Enabling One-sided Communication Semantics on ARM

Pavel Shamis
ARM Research
Austin, TX
pavel.shamis@arm.com

M. Graham Lopez
Computer Science and Mathematics Division
Oak Ridge National Laboratory
Oak Ridge, TN
lopezmg@ornl.gov

Gilad Shainer
Mellanox Technologies
shainer@mellanox.com

***Abstract*—In this paper, we present our work to enable optimized one-sided communication operations on the ARM v8 architecture using a high-performance InfiniBand network interconnect, as well as an evaluation of our implementation. For this study, we started with an OpenSHMEM implementation based on Open MPI/SHMEM, and combined it with the UCX framework and the XPMEM kernel extension for shared memory communication. UCX is a unified communication abstraction that provides high-performance communication services over a variety of network interconnects and shared memory technologies. The UCX, XPMEM, and OpenSHMEM components were specially ported for this work in order to enable efficient access to shared memory and RDMA network capabilities on ARM. To the best of our knowledge, this is the first investigation of one-sided communication semantics and OpenSHMEM on the ARM architecture combined with a high-performance InfiniBand network and XPMEM shared memory transport.**

I. Introduction

One-sided communication is a class of semantics where the initiator instantiates a communication request that can be completed without explicit involvement of the processing element on the target side. Remote memory put (write), get (read), and atomic updates are examples of such operations. Conversely, two-sided communication semantics requires explicit involvement of the initiator and target sides. Typical examples for this class of semantics are send-receive operations where the target must explicitly invoke the receive operation for send to be completed.

SHMEM is a library-based implementation of a partitioned global address space (PGAS) programming model that is built on one-sided communication semantics and a core set of collective, atomic, and synchronization operations. Although there have been many high-quality, proprietary implementations of the SHMEM API, OpenSHMEM [1], [2] is an open standard that attempts to normalize the syntax and features of SHMEM for consistency across platforms and implementations. The 1.0 version of the OpenSHMEM specification is based on SGI's SHMEM API, and with input from other industrial partners, academia, and national laboratories the specification is about to release v1.4 with further regularization and new features. In addition to an open standard, OpenSHMEM provides an open-source reference implementation [3] of the SHMEM API for research purposes.

For this work, we chose to use the OpenSHMEM library as a representative of one-sided communications due to its being relatively lightweight and simple; the same work is readily transferable to MPI and other one-sided implementations as well. The OpenSHMEM reference implementation is designed for maximum portability to enable evaluation and research on the widest variety of platforms. The original version was based on the GASNet [4] communication middleware layer, and later it was redesigned [5] to the Universal Common Communication Substrate (UCCS) [6], [7] low-level network library. The successor to UCCS is named UCX [8], and efforts are underway to further improve the performance of various communication libraries using this next-generation network layer. For instance, there is an OpenSHMEM implementation built on UCX that is now included as part of the Open MPI [9] software distribution. For the remainder of this paper, we will refer to the OpenSHMEM implementation included with Open MPI as `OSHMEM`. With the increased support afforded by the Open MPI project and partnerships with industry, as well as the wider portability and increased performance afforded by the adoption of UCX, OSHMEM is beginning to see greater success on a wider variety of platforms.

We present a preliminary evaluation of one-sided communication on the ARM v8 architecture using an InfiniBand (IB) network interconnect. In order to enable this evaluation we ported the UCX framework and XPMEM kernel module to support the ARM v8 architecture. Moreover, we developed an ARM-optimized version of OpenSHMEM point-to-point synchronization routines by leveraging the advanced capabilities of the ARM instruction set architecture (ISA). To the best of our knowledge, this is the first research work

that enables and demonstrates advanced shared memory and RDMA capabilities on the ARM platform.

In this paper, we give a brief overview of the UCX communication layer, the OpenSHMEM programming model, and related work in these areas in section II. Section III goes into the main details about UCX, XPMEM, and how they were integrated with OSHMEM. We present the results of our evaluations in section IV along with a comparison between the MLX5 and Verbs capabilities in the UCX InfiniBand transport layer and the effects of using ARM ISA optimizations for OpenSHMEM point-to-point synchronization operations. Finally, we provide a brief discussion and some concluding remarks in section V.

II. Background and Related Work

Once the OpenSHMEM specification was started, a reference library implementation [10] was provided by a collaboration between Oak Ridge National Laboratory and University of Houston. Soon after, a variety of research and vendor implementations emerged. Research driven implementations include Ohio State University's version [11] based on the MVAPICH-X runtime, Sandia National Laboratory open source implementation for the Portals [12] network abstraction and libfabrics [13] interface, and GSHMEM [14] from the University of Florida based on GASNet [4]. Vendor oriented implementations that are closely related to specific network hardware include SGI, Cray and HP SHMEM, TSHMEM [15] developed by University of Florida for Tilera many-core processors, and Mellanox ScalableSHMEM [16]. Several of these implementations include various extensions that are not defined in the OpenSHMEM specification.

More recently, some OpenSHMEM implementations have been moving towards using the newly developed Unified Communication X (UCX) [5] high-performance network middleware layer, which is described in more detail in section III-A. Other networking middleware layers and libraries can be compared and contrasted with UCX in a few different categories. Established projects like GASNet [4] and ARMCI [17] tend to be more specialized for uses like global address space and distributed array models. CCI [18] is a generic interface aiming to provide a socket-like replacement for RDMA interconnects. Vendor-supported examples include Intel's PSM [19], Cray's uGNI and DMAPP [20] interfaces, but these are hardware and platform-specific. IBM's PAMI[21] abstraction targets OpenPOWER architectures while supporting multiple interconnect technologies. Another high-level abstraction is Portals [12], being focused on researching network architecture building blocks rather than optimizations for low-level hardware support. Most recently, the Portals specification was embraced by BXI [22], which realizes the Portal's architecture in hardware. In addition, there are several low-level interfaces to various specific network hardware technologies. These include Verbs [23] and libfabrics [13] libaries developed by Openfabrics Alliance, and the previously mentioned UCCS. While both UCX and libfabrics provide support for multiple network technologies, in this work we focus on enabling support for new platforms within the emerging UCX framework. UCX was appropriate for this task because it provides helpful features such as native support for the Mellanox MLX5 architecture and a choice between using a high-level API for abstraction and ease of use, or access to the lower-level API for performance optimizations. In UCX, these APIs are unified under a common implementation and apply to the variety of platforms mentioned above. The implementation is based on the concept of Modular Component Architecture (MCA) [24], which simplifies the framework portability by leveraging already existing components.

Besides internode network communication services, there are also several alternatives for intranode shared-memory access. Cross-partition memory (XPMEM) is a shared-memory technology that was originally developed by SGI and later ported to Cray [25], and now there is experimental support for newer kernels (>=3.12) maintained by Los Alamos National Laboratory [26]. It is a kernel module and user level library that supports memory registration and cross-process mapping of user-allocated memory. It has been used in high-performance communications implementations such as the "vader" transport in Open MPI [27]. Related kernel-level shared-memory technologies that are also supported in UCX but not evaluated in this paper include the linux Cross-Memory Attach (CMA) [28] and Kernel Nemesis (KNEM) [29]. Besides these customized kernel module implementations, native Linux shared-memory access can also be achieved through the natively-supported SysV and Posix APIs. Finally, LiMIC [30] is a specialized shared-memory mechanism used in the MVAPICH runtime to accelerate shared memory operations.

III. Design

In this section we provide an overview of the software stack used for the evaluation report in this paper, as well as ARM specific enhancements for the relevant software components. We will briefly describe UCX and XPMEM, and how they were integrated into OSHMEM.

A. UCX API

UCX is a collaboration between industry, national labs and academia that consolidates technologies from Mellanox MXM [31], ORNL's UCCS [6], [7] and IBM Parallel Active Message Interface (PAMI) into a unified open source framework. It is cross-platform and supports network technologies like InfiniBand, Cray Gemini/Aries, and shared memory architectures for x86-64, POWER, ARM, and GPUs. The UCX API exposes both high- and low-level interfaces to satisfy both better accessibility for exploratory programming model implementation and access to hardware optimizations for performance tuning. To satisfy this design goal, UCX is actually a unification of three separate APIs: UCP for high-level usability, UCT for low-level optimizations, and UCS as a utility glue layer.

- UCT (for "transports") exposes basic network operations supported by networking and shared-memory hardware. It provides reliable and out of order delivery for various

protocols with its main functionality consisting of the setup and instantiation of communication operations.
- UCP (for "protocols") uses UCT to construct protocols commonly found in applications. It provides several communication paradigms such as multi-rail services, device selection protocols, pending queue implementation, tag-matching, and software atomics.
- UCS (for "services") is the infrastructure for component based programming, memory management, and system utilities when using UCT and UCP. This infrastructure includes various platform abstractions, data structures, and debugging facilities.

The ARM enablement work in UCX primarily focused on the UCS and UCT layers. The UCS layer was extended to support native ARM instructions for various memory and instruction barriers, access to the high-resolution clock, atomic operations, and optimized bit manipulation operations. In the UCT layer, we added a few enhancements related to the cache-line alignment and false sharing avoidance in the shared memory transports.

In terms of InfiniBand interconnects, UCX supports two major types of communication conduits: Verbs and MLX5. The Verbs conduit follows the traditional path for RDMA interconnects and leverages the Verbs user-level driver for triggering communication directives. The MLX5 conduit uses the highly optimized UCX built-in driver for the Mellanox InfiniBand ConnectX-4, 5, and 6 architecture. The primary goal of the MLX5 conduit is to optimize application communication paths for latency and message-rate-sensitive operations. The MLX5 conduit was extended to use ARM NEON SIMD extensions (`vqtbl1q_u8` intrinsics) for the following operations: InfiniBand inline memory copy, RDMA work request initialization, and InfiniBand control segment initialization. The shared memory communication across processes sharing the same memory space is realized using the XPMEM conduit, which provides high-performance directives for cross-process memory access, including atomic operations. As a result, we were able to enable efficient InfiniBand and shared-memory support for ARM v8 platforms.

All the above enchancements for UCX are open source and can be found in the UCX repository [32].

B. XPMEM

Cross-partition memory (XPMEM) [26], [25] is a non-standard Linux kernel module providing shared-memory services similarly to Linux CMA or `mmap()` functionality. These services allow an active process to interact with the virtual memory address space of a separate active process using read, write, or combined read/write access. This happens via exposed API calls from XPMEM, where a process must first export a region of its own virtual address space at which point other processes can attach to it. Once the remote memory is mapped into a process's virtual address space, it is used via direct loads and stores, allowing higher-level programming models to execute single-copy intranode communications.

However, there are some semantic differences between XPMEM and other shared memory approaches such as `mmap()`. XPMEM provides the ability to pre-export as-yet unallocated regions of the virtual address space. In such a case, valid addresses (allocated) are mapped between processes on demand, and invalid addresses result in a `SIGSEGV` in the same way as unallocated memory local to the process. In addition, XPMEM does not enforce any particular memory allocation methodology, and therefore it can be used for sharing global, static, heap, and stack memory regions. This capability is especially useful for OpenSHMEM programming that exposes communication semantics enabling remote memory access to global and static memories. Finally, there are two primary differences between XPMEM and other memory sharing approaches such as Linux CMA and KNEM. First, both CMA and KNEM invoke system calls in their communication paths, while XPMEM avoids expensive system overheads associated with the system calls and directly leverages load-and-store semantics. Second, since XPMEM maps the memory across the processes, the memory can be accessed using native (in terms of the underlying architecture) atomic directives. Contrary to this, CMA and KNEM do not provide any support for atomic operations, which is an important class of operations in the OpenSHMEM programming model.

The open source implementation [26] of the XPMEM kernel module originally ignored Linux defined abstractions and directly referenced x86_64 architecture-specific code, which is related to the memory subsystem and huge-pages implementations in the Linux kernel. As part of this work, we addressed this constraint and implementation relevant modifications to the XPMEM kernel module for the ARM v8 architecture. Since these changes are mostly related to the kernel-module implementation mechanics, the details of implementation are out of scope of this paper and can be found in the open source version of XPMEM [26].

C. Open MPI SHMEM Communication Layer

Open MPI [9] is a popular open source implementation of the MPI specification. The Open MPI software architecture takes its roots in the concept of Modular Component Architectures (MCA) [33], which provides a very efficient framework for software capability extensions. Using this framework, the Open MPI community extended the project to support the OpenSHMEM specification. The OpenSHMEM implementation within Open MPI is called OSHMEM. It is important to note that OSHMEM is not implemented on top of the MPI communication semantics but directly accesses the hardware through the intermediate software layer called SHMEM Point-to-Point Management Layer (SPML). For integration with UCX, OSHMEM implements UCX SPML that maps OpenSHMEM directives to the UCX API. Figure 1 summarizes this OpenSHMEM stack for ARM. OSHMEM put, get, and atomic routines are translated to UCP calls through the UCX SPML layer. UCP dispatches the call based on the requested destination, type of memory (global, static, symmetric), and the underlying transport capabilities. In the context of our

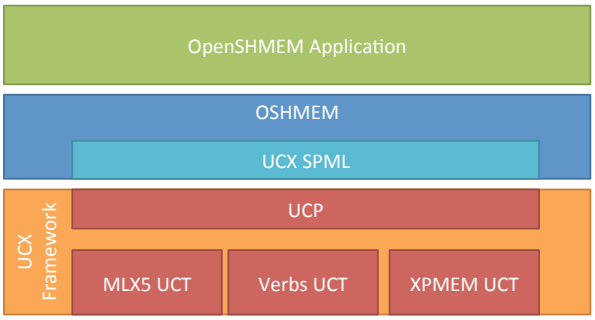

Fig. 1. OpenSHMEM software stack on ARM

work, the request can be dispatched to InfiniBand-MLX5, InfiniBand Verbs reliable connection (RC), or XPMEM UCT. Once UCT receives the message, it forms the request and passes the data to the underlying driver or hardware. In the case of Mellanox MLX5 devices, UCT triggers the PCIe doorbell which pushes the request down to the hardware. For OpenSHMEM collective operations, OSHMEM leverages the MPI collectives interface.

From the beginning, Open MPI was designed to be an architecture-agnostic, platform-independent MPI implementation. OSHMEM inherits this characteristic and therefore it supports the ARM architecture out of the box.

In the context of this work, we decided to focus on potential enhancements of the OSHMEM layer that aim to improve energy efficiency without sacrificing performance. Specifically, we looked at point-to-point synchronization routines in the OpenSHMEM specification. The *shmem_<datatype>_wait()* routines accept two arguments: a pointer to a variable that the user expects to be updated and a compare-value of the same data type. The function blocks until some other processing element changes the value of the pointer to a value that is different from compare-value. *shmem_<datatype>_wait_until()* routines expose similar functionality, but in addition, the user can specify the type of compare operation (equal, not equal, less than, etc). For the rest of the paper we reference the two operations as a general *shmem_wait()*.

Typically, the *shmem_wait()* operations are used in combination with other communication routines. Applications use bulk *put/get* routines for data communication, which are followed by short *put* notification updates. The receiver of the notification uses *shmem_wait()* to block until it receives the notification. OSHMEM implements the blocking functionality as a busy-waiting loop that is polling for the synchronization value to change; other OpenSHMEM implementations use a similar approach for implementing *shmem_wait()*. While such an implementation of this functionality is optimal in terms of latency, it generates substantial load on the CPU. Nevertheless, such a busy-wait workload pattern is not unusual for high-performance system software libraries, like those that have been well explored by the ARM software community. To help in this situation, the ARM ISA defines the Wait-for-Event (*WFE*) instruction. *WFE* suspends execution on the processor (clock is stopped) and puts the processor in a low-power mode. The processor resumes its execution if one of the following events occurs: interrupt, send event, or memory update for an address that is marked as exclusive. As a result, the *WFE* instruction is a perfect fit for an energy efficient implementation of *shmem_wait()*. In our implementation, we mark the synchronization variable as an exclusive and insert the *WFE* into the busy-wait loop. The busy-loop checks the status of the variable and executes the *WFE* instruction, which pauses the clock and puts the CPU in an energy saving mode. On any memory update of the variable, the *WFE* instruction resumes normal operation and checks the status of the variable. This is the first implementation of an OpenSHMEM library known to us that explicitly leverages the this capability of the ARM ISA to improve efficiency of the library.

IV. Results

In this section, we evaluate the performance of OSHMEM and UCX on a testbed ARM v8 based system with an InfiniBand network interconnect. This platform consists of two Softiron Overdrive 3000 servers, and each one is powered by an 8-core AMD Opteron A1100 series processor with 16 GB system memory and Mellanox EDR (100 Gb/s signaling rate) ConnectX-4 IB/VPI host channel adapter (HCA). The HCA is connected to PCIe Gen2 x8 system bus, and so the theoretical bandwidth is constrained to 32 Gb/s speed. The practical bandwidth that takes into consideration the PCIe encoding, header, and protocol overheads can be estimated around 27Gb/s [34]. The machines were running Ubuntu 16.04 (kernel 4.4.0-24), a pre-production version of the Mellanox OFED 3.3-1.5.0.0, XPMEM master version (bdfcc52), UCX master version (0558b41), and Open MPI master version (fed4849).

For the basic performance evaluation of UCP `put`, `get`, and `atomic` operations we used the capabilities of `ucx_perftest` which is distributed as part of the UCX library. These UCP routines underpin the implementation of corresponding OpenSHMEM operations and are therefore important for our study. In `ucx_perftest`, the UCP `put` latency is measured as the total round-trip ("ping-pong") time, divided by two. The UCP `get` latency, `put` and `get` bandwidth, and the time for the atomic memory operations (AMOs) are measured by averaging the total time to complete the corresponding UCP operation over many trials divided by the number of trials. The bandwidth characteristic is measured using the same micro-benchmark and it is calculated as $\frac{request\_size * trials}{total\_time}$.

First, we look at the UCP InfiniBand results which are summarized in figure 2. Panels 'a,' 'b,' and 'c' show the raw performance of the latency, message rate, and bandwidth for various message sizes using both the MLX5 reliable connected (RC) and the Verbs RC transports implemented in the UCT layer. For the rest of the paper we will reference these two transports as MLX5 and Verbs. Panel 'd' of figure 2 compares

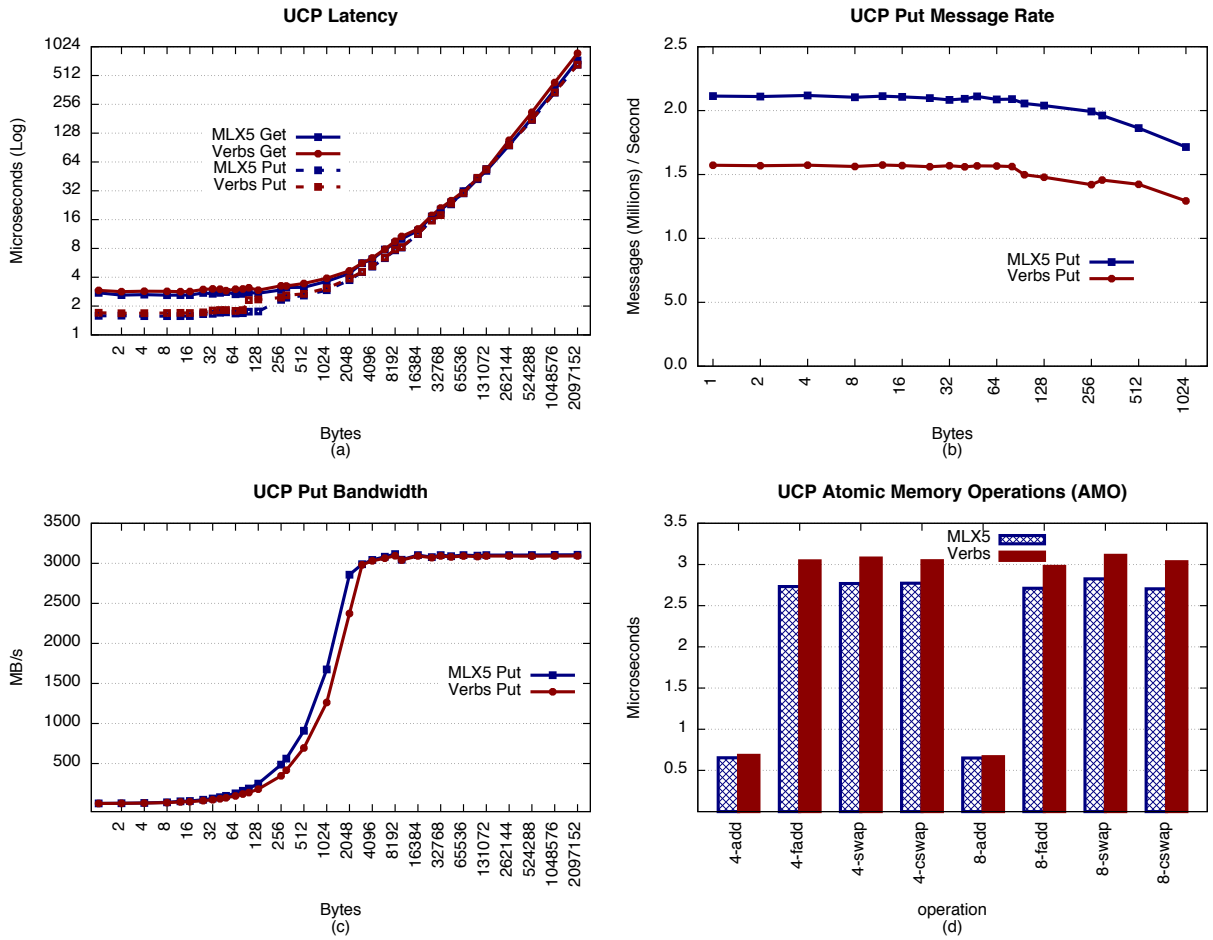

Fig. 2. Summary of InfiniBand results for UCP on ARM v8 with InfiniBand. MLX5 demonstrates better `put` message rates as seen in panel 'b.' MLX5 slightly outperforms Verbs in `put` latency and bandwidth (panels 'a' and 'c'), and atomic operations' performance shown in panel 'd.'

the performance of MLX5 and Verbs for several 4-byte and 8-byte OpenSHMEM atomic memory operations (AMOs). While MLX5 generally outperforms Verbs in all benchmarks, the message injection rate in this case shows a significant difference. The MLX5 transport is implemented natively in UCX on top of the ConnectX-4 hardware abstraction layer, and this allows UCX to reduce the number instructions in the communication path as well eliminate some of the memory barriers. As a result, the MLX5 transport demonstrates a 1.4x improvement in the injection rate. The hardware as described at the beginning of section IV is estimated to support 27Gb/s, and with a single core (2Ghz) we were able to achieve 24.2Gb/s, which is 90% of the practical limit. The substantial difference in the latency of the AMO `add` operation can be attributed to the fact that this is the only non-fetching atomic operation in the group. It can therefore be completed immediately as it is posted, while the rest of AMOs have to wait until the data from remote peer arrives.

In figure 3 we show a summary of the UCP performance when using the XPMEM shared memory transport implemented in UCT and measured using the `ucx_perftest` benchmark. In general, the performance for the latency, message rate, and bandwidth of the UCP `put` and `get` operations are fairly symmetric as expected – especially at larger message sizes. Nevertheless, we observe a substantial difference in the latency of `put` and `get` operations for small message sizes. In the UCT layer both operations are realized as a simple `memcpy()` call, and therefore we expected to see nearly identical performance. This difference is likely due to the different communication patterns (ping-pong vs ping-ping) which are used for measuring `put` and `get` latencies.

In addition, we developed a simple OpenSHMEM `put-and-wait` ping-pong benchmark for measuring `shmem_wait()` efficiency. One Processing Element (PE) initiates `shmem_longlong_put()` and then calls a blocking `shmem_wait()`, which waits for the remote side to respond. Another PE executes the same communication directives in the opposite order. First it calls `shmem_wait()` and then responds with `shmem_longlong_put()`. The number of executed cycles and instructions are measured using the Linux `perf` utility. The `shmem_longlong_put()` latency is measured as a half-round trip latency of the ping-pong communication pattern.

As described in section III-C, we were able to use the

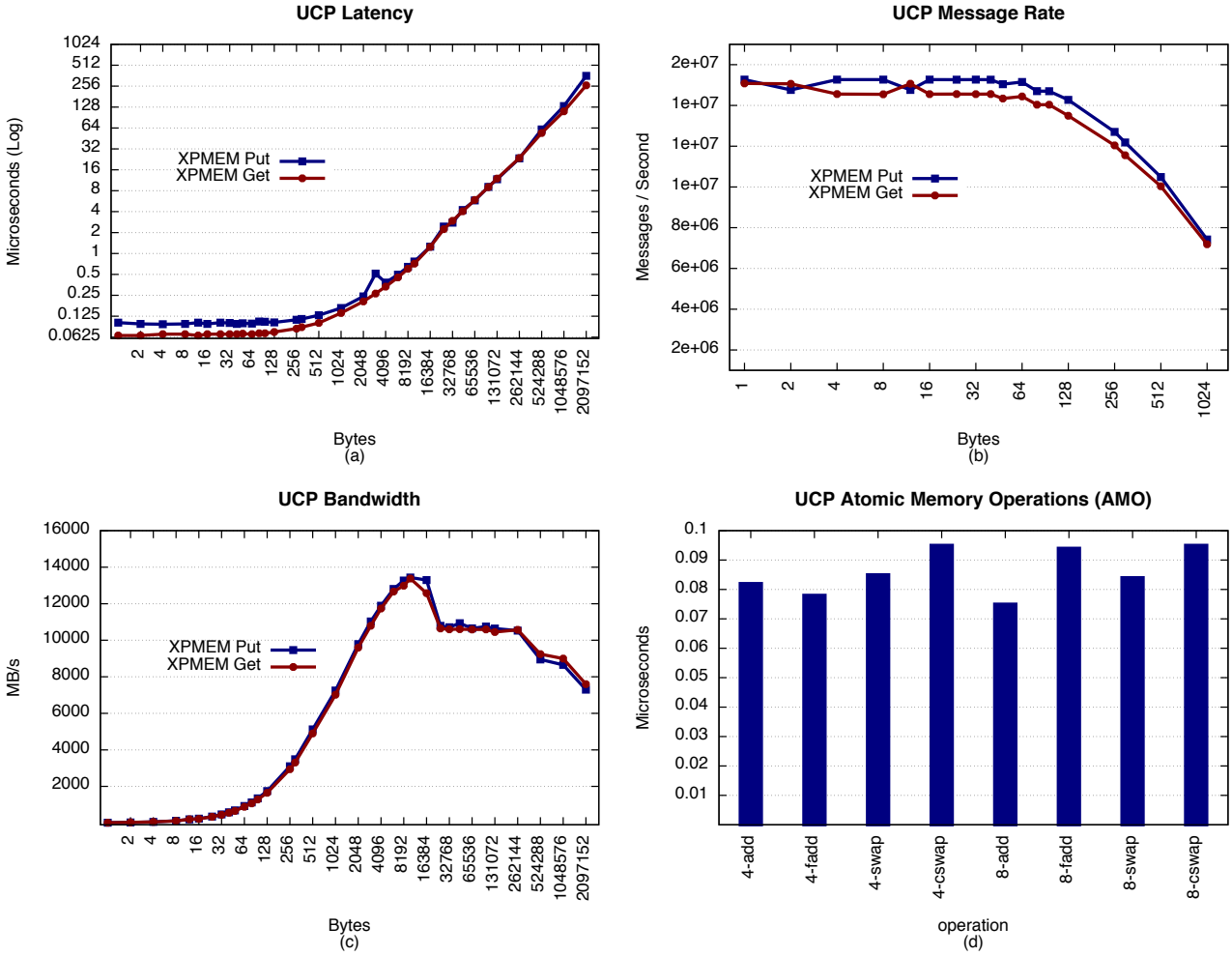

Fig. 3. Summary of the XPMEM results for UCP on ARM v8. As expected, the performance for UCP `put` and `get` operations are essentially symmetric for the UCP latency, message injection rate, and bandwidth as shown in panels 'a,' 'b,' and 'c,' respectively. The measured results for common UCP atomic memory operations are shown in panel 'd.'

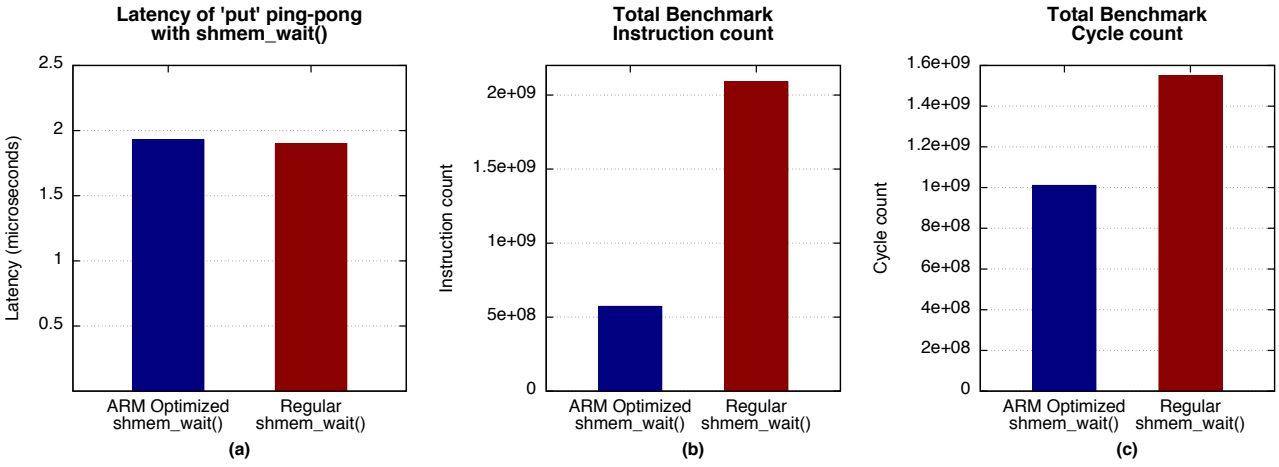

Fig. 4. The performance and efficiency of the `shmem_wait()` operations when using the ARM ISA Wait-for-Event instruction compared with the typical busy-wait loop implementation. Panel 'a' indicates that the performance of the payload-carrying `put` operations are not adversely affected, while panels 'b' and 'c' indicate increased efficiency of 73% for the instruction count and 35% for the cycle count, which should lead to greater power efficiency of the CPU as well.

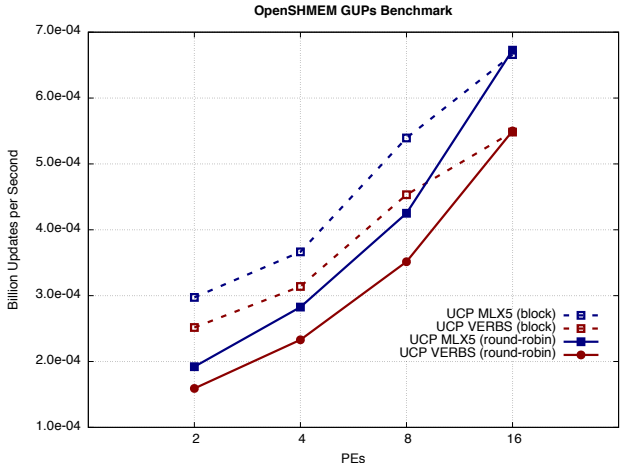

Fig. 5. Using OSHMEM with the GUPs benchmark. The optimized MLX5 transport steadily outperforms the Verbs transport on the ConnectX-4 adapter by about 21% depending on the number of processing elements. The block distribution outperforms the round-robin due to the availability of the XPMEM transport for the 8 processing elements on a node.

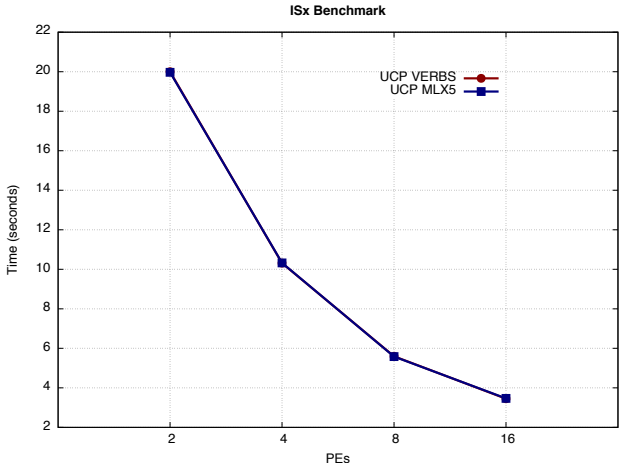

Fig. 7. The ISx benchmark is evaluated using the Verbs and MLX5 transports in UCP with XPMEM enabled for handling shared-memory intra-node transfers. The communication to computation load in this application is lighter than the others, so the effect of the variously-optimized transports is minimal.

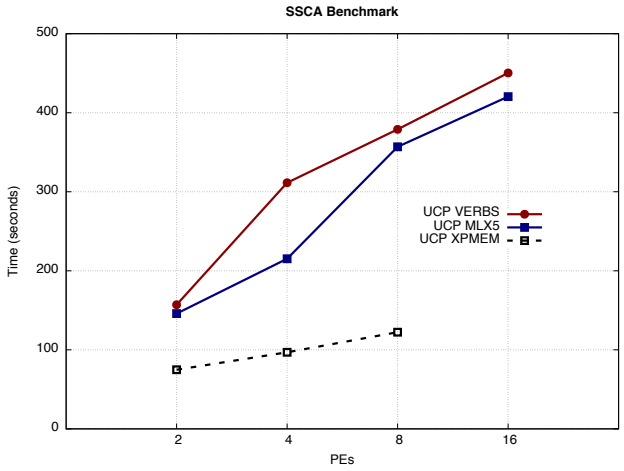

Fig. 6. The HPCS SSCA benchmark is evaluated using the Verbs and MLX5 transports in UCP with XPMEM enabled for handling shared-memory intra-node transfers. For this experiment, we show the results for the MLX5 and Verbs when used with processing elements placed in a round-robin configuration. As shown by the dashed line, the XPMEM transport significantly outperforms when PEs are placed in a block configuration on the same node, and would otherwise dominate the behavior of the MLX5 and Verbs results.

ARM Wait-for-Event (`WFE`) instruction to increase the efficiency of some OSHMEM point-to-point synchronization operations. Figure 4 shows the effect of using this optimization in the OpenSHMEM `shmem_wait()` operation. Overall, the benchmark demonstrates that the ping-pong latency of `shmem_longlong_put()` remains unaffected (panel 'a') even though the thread is no longer actively polling for progress. We also see increased efficiency by looking at the total instruction and cycle counts for the `WFE`-optimized and unoptimized `shmem_wait()` implementations, shown in panels 'b' and 'c,' respectively. Using the `WFE` version of the `shmem_wait()` routine, we observe 73% reduction in the number of the executed instructions and 35% reduction in the number of cycles. This observed increase in efficiency indicates the potential for lower power consumption of these operations as well, but with the preliminary status of our testing platform the measurement tooling is not yet able to provide detailed enough information to directly measure the isolated power consumption of specific operations.

To show more general application-oriented results for the OSHMEM implementation on the ARM v8 InfiniBand platform, we used an OpenSHMEM version of the GUPs benchmarks, which is based on the MPI version of the benchmark distributed as part of the High Performance Computing Challenge (HPCC) [35] benchmark suite. The benchmark measures system performance in terms of Giga Updates per Second (GUPs), which are defined by the number of random memory locations that can be updated in one second. This provides an idea of peak performance of the system under random memory access workloads. More specifically, this benchmark measures how a value is retrieved through a `get`, a new value stored with a `put`, and another value incremented with an atomic operation at a random memory location. We show results for the GUPs benchmark in figure 5. As can be seen in the figure, the MLX5 transport provides a 21% increase in performance over Verbs for 2–16 OpenSHMEM processing elements (PEs).

For this experiment, we use two different PE placement strategies to show the possible differences in performance of the transports that get selected. In a block distribution of PEs, all PEs in the experiments using 2, 4, or 8 PEs are placed on the same machine, and when 16 PEs are used they are distributed across two machines. In the block distribution scheme, only the XPMEM transport is used for communication among PEs in the experiments with 2, 4, and 8 PEs, and as a result the block distribution outperforms the round robin distribution where MLX5 or Verbs are used. It

is important to note that there is variance in XPMEM-MLX5 and XPMEM-Verbs results due to the fact that the Verbs or MLX5 progress function is invoked, even so xpmem is used. For the experiments using 16 PEs, MLX5 outperforms Verbs. This result is not surprising since the benchmark generates multiple put requests, and previous UCP evaluations [8] have shown that MLX5 benefits from a higher message rate.

Another application-oriented benchmark is HPCS SSCA. HPCS SSCA is an implementation of Smith-Waterman local sequence alignment algorithm [36][37][38]. The communication part of the algorithm consists of an outer and inner loop. The inner loop issues many shmem `get` operations and few shmem `put` operations for small data sizes. As a result, the benchmark is sensitive to small message latency and injection rates. The performance metric for the benchmark is defined as a time reported by `kernel1` of HPCS SSCA, which is overall execution time of the kernel including communication and computational parts. Figure 6 summarizes performance results for the benchmark, which shows between a 7–30% improvement when using MLX5 over Verbs corresponding to previous observations of MLX5 `put` operations being faster [8]. When PEs are distributed in a block placement scheme as described above, the XPMEM shared memory results significantly outperform within a single node effectively nullifying the differences between MLX5 and Verbs for this case.

ISx [39][40] benchmark is another representative of application focused benchmark from OpenSHMEM community. The benchmark is a scalable variant of the NAS IS benchmark [41] that implements distributed integer sort algorithm. ISx consists of two main parts - local bucket sort algorithm and all-to-all communication patter that is used for the exchange of keys. Figure 7 presents performance results for the benchmark. For the evaluation we used `isx.strong`, which evaluates strong scaling properties of the system and the algorithm. The performance for the benchmarks is measured and an average execution time of the ISx across all processing elements. From the performance results we can see that OpenSHMEM using MLX5 and Verbs UCT transport layers demonstrate nearly identical execution times. This results aligns well with our expectation; on a small scale system the local sort algorithm dominates overall execution time while the pressure on communication resources is relatively modest.

V. Conclusion and Future Work

This paper documents a step towards enabling ARM v8 architecture support for one-sided communications, using the OpenSHMEM programming model and the InfiniBand interconnect as a demonstrating implementation. In the context of this work, we ported the UCX framework and XPMEM kernel module to the ARM v8 architecture and investigated potential opportunities for ARM-specific optimizations in the OpenSHMEM library. It is important to note that this study is based on pre-production level hardware and software components, and therefore focused on the enablement of the software stack and identification of potential opportunities for further optimization. We demonstrated that using a single AMD Opteron A1100 core the UCP layer reaches 90% of the potential bandwidth while leveraging the underlying capabilities for the RDMA network. In addition, we present an optimization for the `shmem_wait()` operation, which increased the efficiency of the operation in terms of instruction and cycle count by 73% and 35% respectively, without compromising the latency of the OpenSHMEM communication directives. Using the SSCA, ISx, and GUPs OpenSHMEM application-focused benchmarks we demonstrated that the ARM v8 architecture also benefits from the custom developed MLX5 UCT driver and newly enabled XPMEM shared memory transport in UCT. To the best of our knowledge, this is first work demonstrating InfiniBand, OpenSHMEM, UCX, and XPMEM functionality together on ARM. In future work, we plan to focus on further opportunities for energy efficiency and performance optimizations for the ARM architecture, as well as extending these optimizations to MPI one-sided and other communication semantics. With the availability of larger-scale ARM platform installations, and the extensions presented here, further studies to identify opportunities for optimizations specific to scaling up will be of primary interest.

Acknowledgements

We would like to acknowledge UCX developers for their guidance in enabling ARM v8 architecture support in UCX. Also, we would like to thank Nathan T. Hjelm from Los Alamos National Laboratory for his help in debugging the XPMEM kernel module and Alexander Mikheev for his work on UCX and OSHMEM integration.

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

References

[1] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith, "Introducing OpenSHMEM: SHMEM for the PGAS community," in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, ser. PGAS '10, New York, NY, USA, 2010. [Online]. Available: http://doi.acm.org/10.1145/2020373.2020375

[2] S. W. Poole, O. Hernandez, J. A. Kuehn, G. M. Shipman, A. Curtis, and K. Feind, "OpenSHMEM - Toward a Unified RMA Model," in *Encyclopedia of Parallel Computing*, 2011, pp. 1379–1391.

[3] S. S. Pophale, "SRC: OpenSHMEM library development." in *ICS*, D. K. Lowenthal, B. R. de Supinski, and S. A. McKee, Eds. ACM, 2011, p. 374. [Online]. Available: http://dblp.uni-trier.de/db/conf/ics/ics2011.html#Pophale11

[4] D. Bonachea, "GASNet Specification, v1.1," Berkeley, CA, USA, Tech. Rep., 2002.

[5] P. Shamis, M. G. Venkata, S. Poole, A. Welch, and T. Curtis, *OpenSHMEM and Related Technologies. Experiences, Implementations, and Tools: First Workshop, OpenSHMEM 2014, Annapolis, MD, USA, March 4-6, 2014. Proceedings*. Cham: Springer International Publishing, 2014, ch. Designing a High Performance OpenSHMEM Implementation Using Universal Common Communication Substrate as a Communication Middleware, pp. 1–13. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-05215-1_1

[6] P. Shamis, M. G. Venkata, J. A. Kuehn, S. W. Poole, and R. L. Graham, "Universal Common Communication Substrate (UCCS) Specification. Version 0.1," Oak Ridge National Laboratory (ORNL), Tech Report ORNL/TM-2012/339, 2012.

[7] R. L. Graham, P. Shamis, J. A. Kuehn, and S. W. Poole, "Communication Middleware Overview," Oak Ridge National Laboratory (ORNL), Tech Report ORNL/TM-2012/120, 2012.

[8] P. Shamis, M. G. Venkata, M. G. Lopez, M. B. Baker, O. Hernandez, Y. Itigin, M. Dubman, G. Shainer, R. L. Graham, L. Liss, Y. Shahar, S. Potluri, D. Rossetti, D. Becker, D. Poole, C. Lamb, S. Kumar, C. Stunkel, G. Bosilca, and A. Bouteiller, "Ucx: An open source framework for hpc network apis and beyond," in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, Aug 2015, pp. 40–43.

[9] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004, pp. 97–104.

[10] openshmem.org, "Reference Implementation of the evolving OpenSHMEM specification based on GASNet http://www.openshmem.org/," https://github.com/openshmem-org/openshmem, 2016.

[11] J. Jose, K. Kandalla, M. Luo, and D. K. Panda, "Supporting Hybrid MPI and OpenSHMEM over InfiniBand: Design and Performance Evaluation," in *Proceedings of the 2012 41st International Conference on Parallel Processing*, ser. ICPP '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 219–228. [Online]. Available: http://dx.doi.org/10.1109/ICPP.2012.55

[12] R. Brightwell, T. Hudson, K. Pedretti, R. Riesen, and K. D. Underwood, "Portals 3.3 on the Sandia/Cray Red Storm System."

[13] P. Grun, S. Hefty, S. Sur, D. Goodell, R. D. Russell, H. Pritchard, and J. M. Squyres, "A brief introduction to the openfabrics interfaces - a new network api for maximizing high performance application efficiency," in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, Aug 2015, pp. 34–39.

[14] C. Yoon, V. Aggarwal, V. Hajare, A. D. George, and M. B. III, "GSHMEM: A Portable Library for Lightweight, Shared-Memory, Parallel Programming," in *Partitioned Global Address Space*, Galveston, Texas, 2011.

[15] B. C. ho Lam, A. D. George, and H. Lam, "TSHMEM: Shared-Memory Parallel Computing on Tilera Many-Core Processors," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, 2013, pp. 325–334, http://www.odysci.com/article/1010113019802138. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/IPDPSW.2013.154

[16] Mellanox Technologies LTD, "Mellanox ScalableSHMEM: Support the OpenSHMEM Parallel Programming Language over InfiniBand," http://www.mellanox.com/related-docs/prod_software/PB_ScalableSHMEM.pdf, Sunnyvale, CA, USA, 2012.

[17] J. Nieplocha and B. Carpenter, "Armci: A portable remote memory copy libray for ditributed array libraries and compiler run-time systems," in *Proceedings of the 11 IPPS/SPDP'99 Workshops*, UK, 1999.

[18] S. Atchley, D. Dillow, G. Shipman, P. Geoffray, J. M. Squyres, G. Bosilca, and R. Minnich, "The common communication interface (cci)," *HOTI*, 2011.

[19] Qlogic. Qlogic PSM. [Online]. Available: http://qlogic.com/pages/default.aspx/

[20] Cray Inc., "Using the gni and dmapp apis," in *Cray Software Document*, vol. S-2446-4003, Dec. 2011. [Online]. Available: http://docs.cray.com/books/S-2446-4003/S-2446-4003.pdf

[21] S. Kumar, A. R. Mamidala, D. A. Faraj, B. Smith, M. Blocksome, B. Cernohous, D. Miller, J. Parker, J. Ratterman, P. Heidelberger, D. Chen, and B. Steinmacher-Burrow, "Pami: A parallel active message interface for the blue gene/q supercomputer," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, May 2012, pp. 763–773.

[22] S. Derradji, T. Palfer-Sollier, J. P. Panziera, A. Poudes, and F. W. Atos, "The bxi interconnect architecture," in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, Aug 2015, pp. 18–25.

[23] Openfabrics Alliance, "Openfabrics Alliance," https://openfabrics.org/, 2016.

[24] J. M. Squyres and A. Lumsdaine, "The component architecture of open MPI: Enabling third-party collective algorithms," in *Proceedings, 18th ACM International Conference on Supercomputing, Workshop on Component Models and Systems for Grid Applications*, V. Getov and T. Kielmann, Eds. St. Malo, France: Springer, July 2004, pp. 167–185.

[25] "xpmem: Cross-process memory mapping," https://code.google.com/archive/p/xpmem/, 2011.

[26] N. Hjelm, "Linux cross-memory attach," https://github.com/hjelmn/xpmem, 2016.

[27] J. Squyres, "The "vader" shared memory transport in Open MPI: Now featureing3 flavors of zero copy!" http://blogs.cisco.com/performance/the-vader-shared-memory-transport-in-open-mpi-now\-featuring-3-flavors-of-zero-copy, 2014.

[28] J. Vienne, "Benefits of cross memory attach for mpi libraries on hpc clusters," in *Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment*, ser. XSEDE '14. New York, NY, USA: ACM, 2014, pp. 33:1–33:6. [Online]. Available: http://doi.acm.org/10.1145/2616498.2616532

[29] B. Goglin and S. Moreaud, "Knem: A generic and scalable kernel-assisted intra-node MPI communication framework," *Journal of Parallel and Distributed Computing*, vol. 73, no. 2, pp. 176 – 188, 2013. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0743731512002316

[30] H. W. Jin, S. Sur, L. Chai, and D. K. Panda, "Limic: support for high-performance mpi intra-node communication on linux cluster," in *2005 International Conference on Parallel Processing (ICPP'05)*, June 2005, pp. 184–191.

[31] Mellanox Technologies LTD, "Mellanox Messaging(MXM): Message Accelerations over InfiniBand for MPI and PGAS libraries," http://www.mellanox.com/related-docs/prod_software/PB_MXM.pdf, Sunnyvale, CA, USA, 2012.

[32] openucx.org, "Unified Communication-X Framework," https://github.com/openucx/ucx, 2013.

[33] J. M. Squyres and A. Lumsdaine, "The component architecture of open MPI: Enabling third-party collective algorithms," in *Proceedings, 18th ACM International Conference on Supercomputing, Workshop on Component Models and Systems for Grid Applications*, V. Getov and T. Kielmann, Eds. St. Malo, France: Springer, July 2004, pp. 167–185.

[34] J. Lawley, "Understanding Performance of PCI Express Systems," http://www.xilinx.com/support/documentation/white_papers/wp350.pdf, 2014.

[35] P. Luszczek and J. Dongarra, "Analysis of various scalar, vector, and parallel implementations of randomaccess," Innovative Computing Laboratory (ICL) Technical Report, ICL-UT-10-03, June 2010.

[36] D. Bader, K. Madduri, J. Gilbert, V. Shah, J. Kepner, T. Meuse, and A. Krishnamurthy, "Designing scalable synthetic compact applications for benchmarking high productivity computing systems," *Cyberinfrastructure Technology Watch*, vol. 2, pp. 1–10, 2006.

[37] 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.

[38] M. Baker, "HPCS benchmark challenge SSCA1 for OpenSHMEM ," https://gitlab.com/MattBBaker/ssca1, 2016.

[39] U. Hanebutte and J. Hemstad, "Isx: A scalable integer sort for co-design in the exascale era," in *Partitioned Global Address Space Programming Models (PGAS), 2015 9th International Conference on*. IEEE, 2015, pp. 102–104.

[40] ——, "ISx is Scalable Integer Sort Application," https://github.com/ParRes/ISx, 2015.

[41] nasa.gov, "NAS Parallel Benchmarks," https://www.nas.nasa.gov/publications/npb.html, 1994.