A Performance Study of UCX over InfiniBand

Nikela Papadopoulou
National Technical University of Athens
Athens, 15780, Greece
E-mail: nikela@cslab.ece.ntua.gr

Lena Oden
Argonne National Laboratory
Argonne, IL, 60439, USA
E-mail: loden@mcs.anl.gov

Pavan Balaji
Argonne National Laboratory
Argonne, IL, 60439, USA
E-mail: balaji@anl.gov

***Abstract*—UCX is an open-source communication framework with a two-level API design targeted at addressing the needs of large supercomputing systems. The lower-level interface, UCT, adds minimal overhead to data transfer but requires considerable effort from the user. The higher-level interface, UCP, is easier to use, but adds some overhead to the communication. This work focuses on charting the performance of UCX over InfiniBand, motivated by the usage of UCX as middleware for high-level communication libraries. We analyze performance shortcomings that stem from the two-level design and the sources of these performance losses. In particular, we target basic functions of UCP, evaluate their performance over InfiniBand, and analyze sources of overheads compared with UCT and Verbs. We propose and evaluate some fixes to minimize these overheads, in order to enhance UCP performance and scalability.**

***Index Terms*—communication software, communication middleware, UCX, InfiniBand, performance.**

I. Introduction

Modern HPC systems include extreme numbers of lightweight cores, deploying extremely low-latency interconnection networks. In order to exploit the capabilities of these upcoming architectures and to meet their demands in scalability, communication software needs to scale on millions of cores and support applications with adequate functionality to express their parallelism. Moreover, communication software should add as little overhead as possible in order to avoid compromising the native performance of the interconnection network. These requirements make the design of high-performance communication software extremely intricate, since they demand minimal memory requirements and low instruction counts and cache activity while meeting stringent performance targets.

High-level programming models for communication (e.g., MPI, UPC, SHMEM) can be built on top of middleware, such as Portals [1], GASNet [2], UCCS [3], and ARMCI [4] or use lower-level network-specific interfaces, often provided by the vendor. While the former offer high-level communication abstractions and portability across different systems, the latter offer proximity to the hardware and minimize overheads related to multiple software layers. An effort to combine the advantages of both is UCX [5], a communication framework for high-performance computing systems, which comes with a two-level API design.

The lower-level API, *UCT*, provides a unified API that abstracts communication functions for various low-level networks. At the upper level, UCX implements *UCP*, an API that exposes a collection of high-level protocols, such as tag matching, RMA, and atomic operations, and simplifies the initialization of communication. The two-level design of UCX offers portability across network fabrics. A third component, *UCS*, provides the necessary services for memory management, data structures, and more. Currently, UCX supports InfiniBand Verbs [6], Cray uGNI [7], and shared-memory devices (e.g., POSIX, KNEM).

MPICH [8], developed at Argonne National Laboratory and the most widely used MPI implementation, supports UCX in the MPICH 3.3 release series.[1] MPICH uses the UCP API because of its close match to MPI functionality; for example, Isend/Irecv operations are directly implemented by using UCP tag-matching functions. Apart from MPICH, other MPI implementations such as Open MPI, and an implementation of OpenSHMEM [9] have also utilized UCP as one of the communication APIs that they support. While UCT itself is less utilized directly by higher-level programming models, it is more performant than UCP. An implementation of UCT exists for InfiniBand through *libibverbs*, the OFED Verbs library that exposes the InfiniBand Verbs API. Additionally, UCT is also implemented with an accelerated version of Verbs, customized for the Mellanox MLX5 InfiniBand driver. Early results for UCT over InfiniBand [5] show that the accelerated version achieves high performance for remote direct memory access (RDMA) operations. UCP's broader abstractions and support for multiple transport layers add to programming ease and reduce the complexity of the code of the programming model implementations, making it a good target for higher-level programming models. The same attributes, however, also add overheads to UCP, compared with UCT, that may hinder both performance and scalability.

In this work, we describe the UCX design and study the performance of UCX over InfiniBand with regard to its utilization as a communication middleware for MPICH. We target core functions of UCP, aiming to understand the performance shortcomings of the two-level design and the tradeoffs from the utilization of UCP over UCT in performance, scalability, functionality, and programming ease. In particular, we focus on specific UCP functions and (1) compare their performance with the equivalent operations in UCT and Verbs API, (2) analyze their performance in terms of instructions and time,

[1] The alpha release of MPICH 3.3a1 is available at https://www.mpich.org/downloads/.

(3) identify sources of overhead in UCP compared with UCT, and (4) apply fixes in UCX to minimize these overheads. We evaluate the impact of these fixes and discuss optimizations in the UCX design that can enhance the performance and scalability of UCP, without compromising the programmability and functionality it offers.

II. Background

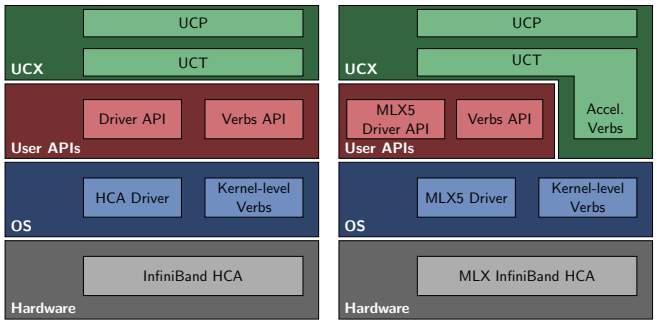
In this section, we provide background information on the InfiniBand interconnection network and the UCX design, which is key to analyzing and understanding the performance of UCX over InfiniBand.

A. The InfiniBand interconnection network

InfiniBand is a switched interconnection network, used for interconnecting large clusters and supercomputers. It is one of the most popular interconnection networks, as indicated by its share in the recent Top500 list,[2] which reached 37.4% in December 2016. It provides two-sided (send-receive) and one-sided (RDMA) semantics for communication. Communication over InfiniBand uses the queue pair (QP) model, where a send and a receive queue are used for issuing and receiving messages, respectively. A work request is submitted to these queues, where the hardware can read it to perform the communication. Additionally, a completion queue is associated with each queue pair for notification of communication completion. Communication over InfiniBand requires all memory regions that are accessed by the hardware to be registered. In order to alleviate the overheads of memory registration, short messages can be inlined in the work requests, whereas larger messages can take advantage of a zero-copy protocol. This strategy means that the work request gets only a description of the memory buffer and later reads the data directly from the buffer, without any CPU involvement.

The current InfiniBand fabric implements various transport mechanisms. The most common are the connection-oriented Reliable Connection (RC) and the connectionless Unreliable Datagram (UD). The latter implements only two-sided communication semantics. Furthermore, UD can transfer only one MTU of data (usually around 4 kilobytes) at a time. Therefore, UD usually offers lower bandwidth and higher latency compared with RC. On the other hand, RC has high demands in resources: to fully connect N processes, RC requires $O(N^2)$ connections and $O(N)$ queue pairs per process. UD is connectionless, and therefore only a single UD QP per process is required. To reduce the memory consumption of RC, the InfiniBand specification introduced shared receive queues, as well as the eXtended RC transport. Recently, Mellanox introduced the Dynamically Connected (DC) transport service, which dynamically creates and destroys connections, constraining the memory consumption close to the level of UD, while offering memory semantics, as RC. However, the scalable design of DC comes at a cost to performance, mainly because of connection transaction overheads [10].

[2]https://www.top500.org/



(a) Common InfiniBand OFED stack and UCX.

(b) MLX5 InfiniBand OFED stack and UCX with accelerated Verbs.

Fig. 1. InfiniBand OFED stack and UCX.

```
status = ucp_put(ep, buffer, size, remote_addr, rkey);   1
if (status != UCS_OK)
    exit(ERROR);                                         3
ucp_ep_flush(ep);
```

Fig. 2. RDMA write with remote completion in UCP

The user-space interface for InfiniBand is the Verbs API, which comes as a user-level library with the OFED stack and lies on top of the kernel-level Verbs API. The kernel-API collaborates with the vendor-specific InfiniBand driver and driver library to access the InfiniBand hardware. An overview of the InfiniBand software stack is depicted in Figure 1a.

B. UCX design

Figure 1a shows how the UCX software stack is placed on top of InfiniBand. UCX consist of two layers: the lower layer *UCT* and the upper layer *UCP*. In the following paragraphs, we discuss the main differences between these two layers, as well as the most important semantics inside UCX.

1) Communication context: The main difference between UCP and UCT is the communication context. UCT is designed as a communication layer over a *single* communication device and transport layer, whereas UCP abstracts the usage of different devices and transport layers for the user. Therefore, UCT defines a *memory domain* over a device (e.g., InfiniBand or shared memory) to allocate and register any memory used for communication and an *interface* for a specific transport over a specific device (e.g., UD and RC for InfiniBand). Both the memory domain and interface come with a set of their own attributes, derived from hardware capabilities. For example, memory domain attributes include memory allocation limits and memory access credentials, whereas interface attributes include the communication and connection capabilities of the transport mechanism and thresholds for protocol switching. UCP encapsulates these multiple UCT memory domains and interfaces in a single *communication context* and selects a suitable interface for a communication operation, according to hardware properties and performance criteria.

2) Communication primitives: The UCT API defines a set of functions for communication, such as remote one-sided and atomic operations and active messages. Thereby, different protocols can be defined for different message transfer methods. For example, for put/get communication immediate (*short*),

buffered copy (*bcopy*) and zero-copy message transfers can be defined. Not every function needs to be implemented for each transport layer, however; the implementation in effect depends on the hardware capabilities. UCP abstracts these low-level protocol functions into high-level functions: the UCP API offers functions for RMA operations, remote atomic memory operations, and tag matching. Tag matching currently is implemented by using UCT active messages. In the future, UCX will make use of hardware support for tag matching. UCP internally selects the appropriate transfer protocol and performs message fragmentation, if necessary. Therefore, UCP exposes a single function for any message size. Moreover, UCP offers nonblocking communication functions for tag matching and RMA operations, which allow immediate reuse of communication resource, whereas UCT provides only nonblocking operations.

3) Communication entities: Workers are the core communication entities in UCX, in both UCP and UCT. The main feature of a worker is that it has its own progress engine. This progress engine always enforces progress over all open interfaces. Progress can be made by calling the progress function *uct/ucp_progress* or by ordering or completion operations (fence/flush), which may require some progress to complete.

To enable communication with another worker, each worker creates an *endpoint* and connects it to the endpoint of the remote peer. A UCT endpoint is tied to a specific interface (e.g., UD or RC), whereas a UCP endpoint holds multiple UCT endpoints, one for each interface in use. Therefore, in UCP, an endpoint always connects two workers. Internally, UCP is responsible for selecting the best interfaces/UCT endpoints from those available to perform a communication operation.

4) Connection establishment: When a worker in UCP creates an endpoint, the UCP layer selects one or more interfaces for each type of operation and creates the respective UCT endpoints, one per interface, all associated with the *parent* UCP endpoint. In practice, UCP can select multiple interfaces for RMA and atomic memory operations, a single interface for active messages, and a single interface for *wireup*.

If an interface corresponds to a connectionless transport, then it can connect to the remote interface immediately. This is what happens in UCP: UCT endpoints over connectionless transports immediately establish connection. If the interface corresponds to a P2P transport, however, UCP creates a *stub endpoint*. The *wireup* UCT endpoint, which is always connectionless, undertakes connection of all UCT endpoints over P2P transports, by immediately sending wireup requests. The stub endpoint is eventually destroyed when all UCT endpoints of the parent UCP endpoint are connected.

III. UCX as middleware over InfiniBand

In this section, we discuss the suitability of UCX interfaces as communication middleware and examine their performance, in comparison with their network-specific counterparts.

A. Middleware tradeoffs among UCP, UCT, and Verbs

The requirements for communication middleware on large-scale systems are manifold: performance, scalability, portability, and functionality. Often, finding the right design can be seen as a tradeoff between these factors: better scalability can come with a loss in performance, and adding new functionality can limit the portability and add new overheads, which then result in lower scalability and/or performance.

While the two-level design of UCX attempts to address all the requirements, the choice of the right middleware results from a tradeoff among the desired properties. UCP offers high functionality, since it handles multiple transports, undertakes connection establishment, and selects the transfer method, all in a transparent way, while exposing high-level abstractions. Unlike UCP, the functionality of UCT is limited: its focus is to provide a unified API for low-level communication functions. Both APIs are portable, since UCT can be implemented for any network architecture. However, since UCT does not require that all functionality be supported by all architectures, its portability is limited. On the other hand, each API adds some overhead over the native network operations; the overhead increases with additional functionality. Therefore, one can expect that UCP has a lower performance than does UCT. Scalability is related to the memory requirements of the software, as well as the memory requirements of the transport.

The InfiniBand support for UCT is built on top of InfiniBand Verbs. Since it is closer to the hardware, we expect Verbs to have better performance than UCT and UCP have. However, the Verbs API does not provide any high-level functionality, and it is specific to InfiniBand and thus not portable to other architectures. Although it is theoretically possible to implement the complete communication of an application on Verbs, doing so requires significantly more programming effort and lines of code.

We note that UCX comes with a highly optimized UCT-Verbs implementation (Accelerated Verbs, hereafter denoted as AVerbs), using the latest Mellanox MLX5 driver. Figures 1a and 1b demonstrate the differences. Using normal Verbs, UCT lies on top of the common InfiniBand OFED stack. In contrast, the optimized UCT implementation over MLX5 directly accesses the driver functionality. Core communication functions are directly implemented inside UCT, which makes calls directly to the driver or utilizes the user-level driver library. This approach leads to fewer and smaller data structures and reduces branches and function calls, thereby increasing the performance of communication by minimizing the overhead.

Overall, the optimal middleware for InfiniBand would be scalable and combine the near-native performance of Verbs with the high-level abstractions and portability of UCP. We argue that in order to maximize all desired properties, the key is not to rely on lower-level middleware but to identify sources of overhead in upper-level functions and work toward their minimization—and, if possible, their elimination. To this end, we have conducted experiments to compare the performance of the various APIs.

B. Experimental setup

For our experiments, we use two nodes of two Intel Xeon E5-2699 CPUs, interconnected over InfiniBand using

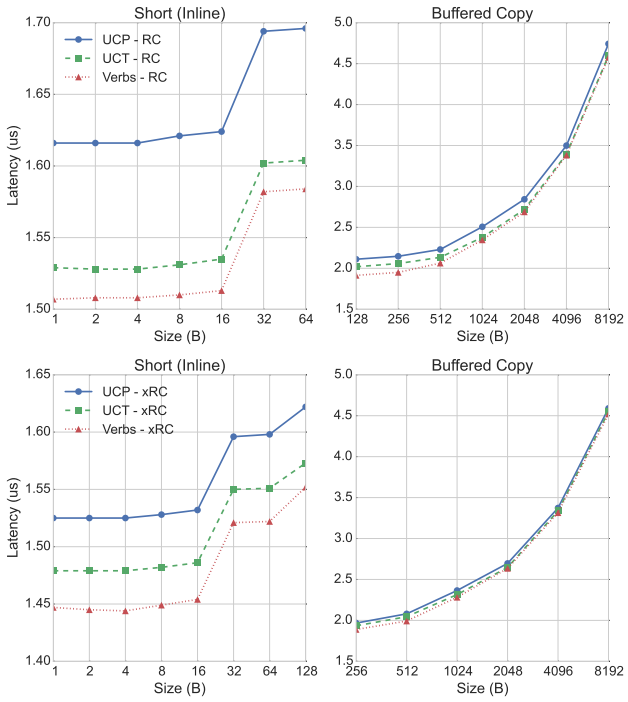


Fig. 4. RDMA write latency over InfiniBand xRC

a Mellanox InfiniBand EDR HCA of the MT27700 Family (ConnectX-4) and a Mellanox FDR-10 switch. We base our experiments on UCX version 1.0, (7/13/2016 snapshot), compiled with the Intel C++ Compiler 16.0.0. We compile UCX with `-O3` optimization, interprocedural optimization, inlining, and enabled AVX/SSE4.2 instructions, so that UCX libraries have minimum instruction count. We use the Intel Software Development Emulator v7.48 for instruction analysis.

C. Assessing the performance of UCP, UCT, and Verbs

To assess the performance overheads added by UCP over UCT and UCT over the lower-level API, Verbs, or AVerbs of UCX, we compare the three APIs in terms of latency for an RDMA write operation with remote completion over InfiniBand. The operation consists of two parts, In the first part, the communication is started (and for blocking operations locally completed). We refer to this part as *Put*. The second part, referred to as *Flush*, guarantees remote completion of all

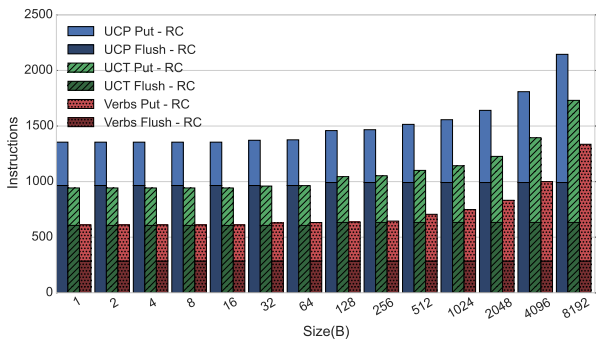


Fig. 5. Instruction breakdown for RDMA write over InfiniBand RC

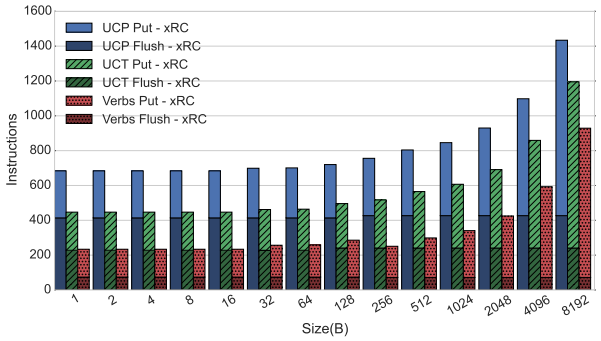


Fig. 6. Instruction breakdown for RDMA write over InfiniBand xRC

previous issued operations. Figure 2 depicts the implementation of this operation in UCP.

The comparison for RDMA write over InfiniBand RC is depicted in Figure 3. To avoid comparison of different transfer protocols (e.g., short, buffered, and zero-copy), we use the same transfer protocols and thresholds for UCT and Verbs as are used by the UCP layer. UCP uses the *short* protocol for messages of length up to 98 bytes for RC and buffered transfers (*bcopy*) for larger messages. Internally, UCT deploys the InfiniBand *inline* option for small messages and performs memory copy operations to preallocated buffers for buffered copies. The buffer size can be configured by the user. In this and all our subsequent experiments, we use the default value of 8 kilobytes. If the message length exceeds the buffer length, it is fragmented into multiple chunks of 8 kilobytes. For comparison, we imitate this behavior in our UCT and IB-Verbs implementations. UCT also implements functions for zero-copy transfers, although in its current version UCP does not utilize those for RDMA operations.

Figure 3 shows that UCP adds some non-negligible latency over UCT, especially for short messages. However, the overhead added by UCT over Verbs is minimal and becomes negligible for larger message sizes. We also compare RDMA write over InfiniBand xRC, that is, the AVerbs implementation in UCT for RC. In Figure 4 we observe that the overall latency drops in comparison with RC. Similar to RC, UCP adds some overhead over UCT, and UCT adds some overhead over AVerbs, although the difference between UCP and UCT is smaller. In addition, for buffered transfers (messages larger than 220 bytes), UCP overheads over UCT become negligible.

To understand the origin of this difference in performance, as a first step we compare the number of instructions for the *Put* and *Flush* operations for the three APIs. Figures 5 and 6 show the instructions needed for RDMA write over InfiniBand RC and xRC, respectively. First, we observe that the difference in latency between the three interfaces in both cases stems from an equivalent difference in instructions. Second, the number of instructions for xRC is less than half that for RC, although this decrease does not translate to a similar decrease in latency, since most of the latency is spent on the network. Thus, even with xRC, hardware latencies render the software overheads insignificant. Third, the number of instructions for *Put* increases with the message size in all cases: the additional

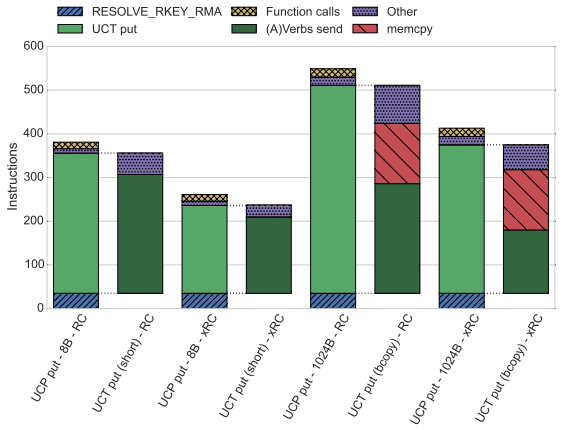


Fig. 7. Instruction analysis for UCP Put

instructions are consumed for memory copies, as the message size grows. This breakdown of instructions to *Put* and *Flush* leads us to important observations about overheads in UCX:

- For small message sizes the majority of instructions in UCP and UCT are consumed by *Flush*, while in Verbs (and AVerbs) more instructions are consumed by *Put*.
- We observe a slight decrease in instructions for *Put* and a significant decrease in instructions for *Flush*, from UCP to UCT to Verbs (and AVerbs). Hence, we can infer that UCX adds significant overheads to the *Flush* operation, which inflate from UCT to UCP, and smaller but not negligible overheads to the *Put* operation.

These observations motivate us to look further into the differences of the three APIs and to identify the sources of overhead in the UCX software stack.

IV. Analyzing UCP overheads

To understand the performance shortcomings of UCX, we analyze core UCP functions in terms of instructions, to identify necessary overheads added by UCP providing high-level abstractions and UCT providing a unified API, and unnecessary overheads, which can be avoided with modifications in the UCX software design.

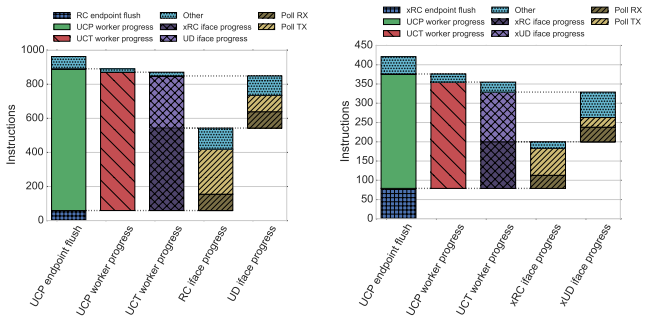
A. RDMA operations

The UCP API provides functions for blocking and nonblocking RDMA write and read. The blocking version of the functions (`ucp_put` and `ucp_get`) first selects the appropriate UCT interface from those available, then selects the transfer protocol (short buffered copy or fragmented buffered copy) and calls the respective UCT function. The nonblocking versions (`ucp_put_nbi` and `ucp_get_nbi`) also select the interface and transfer method. In contrast to the blocking version, however, the nonblocking version does not necessarily call the underlying UCT function; instead it may push a request for the operation in a queue, which is handled later. A request is created under two occasions: if no resources are available at the time of the function call or if the message is longer than a preset size. Here, we analyze the blocking version of RDMA operations; the nonblocking version adds a de facto overhead associated with the request creation and queue insertion. The goal is to determine the minimal overhead that is added because of the UCP layer. We focus on `ucp_put`; however, we can generalize for `ucp_get`, since the structure of the function is identical.

Figure 7 shows the instruction analysis for `ucp_put` for a short message (8 bytes) and a longer message (1,024 bytes), transferred with buffered copy, over InfiniBand RC and xRC (RDMA operations are not supported over UD and currently are not emulated in UCX). Most of the instructions in `ucp_put` are spent on the UCT operations, for both message sizes and transfer methods. Similarly, most of the instructions in UCT are spent on the AVerbs or Verbs send call. There is an extreme decrease in instructions from Verbs (RC) to AVerbs (xRC). Part of the difference results from the assimilation of the Verbs functionality inside UCT. This means that some operations that were previously executed in the Verbs layer are moved to the UCT layer. For example, the network-specific work request is directly created in the UCT layer. This approach avoids the overhead of first creating a Verbs work request and translating this to the network-specific work request later. We count these operations as *other* instructions. Accordingly, `uct_put` spends some instructions on work-request creation and other bookkeeping operations.

The decrease in instructions in (A)Verbs from short to longer messages is artificial: the 8-byte message is sent inline, so the memory copies take place within the (A)Verbs send call. Focusing on `ucp_put`, we see that 15 (for short) to 19 (for buffered copy) instructions are spent on *function calls*. UCP uses function pointers to acquire the appropriate UCT function for the transport in use. This design adds some flexibility, since it allows UCX to choose the best UCT endpoints dynamically during runtime and also allows the support of multiple transport layers, such as InfiniBand and shared memory, at the same time. However, this flexibility comes with some cost, since function pointers cannot be optimized with inlining or interprocedural optimizations. In addition, 6 (for short) to 15 (for buffered copy) instructions are spent on checks for the length of messages and fragmentation handling. Four instructions can be avoided by setting a UCX configuration flag to avoid checks (usually set for the installation configuration).

A large number of instructions (35), however, come from the `RESOLVE_RKEY_RMA` function. Many of these instructions are also expensive (such as the modular operation). As explained in Section II, a UCP endpoint holds multiple UCT endpoints, one for each interface/transport method. If a *Put* operation is initiated, the UCP layer has to decide which transport layer to use. For this decision, UCP holds a map of indexes to look up transport methods that are suitable for RMA operations. For example, one-sided communication is supported for shared memory and (x)RC endpoints but not for (x)UD endpoints. In addition, for RMA operations, UCP holds a bundle of UCT keys for access to the remote memory region. Each key belongs to the memory domain of the transport method. For example, if UCP is used for communication over InfiniBand and shared memory, the UCP



(a) UCP endpoint flush over RC (b) UCP endpoint flush over xRC

Fig. 8. Instruction analysis for UCP endpoint flush after a single UCP Put operation (8B)

key may hold a key for each of the two memory domains. The function `RESOLVE_RKEY_RMA` resolves the UCP endpoint to a UCT endpoint, using the UCP memory key. It determines which UCT endpoint should be used for this communication operation and returns the corresponding configuration, such as thresholds for switching from short to buffered copy and the minimum length of the preregistered buffer for buffer copy.

Again, this design was chosen to provide some flexibility during runtime. One may argue that, for example, shared memory should always be used for intranode communication and that InfiniBand should be used for internode communication, and, therefore, the required information should be saved in the endpoint for all RMA or AMO operations. Yet, this approach would force the user to always use the same communication protocol, which is not always possible. One example is the handling of user-allocated buffers, which cannot be registered for shared-memory communication over most UCT shared-memory interfaces (except for XPMEM), while InfiniBand (and thus the UCT-InfiniBand interface) allows this. Still, we argue that the `RESOLVE_RKEY_RMA` function is expensive and adds unnecessary overhead to UCP *Put* and *Get* operations, since this information becomes available at an earlier time. It is thus not necessary to redo the computation for every new communication operation. We discuss this overhead and its optimization in the next section.

B. Progress / Flush operations

In order to explicitly make progress on communication operations in UCP, `ucp_worker_progress` has to be called. In addition, UCP provides *Flush* operations for the worker and the endpoint, namely, `ucp_ep_flush` and `ucp_worker_flush`, which ensure that all one-sided local and remote communication operations are completed on the local side. Note that blocking functions internally also call progress, if no communication resources are available. We focus our analysis on the `ucp_ep_flush` function, since we can generalize its case for the flush operation on the worker.

Figure 8 depicts the instruction analysis for `ucp_ep_flush` over RC and xRC, after a single RDMA write operation of an 8-byte message with `ucp_put` over RC and xRC, respectively. We ensure that no other operations are outstanding, to avoid counting additional poll instructions. The call path is the same for both interfaces, RC and xRC. Since a UCP endpoint can have multiple UCT endpoints, `ucp_ep_flush` iterates over all UCT endpoints and initiates a flush operation on the interface on which the endpoint is attached. In our execution configuration, we initiate UCP to use only (x)RC endpoints; thus, UCP should flush only the (x)RC interface. The interface flush operation detects whether any communication operations are in progress. If so, progress is called on the UCP and subsequently the UCT worker.

The UCT worker progresses all open UCT interfaces. Our analysis reveals that, besides the (x)RC interface, progress is called on the (x)UD interface. The interface progress always polls first the receive completion queue (RX queue) and subsequently the send completion queue (TX queue). Since in our case all communication functions are completed after calling *progress* once, a final call for flush on the (x)RC interface returns with completion, so the operation is completed. If this is not the case, the flush operation calls progress until all outstanding operations are completed. Note that the UCX working group has decided to add a nonblocking flush operation, although this operation was not implemented at the time of this work. The sequence of calls on the UCP endpoint flush reveals how UCP handles multiple endpoints and interfaces. It also shows that performance is influenced by the way UCP handles initialization and connection establishment.

If UCX is used with InfiniBand, UCP always utilizes the UD interface for wireup, since it is connectionless, whereas the (x)RC interface requires a connection to start any data transfer. Whenever a UCP endpoint is created, UCP creates only a *stub endpoint*, which then creates an auxiliary UCT endpoint over (x)UD. The (x)UD interface is then used to establish the connection of the *actual* UCT endpoint over (x)RC. Note that the (x)UD endpoint currently is needed only for wireup, because, with the current criteria of UCP, (x)RC is always the selected endpoint for all InfiniBand-supported operations: RDMA and active messages/tag-matching. When the wireup is complete, the UD (xUD) endpoint is destroyed. However, the (x)UD interface is still open, and progress is called over it. Progress over the UD interface consumes 306 instructions, and progress over the xUD interface consumes 129 instructions. These results explain the difference in latency between UCP and UCT for RDMA write, demonstrated in Section II: in UCT, we create and connect endpoints directly over the (x)RC interface—the (x)UD interface is never opened. In the following section, we discuss when this overhead is necessary, and we propose an optimization.

A second observation is that the receive queue (RX queue) is polled, although we perform only RMA operations, which are always completed in the TX queue. This overhead is necessary to fully support UCP functionality, since the progress engine needs to check for received active messages. However, this overhead it is not present in our Verbs (and AVerbs) benchmarks for RDMA write latency, thus explaining part of the performance difference between UCT and Verbs (or AVerbs).

Comparing Verbs and Averbs, we note that TX and RX

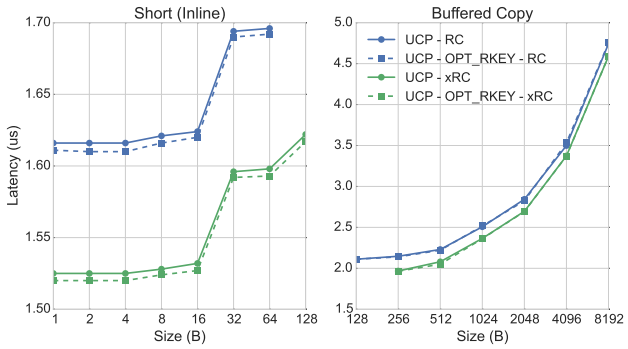


Fig. 9. RDMA write latency in UCP with optimized remote key resolution

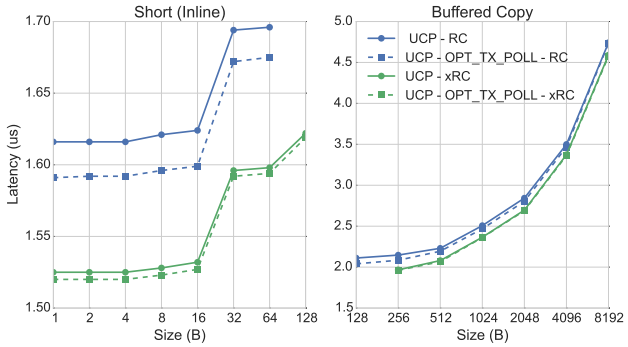


Fig. 10. RDMA write latency in UCP with optimization to avoid TX Polling

polling over UD consume an equal number of 96 instructions, in contrast to UD, where they consume only 38 and 25 instructions, respectively. Additional instructions (measured as *other*) within the xUD interface progress function correspond to polling for receive requests over an additional queue, used for support of IPoIB along with RX polling. Overall, however, the instruction count for polling the RX and TX queues is significantly decreased for AVerbs, compared with Verbs.

V. Optimizing UCX performance for InfiniBand

Based on our analysis of core UCP functions, we apply optimizations in UCX to improve its performance over InfiniBand, and evaluate their impact on overall UCP performance.

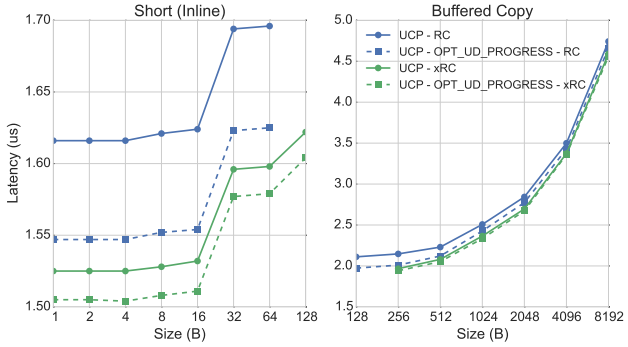


Fig. 11. RDMA write latency in UCP with optimization to avoid progress over the (x)UD interface

A. Optimizations for UCX performance

1) Optimizing RDMA functions in UCP: For RDMA functions, we focus on reducing the overhead coming from the `RKEY_RESOLVE_RMA` function. Two key observations enable this optimization. First, the UCT endpoint and UCT remote key used for RMA operations become available prior to calling an RMA communication function. In UCP, a worker that wishes to perform operations that require registered memory on the remote side (RMA or atomic memory operations) needs to acquire the UCP remote key of the remote registered memory. Once this key is received, it has to be unpacked, by using the `ucp_rkey_unpack` function. This function checks, unpacks, and stores the UCT remote keys for reachable remote interfaces in the UCP remote key structure. Second, it is not necessary to perform the resolution from a UCP to a UCT key with each call to an RMA communication function, since the interface, the corresponding UCT endpoint and UCT key are static after the remote key is unpacked.

To reduce the number of instructions spent on the UCP-to-UCT translations for RMA operations, we call the `RKEY_RESOLVE_RMA` function in the `ucp_rkey_unpack` function. In this way, the operation is performed only once and is no longer on the critical path of communication. To make the outcome of the function available in RMA communication functions, we store the index for the UCT remote key, the RMA configuration, and the UCT remote key for RMA with the `ucp_rkey` data structure. Since the only other types of operations that require a remote key are atomic memory operations, we also store the AMO configuration and the UCT remote key for AMO with the `ucp_rkey` and eliminate the bundle of UCT remote keys from the structure, along with the map of reachable memory domains. Unfortunately, this change duplicates the information for RMA and AMO configurations, which is also stored with the `ucp_ep` data structure. However, this duplication at the moment allows us to save multiple instructions. RMA (and AMO) communication functions access only the index for the UCT endpoint, the RMA (or AMO) configuration, and the UCT remote key. Storing the configuration with the UCP remote key makes access to it faster, since accessing it in the UCP endpoint requires multiple expensive pointer operations.

2) Optimizing the UCP progress engine: As we showed in Section III, overheads in the UCP progress engine are related to the way UCP handles multiple interfaces over InfiniBand and connection establishment in UCP. We now examine the case where the only device is InfiniBand, all communications take place over RC or xRC, and the wireup transport/interface is always UD or xUD. Our optimization suggestions, however, can also prove useful for multiple transports. In the optimal case, we would like to be able to eliminate the overhead from progressing the UD interface, since it is not used for communication. If a new endpoint is created, however, a request for wireup can arrive at any time, Therefore, the progress engine must ensure progress of the (x)UD interface.

We should not eliminate polling on the receive queue for

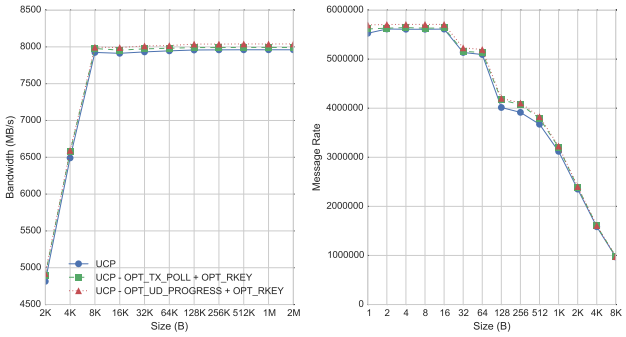
the connectionless interface, because we cannot ensure that an active message will not arrive. Nevertheless, we can eliminate polling on the send queue (TX queue) for any interface, if no messages are sent. Initially, we attempted to perform this optimization by exploiting existing bookkeeping structures in UCT, which count the number of sent messages and would permit us to skip TX polling when this number is zero. However, we found inconsistencies in the value of those counters for the (x)UD interface. Therefore, we take a different approach. In particular, we modify the `uct_iface_t` data structure in UCT, adding a counter of "active" endpoints over the interface. The counter is incremented when a new endpoint is connected and decremented when the endpoint is destroyed. In this way, we can avoid polling the TX queue of an interface if no endpoints are connected over it. We note that this approach results in a more restrictive condition on allowing avoidance of TX polling, and we aim to revisit our original approach in future work.

A deeper study into UCP initialization and wireup process revealed that the current UCP implementation permits the elimination of progress over the (x)UD interface entirely, under certain conditions. As explained in Section III, when using InfiniBand, currently the (x)UD interface is used only for wireup, in other words, for the connection of created UCP endpoints. In the ideal case, a UCP endpoint should be created but never connected until a communication operation is called. In its current design, however, UCP creates stub endpoints for each UCP endpoint, which initiate the wireup of the actual endpoints over the (x)UD interface as soon as the endpoint is created. The wireup requests are enqueued in the respective TX/RX queues upon initiation; and thus, when a worker calls its progress engine (upon `ucp_worker_progress` or `ucp_worker_flush` or `ucp_ep_flush`), any endpoint associated with the worker may be wired up, even if no communication takes place over it. This approach results in all endpoints being connected after a few calls to the progress engine. The stub endpoint is then destroyed, along with the (x)UD endpoint associated with it.

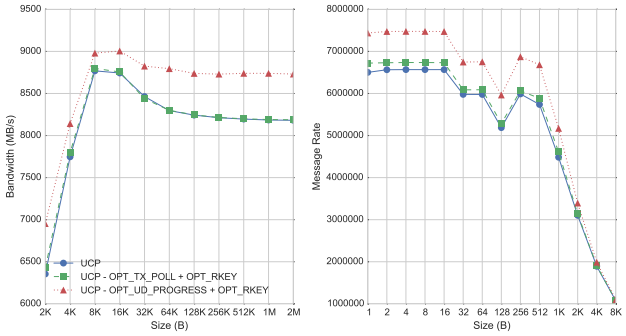
This design guides our second optimization, which is the elimination of progressing the (x)UD interface after wireup. To enable the optimization, we use the counter for "active" endpoints we constructed for avoiding TX polling. In this case, however, when no endpoints are connected over the (x)UD interface (or any other interface under the same condition), the counter value is zero, and the interface is not progressed.

B. Evaluation

To evaluate the overall UCP performance and the impact and potential of our optimizations, we first evaluate each optimization on the latency and instruction count of RDMA write with remote completion. Subsequently, we measure the bandwidth and message rate for RDMA write (*Put*) and RDMA read (*Get*) in UCP, as well as in two optimized versions: one that eliminates the remote key translation and unnecessary TX polling and one that eliminates the remote key translation and unnecessary progress over the (x)UD interface.

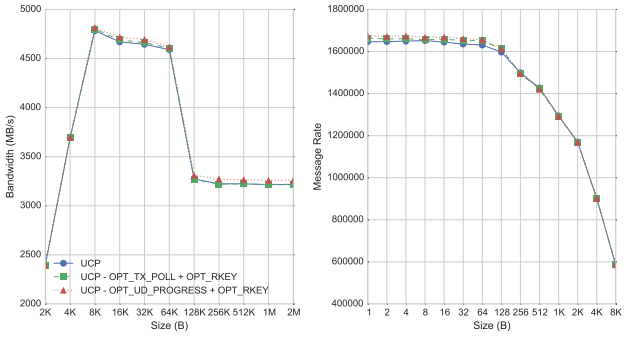


(a) RDMA write over RC

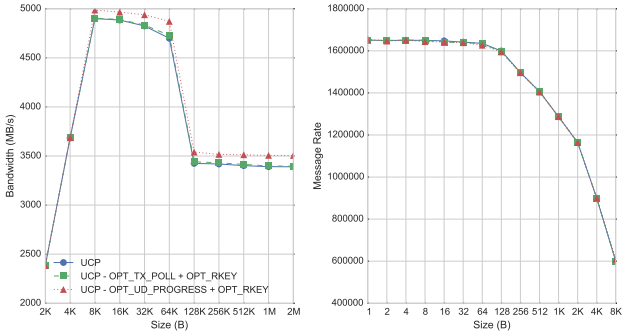


(b) RDMA write over xRC

Fig. 12. Evaluation of bandwidth and message rate with UCP RDMA write



(a) RDMA read over RC



(b) RDMA read over xRC

Fig. 13. Evaluation of bandwidth and message rate with UCP RDMA read

For this benchmark, we use the nonblocking functions of UCP for *Put* and *Get* operations, namely, `ucp_put_nbi` and `ucp_get_nbi`. We issue 256 operations (equal to the default length of the TX queue in UCX) before calling `ucp_ep_flush` for local and remote completion.

Figure 9 shows the impact of eliminating the RESOLVE_RKEY_RMA function on the latency of RDMA write with remote completion in UCP (performed with ucp_put and ucp_ep_flush). Although we observe a decrease in latency, it corresponds to less than 1% and is thus negligible. In all cases, the optimized version eliminates 27 of the 35 instructions of the RKEY_RESOLVE_RMA function; 8 instructions are still consumed to access the UCT endpoint index, remote key, and RMA configuration in the ucp_rkey data structure. We argue that a redesign of UCX data structures such as ucp_ep and ucp_rkey can help eliminate any overhead from UCP-to-UCT resolutions.

Figure 10 depicts the effect of avoiding unnecessary polling of the TX queue on the latency of RDMA write with remote completion in UCP. We notice a significant performance difference in latency for RC and a less significant performance difference for xRC, since TX polling is faster in the case of AVerbs. Figure 11 demonstrates the effect of avoiding unnecessary progress on the (x)UD interface on the latency of RDMA write, where we notice a significant decrease in latency for RC for all message sizes and a smaller but still noteworthy decrease in latency for xRC. As for the instruction count, the instructions spent on ucp_ep_flush, called following a single ucp_put operation for an 8-byte message, decrease by 12% when TX polling is avoided and by 29.8% when progress over the UD interface is omitted, for communication over the RC interface, In contrast, for the xRC interface, the equivalent decrease is 4.8% and 26.6%.

Figure 12 shows the bandwidth and message rate for RDMA write using RC and xRC, respectively. Looking at the reference UCP, we see in both cases that the highest bandwidth is achieved at 8,192 bytes, which equals the length of the internal buffer used for buffered transfers. Beyond this point, the messages are fragmented, and thus the overall number of requests exceeds the length of the TX queue, causing additional polling for completion and the creation of more requests. For the RC interface, the bandwidth remains constant, since the *Put* operations take up a larger percentage of instructions and time, hiding any latency due to polling. For the xRC interface, the analogy is reversed. A larger percentage of instructions is spent on polling functions, hence there is a decrease in bandwidth.

The message rate is constant for all sizes up to 8 bytes, then drops up to 64 bytes for RC and 128 bytes for xRC because of additional memory copies inside the Verbs (AVerbs) functions. As the transfer methods switches from *short* to *bcopy*, for xRC, the message rate increases at 256 bytes before dropping again, revealing that the switching point is suboptimal.

The bandwidth and message rate with our optimizations follow the same behavior, since the optimizations have a flat impact on the instruction count. By eliminating the remote key translation and avoiding TX polling, we optimize the bandwidth and message rate by up to 4% over RC and up to 2.8% for xRC. The impact of eliminating the remote key translation is more significant for medium-size messages that are transfered with buffered copies, that is, within the range of 128 bytes up to 8,192 bytes. In *bcopy* transfers, the function RKEY_RESOLVE_RMA is called twice due to request creation. Avoiding polling the TX queue is more effective on RC, where it takes up a larger percentage of instructions than in xRC.

On the other hand, avoiding progress on the (x)UD interface has a larger impact on performance, especially for xRC, where we notice an improvement of about 14% on the bandwidth and message rate of small messages and 6–7% on the bandwidth of large messages, in combination with the elimination of the remote key translation. We note here that this larger impact is an immediate result of the times progress over the xUD interface is called in the unoptimized version, since it is more often called for the xRC case than for RC. In the case of communication over RC, when progress is called, the TX/RX queues are polled for a number of requests, which can be set at runtime by the user. We use the default value, which is preset to 16 requests. In contrast, in the case of communication over xRC, the TX/RX queues are polled only for a single request. Subsequently, the ucp_ep_flush operation on the 256 *Put* operations issued by our benchmark calls ucp_worker_progress (and progress over the xUD interface alongside) at least 256 times in the case of xRC, compared with at least 16 times in the case of RC.

In Figure 13, we evaluate the bandwidth and message rate for RDMA read. The highest bandwidth is achieved for message sizes of 8,192 bytes, however, it drops beyond this point both for the RC and xRC interfaces. It also drops significantly at the message size of 128 kilobytes, where the aggregate volume of messages read from the remote target exceeds the last level cache size. Since the nonblocking get operation in UCP uses buffered transfers for all message lengths, the message rate drops gradually with the message size, as the number of memory copies increase. We note that reading from a remote target is a round-trip tour to the target side and hence consumes more time on the network and the target side than on the CPU of the side that initiates the operation. Elimination of the function RKEY_RESOLVE_RMA therefore does not improve the performance of the *Get* operation, since it does not consume a significant part of the time spent on the operation. Even combining this optimization with the avoidance of polling the TX queue, no significant improvement is achieved. However, by combining this optimization with the avoidance of progressing the (x)UD interface, we improve the bandwidth and message rate by about 1.5% for the RC interface and by up to 3.5% for the xRC interface. We conclude that any possible optimizations for the *Get* operation in UCP should target the memory copies involved and the overall cache utilization, rather than the instruction count.

VI. Discussion and outlook

Adding functionality to a communication layer often does not come for free, since new functionality usually comes along with new overhead. Still, our study on UCX reveals that there often is room for optimizations that can close the performance gap between between two layers, in our case UCP and UCT, over InfiniBand, without sacrificing any of the functionality or programming ease of the upper layer, UCP.

We performed our optimizations without any major redesign in the UCP or UCT layers, while maintaining the size of UCP data structures, so as not to compromise scalability. Therefore, we were not able, for example, to fully eliminate the instructions spent on UCP-to-UCT translations, or to avoid function pointers. We believe that a careful redesign of UCP data structures can improve and possibly eliminate this as well as other overheads in the handling of multiple UCT interfaces by UCP. We also find the utilization of function pointers from the UCP to the UCT layer problematic, since they increase the cost of calling UCT functions from UCP, a cost that cannot be eliminated with any compiler optimizations. We thus advocate a major restructuring of the UCX code to avoid this practice.

We showed that unnecessary polling of the send queue (TX queue) can easily be avoided in order to improve latency and bandwidth. Another possibility to reduce the overhead could be using a single queue for sends and receives, in order to avoid unnecessary polling under any case. We demonstrate, however, that performance can gain a significant boost by avoiding progress over interfaces that exist but are not in use for the specific operation, as is the case of the (x)UD interface. This practice should be generalized and enforced in UCP, since in a setup with multiple transport layers and multiple interfaces can exist, whereas only a single or a few may be in use at a given time. This is especially useful because UCP allows configuring at runtime which communication protocols should be used for a given context. In MPI, if the implementation uses more than one UCP context, it can be used to allow more scalable communication. For example, one context can be used for one-sided communication on a specific window, while another can be used for point-to-point communication in the global MPI world. That, however, can be implemented efficiently only if the UCX worker progresses only those resources used for the specific context. Moreover, the option to poll a TX/RX queue for more than a single request should be enabled for the xRC interface, to avoid calls to progress functions over the multiple interfaces.

A significant problem for scalability in the design of UCX is the current initialization/wireup process, which forces all endpoints associated to a worker to connect, as soon as the worker progresses once or a few times. This approach can greatly impede scalability in a setup of thousands or millions of processes, where UCP could easily result in an all-to-all connectivity. In such a scenario, each process would maintain connections and all the involved data structures with all other processes, even if it would never use these connections. We advocate a redesign of the UCP initialization process, where endpoint connections will be created on-demand only if communication over the endpoint is initiated. The current design can be useful in small-scale setups, where all-to-all connectivity can be enforced over a connection-oriented interface, such as (x)RC on InfiniBand, eliminating the overheads of an additional interface for wireup.

VII. Conclusions

In this paper, we studied the design and performance of UCX on InfiniBand, motivated by its utilization as communication middleware for MPICH. We measured and compared the performance of core functions in the upper layer of UCX, the UCP API, with the performance of lower-level layers, such as the transport layer of UCX, UCT, and the Verbs API. We analyzed the instruction counts of these functions to identify the sources of overheads; and we applied minimal optimizations to alleviate these overheads, in order to enhance the performance of UCP without altering its functionality. Our study of the UCP design and our evaluation show that the UCP layer can be improved in terms of performance and scalability, to mitigate the performance degradation that comes from offering high-level abstractions of communication functions and transparently handling multiple transport layers.

Acknowledgment

This material was based upon work supported by the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357. Nikela Papadopoulou has received funding from IKY fellowships of excellence for postgraduate studies in Greece-SIEMENS program. We gratefully acknowledge the computing resources provided and operated by the Joint Laboratory for System Evaluation (JLSE) at Argonne National Laboratory. We thank Gail Pieper for proofreading the paper.

References

[1] B. Barrett, R. Brightwell, R. Grant, S. Hemmert, K. Pedretti, K. Wheeler, K. Underwood, R. Riesen, A. MacCabe, and T. Hudson, "The Portals 4.0. 2 Networking Programming Interface, Sandia National Laboratories," October 2014, Tech. Rep. SAND2014-19568, Tech. Rep.

[2] D. Bonachea, "GASNet Specification, v1. 1," 2002.

[3] P. Shamis, M. G. Venkata, S. Poole, A. Welch, and T. Curtis, "Designing a high performance openshmem implementation using universal common communication substrate as a communication middleware," in *Workshop on OpenSHMEM and Related Technologies*. Springer, 2014, pp. 1–13.

[4] J. Nieplocha and B. Carpenter, "ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems," in *International Parallel Processing Symposium*. Springer, 1999, pp. 533–546.

[5] P. Shamis, M. G. Venkata, M. G. Lopez, M. B. Baker, O. Hernandez, Y. Itigin, M. Dubman, G. Shainer, R. L. Graham, L. Liss *et al.*, "UCX: An Open Source Framework for HPC Network APIs and Beyond," in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. IEEE, 2015, pp. 40–43.

[6] Mellanox Technologies Inc., "Mellanox IB-Verbs API (VAPI)," 2001.

[7] Cray Inc., "Using the GNI and DMMAP APIs," Cray Software Document S24463103, 2010.

[8] Argonne National Laboratory, "MPICH - high-performance, portable MPI," http://www.mpich.org, 2016.

[9] M. Baker, F. Aderholdt, M. G. Venkata, and P. Shamis, "Openshmem-ucx: Evaluation of ucx for implementing openshmem programming model," in *Workshop on OpenSHMEM and Related Technologies*. Springer, 2016, pp. 114–130.

[10] *Adaptive transport service selection for MPI with InfiniBand network*. ACM, 2015.