compared with Direct-Write-Send and RFP, *HatRPC* can improve by 6% and 7%, respectively. In terms of throughput, *HatRPC* upgrades the throughput performance by up to 25% and 14% over Hybrid-EagerRNDV and Direct-Write-Send, respectively. When in the NUMA Binding section, RFP is inferior to *HatRPC* by up to 11%. And when the concurrency exceeds the threshold 16, *HatRPC* moves to use RFP and gains a boost of up to 7% over Direct-WriteIMM.

5.4 Evaluation of HatKV with YCSB

To illustrate the applicability of *HatRPC*, we study the performance of HatKV with YCSB benchmark [22]. We deploy 4 nodes in the cluster to run a total of 128 clients and 1 node to run the server. The server node saves LMDB lock file and data file in tmpfs and reserves 32GB for the memory map. We halve the proportion of GET and PUT in YCSB workload-A and B for MultiGET and MultiPUT, respectively. Thus, workload-A has the proportion of 25% for GET, PUT, MultiGET and MultiPUT, respectively and workload-B has 47.5% of GET and MultiGET, 2.5% of PUT and MultiPUT. The key

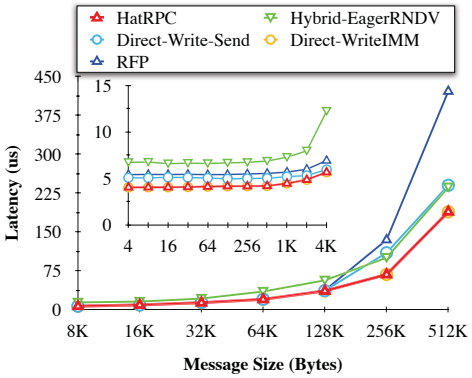
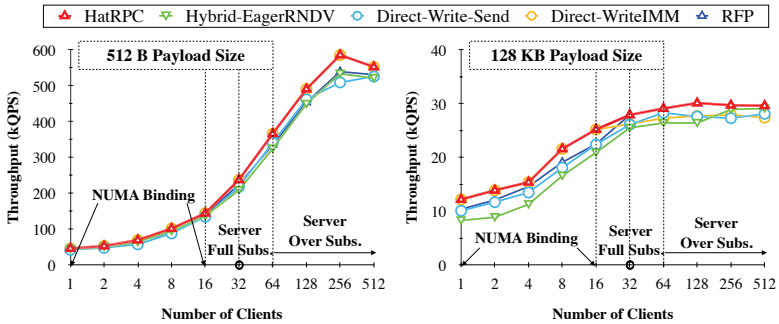


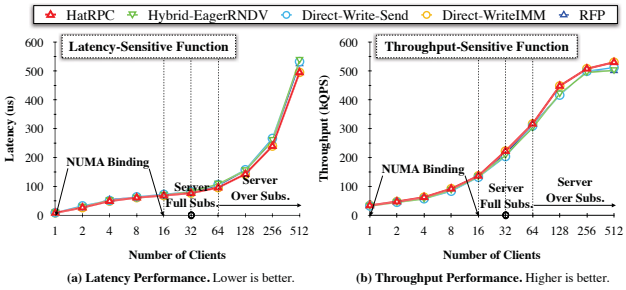
Figure 11: Impact of service-level hints on latency with various payload sizes.



(a) Throughput Performance with 512B Payloads. Higher is better.

(b) Throughput Performance with 128KB Payloads. Higher is better.

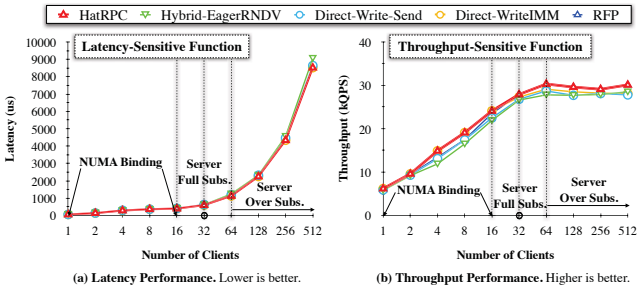
Figure 12: Impact of service-level hints on aggregated throughput with different scales (up to 512 clients).



(a) Latency Performance. Lower is better.

(b) Throughput Performance. Higher is better.

Figure 13: Impact of function-level hints on latency and throughput with various number of clients (512 Byte payload size).



(a) Latency Performance. Lower is better.

(b) Throughput Performance. Higher is better.

Figure 14: Impact of function-level hints on latency and throughput with various number of clients (128 KB payload size).

and field lengths are 24 and 100 Bytes, respectively and the field count is set to 10 which makes the value size 1000 Bytes. For both workload-A and B, the batching size for MultiGET and MultiPUT is 10. Hence, for MultiGET and MultiPUT, each request will have a total of 240 Bytes of keys and 10,000 Bytes of values.

We use two variations of *HatRPC*, HatRPC-Service only sets hints at service level while HatRPC-Function distinguishes each RPC by setting function level hints. We compare them against four popular RDMA systems, AR-gRPC [18], HERD [36], Pilaf [46] and RFP [59]. Since the four systems design their own backends and have different data layouts, it is hard to unify them. Therefore, we only study their communication protocols and emulate them in this evaluation. We make all six candidates share the same backend implementation to avoid unfair comparison.

Figure 15 shows the throughput and latency evaluation results of YCSB workload-A. It can be seen that HatRPC-Service enhances the throughput over AR-gRPC, HERD, Pilaf and RFP by up to 1.19×, 2.68×, 2.21× and 2.22×, respectively. HatRPC-Function further upgrades the throughput by up to 1.68×, 3.80×, 2.29× and 2.31×, respectively. In terms of latency, it can be reduced by HatRPC-Service by up to 35.8%, 73.2%, 53.0% and 52.0%, respectively, compared against AR-gRPC, HERD, Pilaf and RFP. HatRPC-Function can cut off latency by up to 50.3%, 79.7%, 58.1% and 57.2%, respectively.

Figure 16 presents the evaluations results with YCSB workload-B that is read intensive. We can see that HatRPC-Service can raise the throughput by up to 3.80×, 6.38×, 2.28× and 2.36× over AR-gRPC, HERD, Pilaf and RFP, respectively. HatRPC-Function can improve the throughput performance by 4.42×, 7.42×, 2.79× and 2.90×, respectively. Latency performance can be enhanced by HatRPC-Service by up to 75.3%, 84.0%, 62.9% and 64.7%, respectively. It can be further improved by HatRPC-Function by up to 77.4%, 85.5%, 66.3% and 67.9%, respectively.

We come up with possible explanations for the results. For Pilaf and RFP, both of them use RDMA READ for GET to fetch results without server's participation. They expect short or no server processing time and are designed for small payload sizes. Thus, their performance drops abruptly from GET to MultiGET, as the fetched sizes are 10 times larger. HERD uses RDMA SEND for sending server's response, thereby it can not deliver good performance for GET or MultiGET operations. On the other hand, PUT only returns few bytes from server and HERD is expected to have higher performance in this case. AR-gRPC uses two protocols (Eager and Read-RNDV) and adaptively switches between them. It can handle large payload sizes by using Read-RNDV, but will incur more control messages when payload sizes are slightly larger than switching point. These protocols can deliver good performance in their own comfort zones but none of them can adapt to different workloads and scenarios. However, *HatRPC* can adapt to different settings and deliver good performance based on the user given hints. Apart from the improvement from communication perspective, the backend's optimization by hints also plays an important role in magnifying

the boost. The detailed evaluations of the co-designed KV store sufficiently demonstrate the effectiveness of *HatRPC* and the potential of hints in other applications and systems.

5.5 Evaluation with TPC-H Workload

To further show benefits of *HatRPC*, we successfully apply *HatRPC* to a commercial database system to enable hint-accelerated RDMA communications. We conduct our experiments on the standard TPC-H benchmark with a scale factor of $SF1000$ (i.e., $1TB$ data) to evaluate *HatRPC*. These experiments are run on all 10 InfiniBand nodes as described in Section 5.1. The TPC-H benchmark is a Decision Support System (DSS) benchmark consisting of complex business-oriented queries against a database scheme that models real-world business databases.

Figure 17 gives the execution time of all TPC-H queries. Compared with default Thrift over IPoIB, the *HatRPC*-Service approach reduces the total execution time of all 22 queries by 7.2% and improves the query performance by up to 21.2% ($Q20$). To further extract the performance potential of *HatRPC*, we utilize function-granularity performance hints as well as NUMA binding hints and hybrid transport hints, and we call this approach *HatRPC*-Function. By using *HatRPC*-Function, the database obtains noticeable performance improvements than using *HatRPC*-Service. Figure 17 shows *HatRPC*-Function outperforms Thrift over IPoIB and *HatRPC*-Service by 1.27× and 1.18×, respectively, with respect to total execution time. For specific queries, *HatRPC*-Function delivers performance improvement by up to 1.51× ($Q19$) and 1.44× ($Q19$) if compared with Thrift over IPoIB and *HatRPC*-Service, respectively.

6 RELATED WORK

Application Definability: Existing systems have proposed several solutions for users to define the behaviors of applications. Apart from the widely adopted methods like configuring through systems APIs [8, 10, 16, 26, 27, 33, 49] or individual configuration files [30, 67], OpenMP [5] exploits preprocessing directives in C/C++ and uses '#pragma' to create, manage, and synchronize parallel code segments. The idea of using hints to define application behaviors was previously proposed in SQL [4] and supported by various vendors [3, 6, 12, 63] to optimize query, resource utilization, and consistency, etc. Inspired by these previous efforts, this paper proposes the *HatRPC* framework, which is the first work to propose a hierarchical hint scheme towards achieving various optimization goals for heterogeneous RPCs over RDMA. *HatRPC* is the first attempt to explore a practical code generation approach for RDMA.

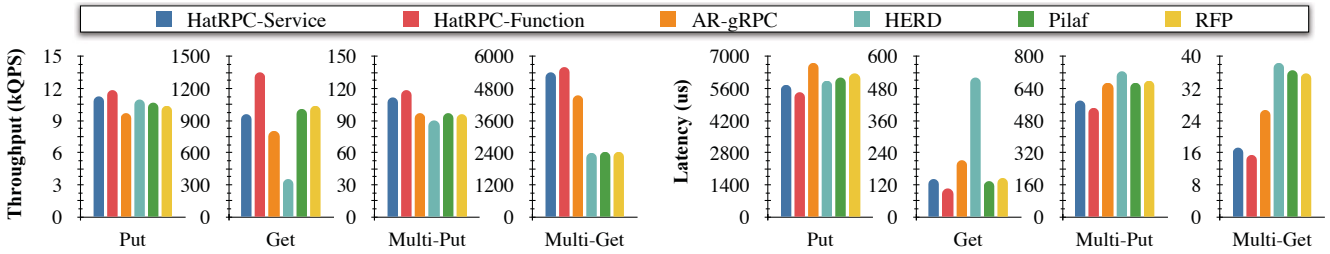
Remote Procedure Call Optimization: There are many existing methods to implement RPC for higher throughput and lower latency. Previously, [64] defers procedure selection from client to runtime, hence expediting the procedure calling and further increasing scalability. [44] puts forward an RPC model in Java using specialized serialization. Serialization and deserialization guarantee the portability and [14, 15] adopt differential approaches to cut the cost. The approach in [14, 15] is based on the observation of repeated procedure calls in the service. It saves the serialized messages after each RPC call and only serialize the differential parts of subsequent calls to reduce the overhead. LRPC [53] uses coarse-grained protection architecture and transfer control to eliminate high overhead within the same protection domain. eRPC [35] designs new RPC library tuned for data center.

High-Performance RDMA Applications: Many recent studies have focused on applying RDMA to different application scenarios. [17, 24] merge the features of RDMA and DBMS to eliminate the bottleneck of communications between nodes. FaRM [23] adopts RDMA over TCP/IP and Ethernet and reports great performance improvement. HERD [36] and other models [46, 65] utilize RDMA to cut the round trip cost by using one-sided RDMA read and write in key-value systems. The studies in [21, 68] construct scalable distributed systems using RDMA instead of the prevailing co-partitioning. Researchers in [18, 34, 50] combine RDMA with Deep Learning frameworks like Tensorflow, Caffe, and Parameter Server model. [55–57] leverage RDMA and Erasure Coding (EC) offload capability on modern RNICs to build high-performance erasure-coded distributed storage systems. Despite a large amount of existing works on designing high-performance RDMA-accelerated systems, few have studied how to design RPC frameworks for heterogeneous services and functions. This work complements this important support to a great extent for the community.

RDMA Protocol Selection: Existing RDMA systems typically select their RDMA protocol depending on request/response packets, payload sizes, or server delays. In addition, most of the past RDMA designs in RPC engines such as AR-gRPC [18], HERD [36], Pilaf [46], RFP [59] etc. only support two or three different protocols. For instance, AR-gRPC only provides eager or read rendezvous protocols. Herd only supports direct-write for request and send-recv for response. Pilaf only supports RDMA-READ based polling. RFP uses RDMA-READ polling first and then falls back to send-recv. As we can see, none of them can adapt their designs to different application scenarios easily. In contrast, our work supports various protocols and utilize user given hints to adapt to heterogeneous application needs and improve the system's performance.

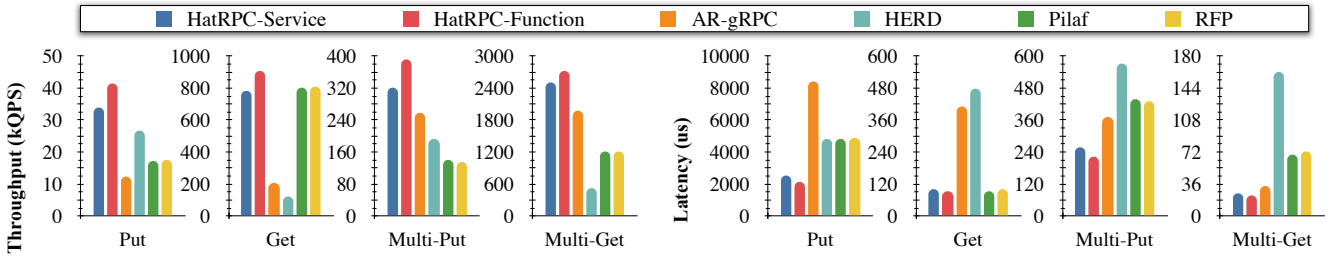
7 CONCLUSION AND FUTURE WORK

In this paper, we propose *HatRPC*, a hint-accelerated Thrift RPC framework over RDMA transport. *HatRPC* adopts a hierarchical hint scheme to narrow down the RDMA protocol design space for achieving different optimization goals. The proposed hint design consists of service-granularity and function-granularity hints and supports optimization isolation. With the hint design, *HatRPC* enables upper-layer applications to mark each RPC service and function with different sets of hints to guide the underneath RDMA communication engine for particular optimization preferences. Therefore, *HatRPC* is a hint-accelerated design towards heterogeneous RPC services, which are the practical use cases in common systems. Performance evaluations with our proposed Apache Thrift Benchmarks (ATB), extended YCSB benchmark, and TPC-H workload demonstrate the effectiveness and efficiency of the proposed *HatRPC*. Quantitatively, *HatRPC*-Function can deliver up to 55% performance improvement for ATB benchmarks and up to 1.51× speedup for TPC-H queries if compared with Thrift over IPoIB. In addition, the co-designed HatKV with *HatRPC* and LMDB can also achieve up to 85.5% performance improvement in the YCSB evaluations. In the future, we will try to support more hint categories



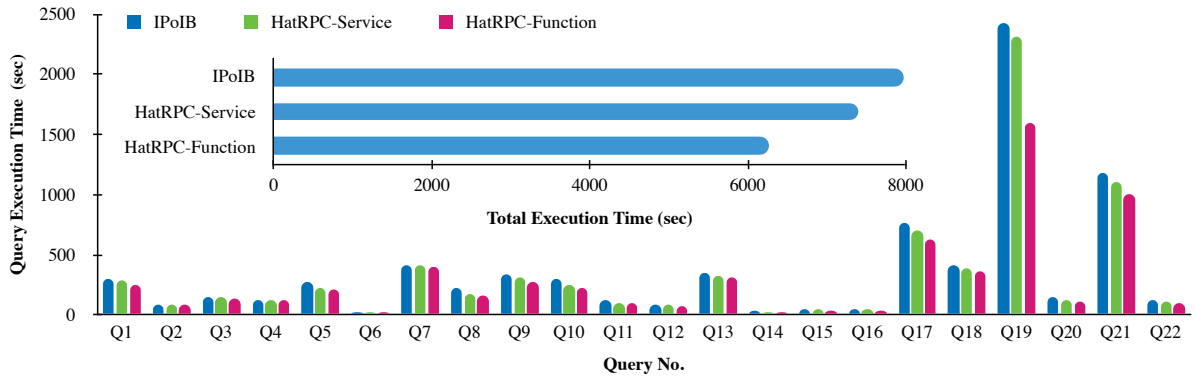
(a) Throughput Performance of Different Operations. Higher is better. **(b) Latency Performance of Different Operations.** Lower is better.

Figure 15: Benchmarking HatKV with YCSB-A Workload and 128 Clients



(a) Throughput Performance of Different Operations. Higher is better. **(b) Latency Performance of Different Operations.** Lower is better.

Figure 16: Benchmarking HatKV with YCSB-B Workload and 128 Clients



	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12	Q13	Q14	Q15	Q16	Q17	Q18	Q19	Q20	Q21	Q22	Total
IPoIB	297	90	153	128	272	14	419	222	342	294	119	85	349	30	46	48	771	415	2420	151	1184	121	7970
HatRPC-Service	289	84	149	123	221	13	416	180	314	244	104	81	329	23	44	43	705	382	2312	119	1109	115	7399
HatRPC-Function	246	83	137	122	215	12	400	167	274	222	99	74	313	22	42	41	626	358	1599	108	1001	97	6258

Figure 17: Performance Comparison with TPC-H Benchmark (1*TB* Data). *HatRPC*-Service approach only adopts service-granularity hints, while *HatRPC*-Function adopts function-granularity hints.

beyond the scope of performance. We plan to make the system more generic and adapt it to other RPC systems.

ACKNOWLEDGMENTS

We would like to sincerely thank Yujie Hui from The Ohio State University for his help in conducting some of the experiments. We want to thank the anonymous reviewers for their insightful comments and suggestions. This work was done when Tianxi Li and Dr. Haiyang Shi were students in PADSYS Lab, led by Prof. Xiaoyi Lu. We want to sincerely thank all the sponsors to PADSYS Lab. This work was supported in part by the NSF research grant CCF #1822987.

REFERENCES

[1] 2016. LMDB: Lightning Memory-Mapped Database Manager (LMDB). http://www.lmdb.tech/doc/.
[2] 2021. Bison - GNU Project - Free Software Foundation. https://www.gnu.org/software/bison/.
[3] 2021. EDB Optimizer Hints. https://www.enterprisedb.com/edb-docs/d/edb-postgres-advanced-server/user-guides/database-compatibility-for-oracle-developers-guide/11/Database_Compatibility_for_Oracle_Developers_Guide.1.038.html.
[4] 2021. Hint (SQL). https://en.wikipedia.org/wiki/Hint_(SQL).
[5] 2021. Home - OpenMP. https://www.openmp.org/.

[6] 2021. MySQL :: MySQL 8.0 Reference Manual :: 8.9.3 Optimizer Hints. https://dev.mysql.com/doc/refman/8.0/en/optimizer-hints.html.

[7] 2021. OpenUCX/UCX: Unified Communication X. https://github.com/openucx/ucx.

[8] 2021. The Info Object. https://www.mpi-forum.org/docs/mpi-2.1/mpi21-report-bw/node194.htm.

[9] 2021. Thrift Interface Description Language. https://github.com/apache/thrift/blob/master/doc/specs/idl.md.

[10] 2021. Tool Interfaces (MPI-T), MPICH Parameters and Instrumentation - MPICH.

[11] 2021. UCP Hello World Example. https://github.com/openucx/ucx/blob/master/examples/ucp_hello_world.c.

[12] 2021. Using Optimizer Hints. https://docs.oracle.com/cd/B19306_01/server.102/b14211/hintsref.htm#i8327.

[13] 2021. westes/flex: The Fast Lexical Analyzer - scanner generator for lexing in C and C++. https://github.com/westes/flex.

[14] N Abu-Ghazaleh and MJ Lewis. 2004. Madhusudhan Govindaraju. Differential Serialization for Optimized SOAP Performance. In *Proceedings of the 13th IEEE International Symposium on High Performance. Distributed Computing (HPDC-13), Honolulu, Hawaii*, Vol. 55.

[15] Nayef Abu-Ghazaleh and Michael J Lewis. 2005. Differential Deserialization for Optimized Soap Performance. In *SC'05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*. IEEE, 21–21.

[16] M. Bayatpour, N. Sarkauskas, H. Subramoni, J. Hashmi, and D. K. Panda. 2021. BluesMPI: Efficient MPI Non-blocking Alltoall Offloading Designs on Modern BlueField Smart NICs. In *Proceedings of ISC HIGH PERFORMANCE* (Frankfurt, Germany).

[17] Carsten Binnig, Andrew Crotty, Alex Galakatos, Tim Kraska, and Erfan Zamanian. 2016. The End of Slow Networks: It's Time for a Redesign. *Proceedings of the VLDB Endowment* 9, 7 (2016), 528–539.

[18] Rajarshi Biswas, Xiaoyi Lu, and Dhabaleswar K Panda. 2018. Accelerating Tensorflow with Adaptive RDMA-Based gRPC. In *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*. IEEE, 2–11.

[19] Nicolas Bruno, Surajit Chaudhuri, and Ravishankar Ramamurthy. 2009. Interactive Plan Hints for Query Optimization. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data* (Providence, Rhode Island, USA) *(SIGMOD '09)*. Association for Computing Machinery, New York, NY, USA, 1043–1046. https://doi.org/10.1145/1559845.1559976

[20] Nicolas Bruno, Surajit Chaudhuri, and Ravi Ramamurthy. 2009. Power Hints for Query Optimization. In *2009 IEEE 25th International Conference on Data Engineering*. 469–480. https://doi.org/10.1109/ICDE.2009.68

[21] Youmin Chen, Youyou Lu, and Jiwu Shu. 2019. Scalable RDMA RPC on Reliable Connection with Efficient Resource Sharing. In *Proceedings of the Fourteenth EuroSys Conference 2019*. ACM, 19.

[22] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.

[23] Aleksandar Dragojević, Dushyanth Narayanan, Miguel Castro, and Orion Hodson. 2014. FaRM: Fast Remote Memory. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*. 401–414.

[24] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. 2015. No Compromises: Distributed Transactions with Consistency, Availability, and Performance. In *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 54–70.

[25] Philipp Fent, Alexander van Renen, Andreas Kipf, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2020. Low-Latency Communication for Fast DBMS Using RDMA and Shared Memory. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 1477–1488. https://doi.org/10.1109/ICDE48307.2020.00131

[26] William Fox, Devarshi Ghoshal, Abel Souza, Gonzalo P. Rodrigo, and Lavanya Ramakrishnan. 2017. E-HPC: A Library for Elastic Resource Management in HPC Environments. In *Proceedings of the 12th Workshop on Workflows in Support of Large-Scale Science* (Denver, Colorado) *(WORKS '17)*. Association for Computing Machinery, New York, NY, USA, Article 1, 11 pages. https://doi.org/10.1145/3150994.3150996

[27] Edgar Gabriel, Graham E Fagg, George Bosilca, Thara Angskun, Jack J Dongarra, Jeffrey M Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, et al. 2004. Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*. Springer, 97–104.

[28] Google. 2021. grpc/grpc: The C Based gRPC (C++, Python, Ruby, Objective-C, PHP, C). https://github.com/grpc/grpc.

[29] Shashank Gugnani, Xiaoyi Lu, and Dhabaleswar K Panda. 2017. Swift-X: Accelerating OpenStack Swift with RDMA for Building an Efficient HPC Cloud. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 238–247.

[30] Chris Harris, Patrick O'Leary, Michael Grauer, Aashish Chaudhary, Chris Kotfila, and Robert O'Bara. 2016. Dynamic Provisioning and Execution of HPC Workflows Using Python. In *2016 6th Workshop on Python for High-Performance and Scientific Computing (PyHPC)*. IEEE, 1–8.

[31] Nusrat S Islam, Xiaoyi Lu, Md Wasi-ur Rahman, and Dhabaleswar K Panda. 2014. SOR-HDFS: A SEDA-Based Approach to Maximize Overlapping in RDMA-Enhanced HDFS. In *Proceedings of the 23rd international symposium on High-performance parallel and distributed computing*. ACM, 261–264.

[32] Nusrat Sharmin Islam, Xiaoyi Lu, Md Wasi-ur Rahman, Dipti Shankar, and Dhabaleswar K Panda. 2015. Triple-H: A Hybrid Approach to Accelerate HDFS on HPC Clusters with Heterogeneous Storage Architecture. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 101–110.

[33] J. Zhang and X. Lu and D. K. Panda. 2017. High-Performance Virtual Machine Migration Framework for MPI Applications on SR-IOV Enabled InfiniBand Clusters. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Orlando, USA.

[34] Chengfan Jia, Junnan Liu, Xu Jin, Han Lin, Hong An, Wenting Han, Zheng Wu, and Mengxian Chi. 2018. Improving the Performance of Distributed Tensorflow with RDMA. *International Journal of Parallel Programming* 46, 4 (2018), 674–685.

[35] Anuj Kalia, Michael Kaminsky, and David Andersen. 2019. Datacenter RPCs can be General and Fast. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*. 1–16.

[36] Anuj Kalia, Michael Kaminsky, and David G Andersen. 2014. Using RDMA Efficiently for Key-Value Services. In *ACM SIGCOMM Computer Communication Review*, Vol. 44. ACM, 295–306.

[37] Anuj Kalia, Michael Kaminsky, and David G. Andersen. 2016. Design Guidelines for High Performance RDMA Systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, Denver, CO, 437–450. https://www.usenix.org/conference/atc16/technical-sessions/presentation/kalia

[38] Xuhui Li, Ashraf Aboulnaga, Kenneth Salem, Aamer Sachedina, and Shaobo Gao. 2005. Second-Tier Cache Management Using Write Hints. In *Proceedings of the 4th Conference on USENIX Conference on File and Storage Technologies - Volume 4* (San Francisco, CA) *(FAST'05)*. USENIX Association, USA, 9.

[39] Jiuxing Liu, Jiesheng Wu, Sushmitha P. Kini, Pete Wyckoff, and Dhabaleswar K. Panda. 2003. High Performance RDMA-based MPI Implementation over InfiniBand. In *Proceedings of the 17th Annual International Conference on Supercomputing, ICS 2003, San Francisco, CA, USA, June 23-26, 2003*, Utpal Banerjee, Kyle A. Gallivan, and Antonio González (Eds.). ACM, 295–304. https://doi.org/10.1145/782814.782855

[40] Xiaoyi Lu, Nusrat S Islam, Md Wasi-Ur-Rahman, Jithin Jose, Hari Subramoni, Hao Wang, and Dhabaleswar K Panda. 2013. High-Performance Design of Hadoop RPC with RDMA over InfiniBand. In *2013 42nd International Conference on Parallel Processing*. IEEE, 641–650.

[41] Xiaoyi Lu, Dipti Shankar, Shashank Gugnani, and Dhabaleswar K Panda. 2016. High-Performance Design of Apache Spark with RDMA and Its Benefits on Various Workloads. In *2016 IEEE International Conference on Big Data (Big Data)*. IEEE, 253–262.

[42] Xiaoyi Lu, Haiyang Shi, Rajarshi Biswas, M Haseeb Javed, and Dhabaleswar K Panda. 2018. DLoBD: A Comprehensive Study of Deep Learning over Big Data Stacks on HPC Clusters. *IEEE Transactions on Multi-Scale Computing Systems* 4, 4 (2018), 635–648.

[43] Youyou Lu, Jiwu Shu, Youmin Chen, and Tao Li. 2017. Octopus: an rdma-enabled distributed persistent memory file system. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*. 773–785.

[44] Jason Maassen, Rob Van Nieuwpoort, Ronald Veldema, Henri Bal, Thilo Kielmann, Ceriel Jacobs, and Rutger Hofman. 2001. Efficient Java RMI for Parallel Programming. *ACM Transactions on Programming Languages and Systems* 23, 6 (2001), 747–775.

[45] Mellanox. 2019. RDMA Aware Networks Programming User Manual. http://www.mellanox.com/related-docs/prod_software/RDMA_Aware_Programming_user_manual.pdf.

[46] Christopher Mitchell, Yifeng Geng, and Jinyang Li. 2013. 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)*. 103–114.

[47] MPICH. 2021. MPICH. https://www.mpich.org/.

[48] Oracle. 2021. Using Optimizer Hints. https://docs.oracle.com/cd/B19306_01/server.102/b14211/hintsref.htm.

[49] Aarthi Raveendran, Tekin Bicer, and Gagan Agrawal. 2011. A Framework for Elastic Execution of Existing MPI Programs. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. 940–947. https://doi.org/10.1109/IPDPS.2011.240

[50] Yufei Ren, Xingbo Wu, Li Zhang, Yandong Wang, Wei Zhang, Zijun Wang, Michel Hack, and Song Jiang. 2017. iRDMA: Efficient USE of RDMA in Distributed Deep Learning Systems. In *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE, 231–238.

[51] Wolf Rödiger, Tobias Mühlbauer, Alfons Kemper, and Thomas Neumann. 2015. High-Speed Query Processing over High-Speed Networks. *Proc. VLDB Endow.* 9, 4 (2015), 228–239. https://doi.org/10.14778/2856318.2856319

[52] Microsoft SQL Server. 2021. Hints (Transact-SQL). https://docs.microsoft.com/en-us/sql/t-sql/queries/hints-transact-sql?view=sql-server-ver15.
[53] Keith Seymour, Hidemoto Nakada, Satoshi Matsuoka, Jack Dongarra, Craig Lee, and Henri Casanova. 2002. Overview of GridRPC: A Remote Procedure Call API for Grid Computing. In *International Workshop on Grid Computing*. Springer, 274–278.
[54] Dipti Shankar, Xiaoyi Lu, Nusrat Islam, Md Wasi-Ur-Rahman, and Dhabaleswar K Panda. 2016. High-Performance Hybrid Key-Value Store on Modern Clusters with RDMA Interconnects and SSDs: Non-Blocking Extensions, Designs, and Benefits. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 393–402.
[55] Haiyang Shi and Xiaoyi Lu. 2019. TriEC: Tripartite Graph Based Erasure Coding NIC Offload. In *The 32nd International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*.
[56] Haiyang Shi and Xiaoyi Lu. 2020. INEC: Fast and Coherent In-Network Erasure Coding. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '20)*. IEEE Press.
[57] Haiyang Shi, Xiaoyi Lu, Dipti Shankar, and Dhabaleswar K Panda. 2019. UMR-EC: A Unified and Multi-Rail Erasure Coding Library for High-Performance Distributed Storage Systems. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 219–230.
[58] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. 2007. Thrift: Scalable Cross-Language Services Implementation. *Facebook White Paper* 5, 8 (2007).
[59] Maomeng Su, Mingxing Zhang, Kang Chen, Zhenyu Guo, and Yongwei Wu. 2017. RFP: When RPC is Faster than Server-Bypass with RDMA. In *Proceedings of the Twelfth European Conference on Computer Systems, EuroSys 2017, Belgrade, Serbia, April 23-26, 2017*, Gustavo Alonso, Ricardo Bianchini, and Marko Vukolic (Eds.). ACM, 1–15. https://doi.org/10.1145/3064176.3064189
[60] Sayantan Sur, Hyun-Wook Jin, Lei Chai, and Dhabaleswar K. Panda. 2006. RDMA Read Based Rendezvous Protocol for MPI over InfiniBand: Design Alternatives and Benefits. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2006, New York, New York, USA, March 29-31, 2006*, Josep Torrellas and Siddhartha Chatterjee (Eds.). ACM, 32–39. https://doi.org/10.1145/1122971.1122978
[61] Transaction Processing Performance Council. 2019. TPC-H Benchmark. http://www.tpc.org/tpch/.
[62] The Ohio State University. 2021. MVAPICH. http://mvapich.cse.ohio-state.edu/.
[63] VanMSFT. 2021. Hints (Transact-SQL) - SQL Server | Microsoft Docs. https://docs.microsoft.com/en-us/sql/t-sql/queries/hints-transact-sql?view=sql-server-ver15.
[64] Rangaswamy Vasudevan and Caveh Jalali. 1999. Remote Procedure Call System and Method for RPC Mechanism Independent Client and Server Interfaces Interoperable with Any of a Plurality of Remote Procedure Call Backends. US Patent 5,887,172.
[65] Yandong Wang, Xiaoqiao Meng, Li Zhang, and Jian Tan. 2014. C-hint: An Effective and Reliable Cache Management for RDMA-Accelerated Key-Value Stores. In *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 1–13.
[66] Yandong Wang, Xiaoqiao Meng, Li Zhang, and Jian Tan. 2014. C-Hint: An Effective and Reliable Cache Management for RDMA-Accelerated Key-Value Stores. In *Proceedings of the ACM Symposium on Cloud Computing* (Seattle, WA, USA) *(SOCC '14)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/2670979.2671002
[67] Md Wasi-ur Rahman, Nusrat Sharmin Islam, Xiaoyi Lu, Dipti Shankar, and Dhabaleswar K DK Panda. 2018. MR-Advisor: A Comprehensive Tuning, Profiling, and Prediction Tool for MapReduce Execution Frameworks on HPC Clusters. *J. Parallel and Distrib. Comput.* 120 (2018), 237–250.
[68] Erfan Zamanian, Carsten Binnig, Tim Harris, and Tim Kraska. 2017. The End of a Myth: Distributed Transactions can Scale. *Proceedings of the VLDB Endowment* 10, 6 (2017), 685–696.

Appendix: Artifact Description/Artifact Evaluation

SUMMARY OF THE EXPERIMENTS REPORTED

HatRPC

Introduction This is the README of the HatRPC (Hint-Accelerated Thrift RPC over RDMA). HatRPC exploits hints for users to define the behavior of RPC communication in a convenient and easy way. For details, please check out the paper: HatRPC: Hint-Accelerated Thrift RPC over RDMA

This repository includes the necessary components to reproduce the results in the paper. lib contains libraries of HatRPC with its dependencies. bin contains the executables for the experiments and evaluations. cluster_a_env.out describes the hardware and environment information of the cluster in the evaluation setup section in the paper. example includes two example HatRPC IDL (Interface Description Language) files for generating templates for atb and ycsb experiments.

Dependencies All dependencies and the pre-compiled libraries are included in lib. HatRPC library is dependent on Gflags (v2.2.1), Glog (v0.3.5), Hwloc(v2.0), TBB(2019_U2) and our RDMA communication library Marlin. ATB and YCSB benchmark executables are dependent on Boost (v1.58.0) YCSB experiments are backed by LMDB (v0.9.29). LMDB libraries are not included because of file size limit.

Build All executables are pre-built and included in bin directory. The bin/hatrpc_gen is the HatRPC generator which takes HatRPC idl files as input and output generated templates with corresponding hints. For instance, to generate templates from atb_example.thrift in direcotry gen, one can use hatrpc_gen -out gen –gen cpp ar_grpc.thrift

Run For Figure 11 14, we use ATB benchmark suites. The corresponding executables are named as bin/atb_*. For Figure 15 16, we use YCSB benchmark suites. The executables are named as bin/ycsb_*.

General Usage: ATB Latency:

bin/atb_lat_server –port <port_no> bin/atb_lat_client –ip <server_ip> –port <port_no> –iter <n_iterations> –min <min payload size> –max <max payload size>

ATB Throughput:

bin/atb_thr_server –port <port_no> –clients <n_clients> bin/atb_thr_client –ip <server_ip> –port <port_no> –iter <n_iterations> –size <payload size> –threads <n_threads>

ATB Mix Comm:

bin/atb_thr_server –port <port_no> –clients <n_clients> bin/atb_thr_client –ip <server_ip> –port <port_no> –iter <n_iterations> –l_req_sz <latency request size> –l_res_sz <latency response size> –t_req_sz <throughput request size> –t_res_sz <throughput response size> –threads <n_threads> –latency_percent <percentage of latency functions>

YCSB:

ycsb_hatrpc_server –port <port_no> –clients <n_clients> ycsb_hatrpc_client -db hatrpc -threads <n_threads> -host <server_ip> -port <port_no> -P <workload_file>

Experiments: We use the following commands and parameters for our experiments. Note numa is not used for over-subscription.

ATB Latency:

THRIFT_RDMA_ROUNDROBIN_ENABLED=0 THRIFT_RDMA_LIMIT=20480 GLOG=-1 numactl –membind=1 –cpunodebind=1 otb_lat_server –port 9090 THRIFT_RDMA_ROUNDROBIN_ENABLED=0 THRIFT_RDMA_LIMIT=20480 GLOG=-1 numactl –membind=1 –cpunodebind=1 otb_lat_client –ip ${server_ip} –port 9090 –iter 10000 –max 1048576

ATB Throughput (Note that the sum of all client processes thread count must equal thread count for server):

THRIFT_RDMA_ROUNDROBIN_ENABLED=0 THRIFT_RDMA_LIMIT=20480 GLOG=-1 numactl –membind=1 –cpunodebind=1 otb_thr_server –port 9090 –port 9090 –clients <n_threads: 1 512> THRIFT_RDMA_ROUNDROBIN_ENABLED=0 THRIFT_RDMA_LIMIT=20480 GLOG=-1 numactl –membind=1 –cpunodebind=1 otb_thr_client –ip ${server_ip} –port 9090 –size <payload size: 512 or 131072> –threads <threads_per_process>

YCSB

THRIFT_RDMA_ROUNDROBIN_ENABLED=0 THRIFT_RDMA_LIMIT=20480 GLOG=-1 MULTI-READ_BATCH=10 MULTIUPDATE_BATCH=10 ycsb_hatrpc_server –port 9090 –clients 128 THRIFT_RDMA_ROUNDROBIN_ENABLED=0 THRIFT_RDMA_LIMIT=20480 GLOG=-1 MULTI-READ_BATCH=10 MULTIUPDATE_BATCH=10 ycsb_hatrpc_client -db hatrpc -threads 128 -host <server_ip> -port 9090 -P workloady.spec

Author-Created or Modified Artifacts:

```
Persistent ID: https://github.com/FlyingHObb1t/HatRP⌋
↪ C-Artifact.git
Artifact name: HatRPC-Artifact
```

BASELINE EXPERIMENTAL SETUP, AND MODIFICATIONS MADE FOR THE PAPER

Relevant hardware details: MLNX_OFED_LINUX_4.7

Operating systems and versions: Linux kernel 3.10.0

Compilers and versions: g++ 4.8.5