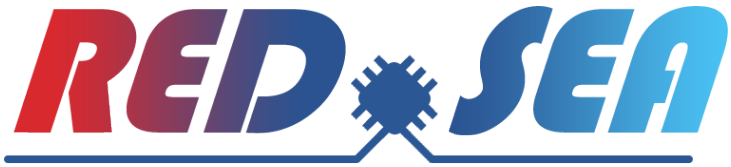
Network Solution for Exascale Architectures



D4.7: Optimized MPI and compute in network implementations

Document Properties

Contact Number	955776
Contractual Deadline	30/11/2023
Dissemination Level	Public
Nature	Report
Edited by:	Simon Pickartz
Authors	Xu Huang, Simon Pickartz, Carsten Clauss, Gilles Moreau, Hugo Taboada, Marc Pérache, Timo Schneider
Reviewers	Nikos Chrysos, Giuseppe Piero Brandino
Date	November 2023
Keywords	BXI, Portals 4, MPI, sPIN
Status	Final
Release	1.0

This project has received funding from the European High-Performance Computing Joint Undertaking (JU) under grant agreement No 955776. The JU receives support from the European Union's Horizon 2020 research and innovation programme and France, Greece, Germany, Spain, Italy, Switzerland.

Contents

List of Figures

List of Tables

Executive Summary

This deliverable presents improvements of the support for the BullSequana eXascale Interconnect (BXI) by the two Message-Passing Interface (MPI) implementations ParaStation MPI and Multi-Processor Computing (MPC). The multi-rail support in MPC was further improved especially with respect to its support for the rendezvous protocol. This defers the transmission of MPI payload until the target application buffer is known to the communication layer to avoid intermediate copies of large memory regions. The *pscom4portals* plugin of the pscom library enabling BXI support in ParaStation MPI has been adapted to the new Remote Memory Access (RMA) interface. This interface has been developed within the DEEP-SEA project and provides upper software layers with a more direct access to the hardware's RMA capabilities. With these adaptations, applications benefit from an improved performance of MPI one-sided communication on top of BXI.

Finally, this deliverable presents streaming Processing in Network (sPIN), a micro-architecture for network accelerators, as well as FPsPINi, constituting its first full-system prototype implementation in hardware. These works demonstrate how package processing tasks, that are commonly performed by the Central Processing Unit (CPU), can be offloaded to a smart Network Interface Card (NIC) to enable a better overlap of communication and computation in parallel applications.

1 Introduction

The MPI standard is a central component of the software stack in High Performance Computing (HPC) systems. It acts as the main interface towards the application layer while providing efficient access to the resources of the HPC interconnect and still abstracting from the low-level hardware details. This way, MPI applications can run on any supercomputing system without the need for system-specific adaptations.

This deliverable builds upon the work presented in Deliverable D 4.3 of the RED-SEA project [29]. This presented the software architecture of the two MPI implementations MPC and ParaStation MPI while focusing on the support for the BXI network. In doing so, the respective extensions developed in RED-SEA were introduced: Commissariat à l'énergie atomique et aux énergies alternatives (CEA) developed multi-rail support for MPC to make efficient use of multiple NICs per compute node and ParTec added support for BXI to ParaStation MPI, enabling efficient communication in HPC system using this high-speed interconnect. This way, also Modular Supercomputer Architecture (MSA) systems exhibiting a heterogeneous network landscape can be supported, as these developments enable the transparent MPI bridging to/from BXI.

This deliverable introduces further optimisations of these two MPI w.r.t. the support for BXI. CEA could further improve the multi-rail support and its integration within the MPC framework. Efforts were distributed into validating the prototype developed in Deliverable D 4.3 in terms of functionalities and performances and introducing new key software designs to strengthen the foundation of the communication module. ParTec adapted the *pscome4portals* plugin enabling BXI support in ParaStation MPI to the re-design of pscom's RMA Application Programming Interface (API). This re-design is part of the results of the DEEP-SEA project with the goal to enable more efficient implementations of MPI one-sided communication on top of pscom by providing direct access to the hardware's RMA capabilities. Additionally, these developments enable the efficient composability of MPI and Partitioned Global Address Space (PGAS) workloads by relying on a common low-level communication stack. By adapting the pscom4portals plugin to this redesign, all the aforementioned benefits are brought to systems leveraging BXI. Additionally, a preliminary analysis of the network bridging support in ParaStation MPI across BXI and InfiniBand (IB) was conducted on the DEEP system at Jülich Supercomputing Centre (JSC).

Finally, this deliverable presents sPIN, which is a micro-architecture for network accelerators developed at ETH Zurich. This micro-architecture is optimised for packet processing, fine-grain memory hierarchies, and data movement acceleration. SmartNICs built upon this architecture can be used for offloading packet processing tasks to the NIC to give the CPU more time for traditional computational tasks.

2 Multi-Processor Computing

2.1 Introduction

Nowadays, HPC applications scale to several hundred nodes, and MPI has become the de facto standard for internode communication. It abstracts P2P communications over many different high performance interconnect networks while taking advantage of their specific capabilities. With ever increasing demand for bandwidth-oriented communications, new architectures have arised featuring multiple NICs. In addition, those NICs are capabable of handling tasks previously devoted to the CPU (DMA communication, tag-matching offloading, triggered operations,...) thus decreasing the communication overhead.

Sending messages over multiple NICs englobes two kind of use cases: first, *data striping* that is to split one message and distribute fragments among the NICs, second, *multiplexing* that is to dynamically schedule messages on the NICs. In this report, we will expose the development that were made in the MPC framework [1], CEA's MPI implementation [30], to support both use cases while taking advantages of the offloading capabilities of the BXI NIC. There are other use cases for multirail such as network failover or redundancy [32, 7] but they are out of the scope of this study.

We decided to describe our solution through three parts. The first part discusses the architectural design choices that were made to support the multirail. Secong, those design constaints were then ported to support BXI networks. And finally, we give a short overview of the support for the multi-thread version of MPC.

2.2 Point-to-point communications with multirail

Most MPI implementation rely on a two-layered architecture for the implementation of P2P communications [16, 36]. To satisfy requirements for fast communication of any kind (size) of data, P2P operations are orchestrated first through the selection of a suitable protocol that is then operated thanks to the transport layer. We describe in the following the different types of protocols, we explain how they were implemented in MPC legacy and how this conflicted with the implementation of the multirail. We finally discuss the new architectural designs chosen to overcome current limitations.

2.2.1 Generalities

Let's first discuss some general notions commonly used in MPI communication libraries.

2.2.1.1 Two-sided and one-sided communications

In the context of MPI, two-sided and one-sided communication refer to different models for processes (typically running on different nodes in a cluster) to exchange data and coordinate their work. These models have distinct characteristics and are used in different scenarios.

Two-sided communication is the traditional form of message passing where communication occurs through explicit send and receive operations. These operations

[1] For more information, visit `https://mpc.hpcframework.com/`

are often blocking, meaning the sender waits until the receiver is ready to receive the message, and the receiver waits until the message arrives. In two-sided communication, processes explicitly synchronize their communication by invoking matching send and receive operations. This synchronicity ensures that the sender and receiver coordinate their actions effectively. MPI provides functions like `MPI_Send` and `MPI_Recv` for two-sided communication. We mention that while we have an explicit synchronization, both primitive have their non-blocking counter-part `MPI_Isend` and `MPI_Irecv`.

One-sided communication is a more relaxed model where one process can directly access and modify the memory of another process without the need for explicit synchronization. This allows for non-blocking communication, where processes can continue their work without waiting for each other. In one-sided communication, processes use PUT and GET operations to access remote memory. For example, a process can initiate a PUT operation to write data directly into the memory of another process. These are commonly called RMA operations.

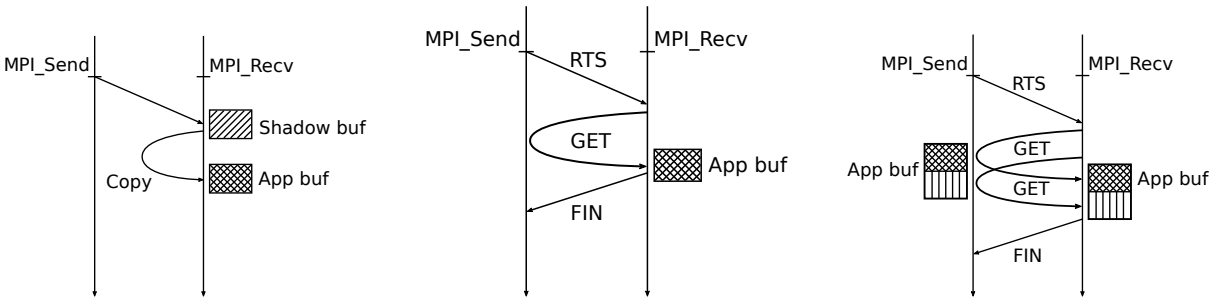
We will see in the following that both models can be used to transfer data efficiently depending on the transfer size and thus using different types of protocols.

2.2.1.2 Protocols

As said earlier, nowdays applications require fast compute and fast network to simulate more and more complex phenomenon and they do so by distributing computation onto hundreds of nodes thanks to communication middlewares implementing the MPI standard. One of the major bottleneck of such middlewares are *copies*. We will focus here on two kind of copies that come from the fact that to send user data from one process to the other, intermediary copies may be needed: first, some are inherent from the underlying protocol (TCP/IP for example) and second, others are due to the non-blocking behavior of MPI communications. The former can be illustrated when an `MPI_Send` call is performed by the sender before and `MPI_Recv` has been posted by the receiver. In order not to block the sender from doing actual computation, the send data is buffered to a temporary buffer, and data will be received and stored on the receiver to an other internal (or shadow) buffer until the user buffer is available.

To avoid the first kind of copies, smart-NICs were devised that are capable of doing *zero-copy* communications. In other words, they are able to read from the send buffer, send it through the network and write it straight to the receive buffer as illutrated in Figure 1b, the technology is also called Remote Direct Memory Access (RDMA) [27]. To avoid the second kind, and in the context of two-sided communications [2], actual data transfert must be preceeded by a synchronization to prepare the memory that will be targeted and exchange information mandatory for the transfert (such as local memory addresses for example). Such synchronization is performed through a *rendez-vous*, one example is given Figure 1b. While this would be suited for bandwidth-oriented communication, message can also be sent *eagerly* to improve latency at the cost of copies, see Figure 1a. It is then the role of the implementation to choose which protocol is best, based on the underlying network and other metrics. [37] give an interesting overview of the different protocols.

[2] We refer the reader to the MPI standard for more information on one-sided and two-sided communications, `https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf`



(a) Eager protocol. (b) Rendez-vous protocol. (c) Data striping protocol.

Figure 1: Illustration of protocols for P2P communications. *Eager*: payload is sent on the first message. *Rendez-vous*: Ready-To-Send (RTS) control message is sent along with tag information for matching, upon successful match, receiver emits a RDMA GET operation using protocol data exchanged with RTS message. Once RDMA operation is completed, receiver sends a FIN control message to notify sender of completion. *Data striping*: rendez-vous multiple RDMA operations.

(a) Data striping (b) Multiplexing

Figure 2: Use cases for multirail.

Finally, we mention protocols related to the multirail feature, and first the *data striping* protocol. Data striping protocol first perform a rendez-vous and then send the data in fragments to the receiver using zero-copy communications with offset management, see Figure 2a. This protocol may be forced whenever the message size exceeds the network Maximum Transfer Unit (MTU). However, it can also be used to improve bandwidth when multiple NICs are available when choosing an adequate fragment size. *Multiplexing* is more a scheduling policy for P2P communication than a protocol and it can also be used to schedule and parallelize communications when multiple NICs are available, see Figure 2b.

2.2.2 Legacy architecture: limitations and drawbacks

As shown in Figure 3, MPC's architecture can be represented by two-layers. The first layer implements all MPI-related constraints specified by the standard such as tag-matching, communicators, or message ordering. The second layer (Rail layer) exposes an API that implements P2P communications over the different type of networks such as TCP/IP (tcp), InfiniBand (ib) or BXI (ptl).

One major implementation design is that protocols described earlier are implemented transparently by the Rail layer. As a consequence, using such API to implement the data striping protocol could imply a synchronization for all fragments whenever their size is larger than the rendez-vous threshold, thus inducing an important performance penalty. Indeed, only one rendez-vous would be necessary to implement this protocol and prepare the memories. The rendez-vous threshold is chosen based on performance tradeoffs between the cost of copies with the eager protocol and the synchronization with rendez-vous. Moreover, since protocols are common to almost all networks (ib, ptl, ofi), it creates code redundancy and thus software maintainability issues.

Another limitation was the design of the Rail API. The API's rigidity becomes a limiting factor, as it enforces stringent rules on how a message is sent and offers little room for customization or adaptability. This hinders the development of tailored protocol management and thus leads to suboptimal performance. Indeed, all message metadata needed for protocol management were included in a single data structure that was then packed as is and sent on the network. While this centralizes informations, this is not efficient, especially in such environments where latency is critical, as an excess of metadata will be sent each time even though different control messages require different metadata.

As a consequence, we decided to make structural changes to overcome this limitation by introducing a dedicated protocol layer. In the following, we discuss the developments in the MPC framework to improve code quality and efficiency to implement the multirail feature.

2.2.3 Recent developments and designs

We discuss here some of the implementation details of the developments realized for the implementation of the multirail feature.

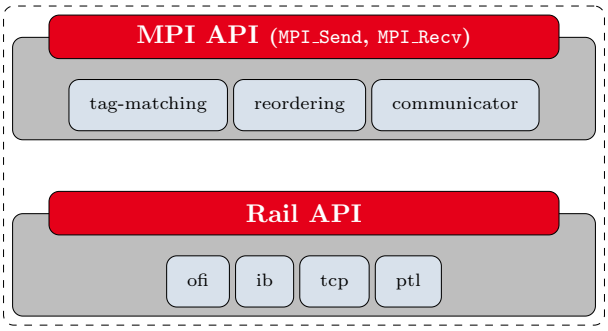


Figure 3: Overview of the previous MPC architecture. In the Rail layer, protocols and data transfert are transparently implemented by the Rail API.

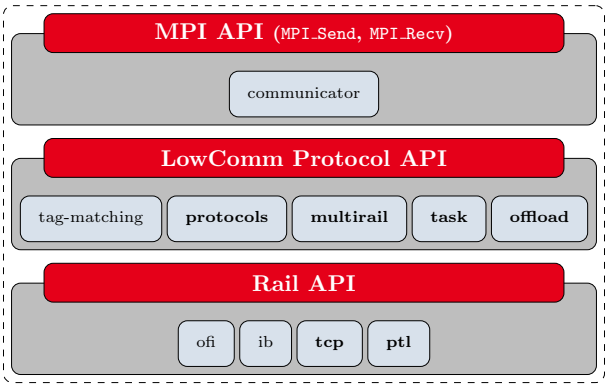


Figure 4: Overview of the new MPC architecture. A dedicated protocol layer is now on top of the Rail layer.

2.2.3.1 Active Message paradigm

The Active Message paradigm for remote communication is a programming model and communication approach that can be used in HPC communication library such as MPI. In this model, communication is asynchronous, decoupling it from computation. It relies on lightweight message passing, where messages contain both control information and data payloads. Each process has predefined message handlers responsible for processing incoming messages. More specifically, messages contain in their header the identifier of a *user-level handler* executed on message arrival and with the message body as argument [13]. The handler must execute quickly and to completion. This approach optimizes protocol processing, making it efficient for control messages and data packets. Customization and extensibility enable developers to define their own message types and handlers for application-specific communication patterns, making it a valuable tool for the development of custom protocols.

For example, consider the development of the rendez-vous protocol such as described in Figure 1b. A specific handler corresponding to the Ready To Send (RTS) control message that will be then responsible for performing the RMA GET operations. On its termination, another control message FIN will be sent which will activate the corresponding handler on the sender so it can complete its request. All necessary handlers are implemented in the protocol layer, see Figure 4. In addition, specific care has been given to use the offloading capabilities of the NIC when available, this

is discussed in more details in Section 2.3. Also, another datapath is reserved for communication between tasks (or threads), see Section 2.4.

To satisfy such requirements and answer to the limitations discussed in Sect. 2.2.2, the Rail API has been extented and new datastructures introduced.

2.2.3.2 Core datastructures and Rail APIs

Our implementation relies on two main datastructures: endpoints and interfaces.

Conceptually, endpoints refer to the communication channels or connection points through which processes can exchange messages and data. They are virtual communication channels associated with each process in an MPI program. They serve as the entry and exit points for communication between processes. Each process has a set of endpoints, and each endpoint is uniquely identified by a specific endpoint number. They are created dynamically which means processes can create and manage endpoints at runtime as needed. This flexibility is useful when the number of processes or communication patterns is not known in advance. We distiguish two types of endpoints: the protocol endpoints and the rail endpoints. The latter are defined in the Rail layer and contains connection information for the specific network that it targets (for example, IP address for TCP, Queue Pairs for InfiniBand or Process Identifiers in Portals4), plus an interface object which are defined in the next paragraph. On the other hand, protocol endpoints are used to englobe the potential multiple rail endpoints. As a consequence, they are the datastructure used to implement the multirail feature. Indeed, internal implementation can choose to iterate (round-robin for example) on the different rail endpoints to send individual messages for multiplexing or fragments in the case of data striping.

Interfaces are designed to abstract the differences between various network technologies, such as InfiniBand, Ethernet, or shared-memory communication. This abstraction allows MPI applications to remain network-agnostic, making it easier to develop portable code that can run on different HPC systems. Interfaces provide a standardized and uniform set of function pointers for key communication tasks. These tasks can include sending data between processes, managing connections and endpoints, and handling various communication-related operations adapted both for two-sided and one-sided communication models. There is basically one interface object per physical NICs.

We will now elaborate on the API comprising the set of operations linked to the Rail. A subset is given in Listing 1. As mentioned in Section 2.2.3.1, two-sided operations provides an argument `id` to specify the identifier of the handler that will be called on the distant process. All API calls provide a sufficiently flexible interface to send custom data making the implementation of any protocol easier and more efficient. Finally, for two-sided operations, we distinguish two types of operations: `bcopy` for Buffered Copy and `zcopy` for Zero Copy. For `bcopy` sends, prior to being sent, user data is copied and packed to a shadow buffer. This allows the user buffer to be reused directly and thus the `MPI_Send` to return immediately and correpond to the eager protocol. For `zcopy` on the other hand, a completion callback is called by the Rail layer whenever the user data has been sent and can thus be reused.

```
//Two-sided operations
ssize_t lcr_send_am_bcopy_func_t(_mpc_lowcomm_endpoint_t *ep,
                                 uint8_t id, lcr_pack_callback_t pack,
                                 void *arg, unsigned flags);

int lcr_send_am_zcopy_func_t(_mpc_lowcomm_endpoint_t *ep,
                             uint8_t id, void *header,
                             unsigned header_length,
                             const struct iovec *iov,
                             size_t iovcnt,
                             unsigned flags,
                             lcr_completion_t *comp);
//One-sided operations
int lcr_put_zcopy_func_t(_mpc_lowcomm_endpoint_t *ep,
                         uint64_t local_addr,
                         uint64_t remote_addr,
                         lcr_memp_t *remote_key,
                         size_t size,
                         lcr_completion_t *ctx);
int lcr_get_zcopy_func_t(_mpc_lowcomm_endpoint_t *ep,
                         uint64_t local_addr,
                         uint64_t remote_addr,
                         lcr_memp_t *remote_key,
                         size_t size,
                         lcr_completion_t *ctx);
```

Listing 1: Two-sided and one-sided operations of the Rail interface

However, this API and especially the Active Message (AM) API falls short when targetting the offloading capabilities of NICs of BXI networks due to the nature of the tag-matching offloading mechanism.

2.3 Application to BXI networks

In this section, we give an overview of the Portals4 standard that is the interface implemented by the BXI to expose the NIC to the application. We then discuss some of the offloading capabilities of BXI NICs, especially the tag-matching offloading, and how we exploited them to support the multirail feature. We then expose some of its limitations and explain why we decided to support the AM API in our Portals4 driver.

2.3.1 Portals4 standard

Portals4 is a communication interface and protocol that focuses on optimizing communication in high-performance and parallel computing environments [4]. It is network-agnostic and it emphasises on low-latency, high-bandwidth communications. It includes support for DMA operations, which enable efficient data transfers without involving the CPU. It also provides messaging capabilities for reliable communication between processes.

Most importantly, it is the standard chosen by BXI to interface with the NIC. Support for BXI network within MPC has been implemented through the Portals4 interface.

2.3.2 Offloaded multirail communications

One of the primary purposes of offloading operations onto the NIC is to improve the global computation to communication overlap by relieving the CPU from some of the communication overhead, such as tag-matching.

2.3.2.1 Tag-matching offloading Tag-matching offloading is a hardware-based feature that allows a NIC to offload certain packet filtering and matching operations from the host CPU to the NIC itself. Instead of handling these operations in software, which can be resource-intensive and introduce latency, the NIC uses dedicated hardware components and algorithms to filter, classify, and process network packets based on specific tags or metadata associated with each packet. The NIC examines the tags in incoming packets and compares them to pre-configured rules or policies.

In the context of an MPI two-sided communication, the primary benefit of tag-matching offloading typically lies in optimizing receive operations. Receive-side offloading reduces the CPU overhead associated with message processing, leading to lower latency, improved scalability, and better performance. In the case where a `MPI_Recv` has been previously posted (before remote data is received) with the specific tag information, the NIC compares the tag and metadata in the message header and if a match is found, it may directly deliver the message to the appropriate memory location. As a consequence, this avoids interruption of the host CPU to handle the incoming message and participate to a better computation/communication overlap. This mechanism is complementary with zero-copies where the tag acts as memory key to access the registered memory created by the `MPI_Recv`. Once the data transfer is complete, the NIC can notify the application that the data has been received and is available for processing. This mechanism may avoid temporary copies during the eager protocol whenever memory registration done during the `MPI_Recv` is performed before the send operation.

However, such communication model based on tag-matching is not suitable with the AM model exposed in Section 2.2.3.1. For other transports such as TCP or InfiniBand implemented in MPC, tag-matching is performed by the software with the classical implementation that uses the combination of the Posted Receive Queue (PRQ) and Unexpected Message Queue (UMQ), see [35] for more information. But this becomes unnecessary when using a tag-matching interface since it is done by the hardware. As a consequence, we developed a new data path in the protocol layer adapted to offload the matching and we extended the Rail API to expose specifically tag-matching, see Listing 2 and the so-called TAG API. The new datapath has been developed in the protocol layer and realizes all protocols (eager, rendez-vous, data striping). We mention that prior to this work, MPC already supported a version of the Portals4 driver.

```
//Tag send operation
int lcr_send_tag_zcopy_func_t(_mpc_lowcomm_endpoint_t *ep,
                              lcr_tag_t tag,
                              uint64_t imm,
                              const struct iovec *iov,
                              size_t iovcnt,
                              unsigned flags,
                              lcr_completion_t *ctx);

//Tag recv operation, or memory registration based on tag
int lcr_recv_tag_zcopy_func_t(sctk_rail_info_t *rail,
                              lcr_tag_t tag,
                              lcr_tag_t ign_tag,
                              const struct iovec *iov,
                              size_t iovcnt,
                              unsigned flags,
                              lcr_tag_context_t *ctx);
```

Listing 2: Tag-matching operations of the Rail interface

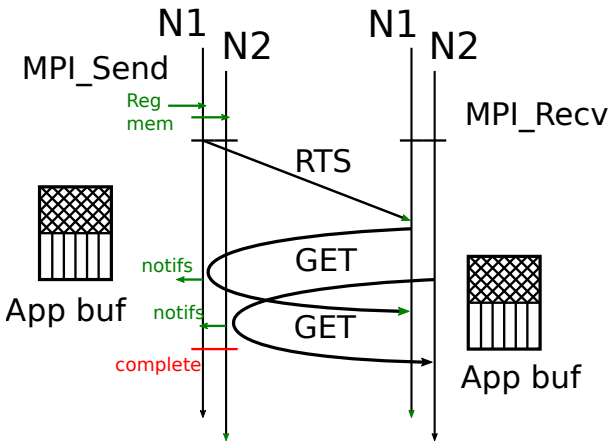


Figure 5: Overview of the datastriping protocol for multirail.

2.3.2.2 Offloaded data-striping

One of the main differences compared to classic data striping protocol is the notification mechanism on registered memory proposed by the Portals4 specification.

Whenever an operation is performed on a registered memory that has been correctly configured, the NIC generates a notification to the application. These are called "notified RMA" operations and they can be used to suppress the need for the FIN control message. The algorithm resulting from this functionality is depicted in Figure 5. In the context of communication on multiple rails, the sender buffer is registered on all available NICs so that the succession of GET operations can be performed by the receiver on each of the fragments through offset management. Each notification produced on the sender are then used to progress and complete the request. Some technical details such as tag management have voluntarily been omitted here.

There are however inherent limitations of the tag-matching offloading mechanism, especially in the context of MPC. We discuss some of them in the next section.

2.3.2.3 Limitations and Active Message support

There are three main limitations encountered with tag-matching offloading that justified the support for AM model by the Portals4 driver in MPC.

First, there is no easy way to implement multiplexing. Indeed, for a message to be received, memory needs to be registered to a specific NIC, however the receiver has no trivial way of knowing on which NIC the next message will be received. And the problem is clear especially for wildcard messages (`MPI_ANY_SOURCE`). As a consequence, multiplexing is still not supported within MPC whenever the tag-matching offloading capabilities is turned on by the driver implementation.

Second, the metadata that can be piggy-backed while sending user data is limited. The Portals4 specification only allows the matching bits and 8 bytes header to be sent as metadata which may hinder the development of more advanced protocols.

Finally, using the thread-based feature of the MPC framework is not possible. Indeed, suppose two MPI threads P0 and P1 post a wildcard (`MPI_ANY_SOURCE`) to one NIC, and a distant MPI P3 sends to P0. Since the tag list of P0 and P1 are shared through the NIC, there are no way to discriminate both receives and thus the message from P3 can not be unambigously delivered to P0.

For all these reasons, it was decided to support the AM model on our Portals4 driver. The support for AM API consisted in implementing the respective calls as presented in Listing 1 while preserving the possibility the use the TAG API at runtime. Overall, our main contributions was the implementation of the offloaded data striping protocol and the support for the AM API. In addition, we improved the performances of the driver, see Section 2.5, through multiple optimizations that we felt were out of the scope of this report.

2.4 Multithreaded communications

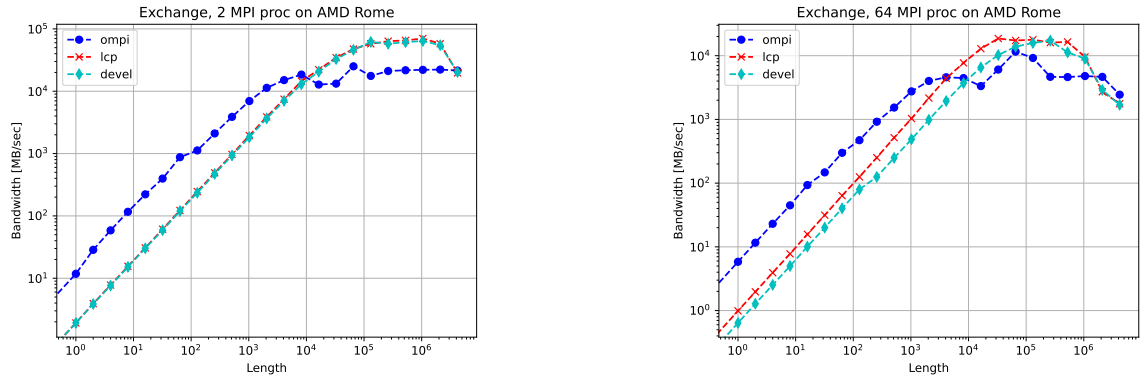
The remaining part of the work in the project was dedicated to inter-thread communications within the MPC framework. We thus backported the previous implementation to the new architecture described in Figure 4 and next we give it a short description.

The communication scheme in MPC involves utilizing shared memory for communication between tasks within a node, while inter-node communication relies on sockets. This model enables an optimized implementation of communications between MPI-tasks, leveraging user-level zero-copy techniques made possible by the shared address space among tasks [30]. As mentioned in Figure 4, another datapath has been dedicated to thread communication.

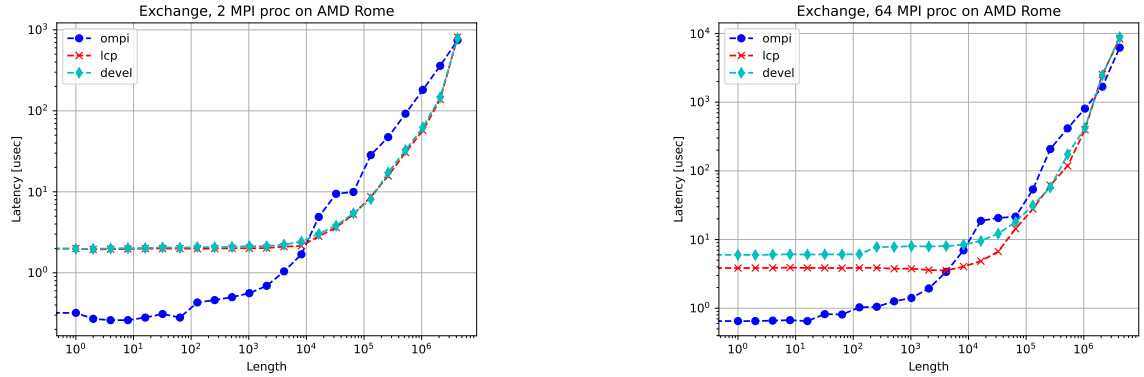
2.5 Evaluation

Validation benchmarks have been run on machines provided by the CEA Inti supercomputer, some of them featuring multi-NIC nodes on BXI network. In this section, we aim at validating our implementation and compare it with other MPI implementations (OpenMPI and older version of MPC).

We validate on differents testbeds:



(a) IMB Exchange bandwidth 2 (left) and 64 (right) MPI ranks on AMD Rome.



(b) IMB Exchange latency 2 (left) and 64 (right) MPI ranks on AMD Rome.

Figure 6: IMB Exchange on a single node: `ompi` is the reference OpenMPI 4.1.5-Bull version, `lcp` is the new implementation and `devel` is the reference old version of MPC.

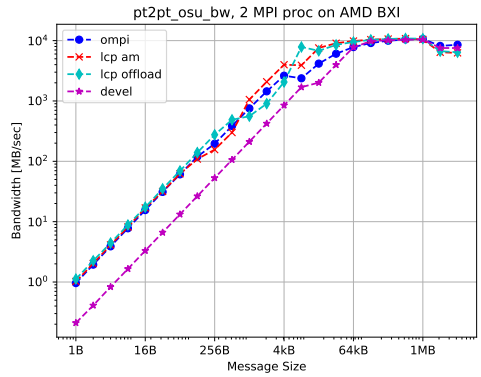
1. AMD Rome: AMD EPYC 7H12 bi-socket (2×64cores) with 256GB of RAM on InfiniBand network.
2. AMD BXI: AMD EPYC 8H12 bi-socket (2×64cores) with 256GB of RAM and 4×100Gb/s BXI network cards.

The first testbed is used to validate only the thread-based version while the second one is used to validate the multirail feature. As a consequence, runs on the first two testbeds will be done on a single node, thus all communications will use shared memory mechanism (either inter-process shared-memory for OpenMPI or inter-thread shared-memory for MPC).

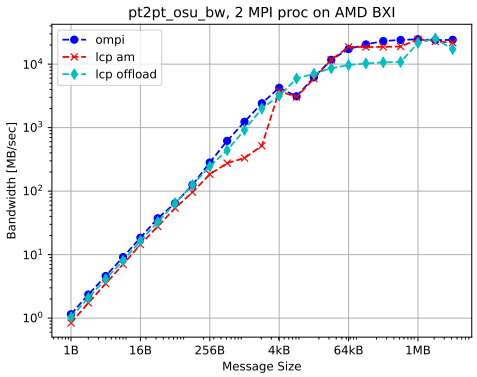
Moreover, results are validated using mini-benchmarks: OSU bandwidth and latency and a subset of MPI1 IMB benchmarks[3]. These benchmarks are performance-oriented and typically measure raw bandwidth and latency.

Figure 6 shows latency and bandwidth results on the IMB Exchange benchmark in its default configuration (number of iterations, message sizes,...). Firstly, our developments

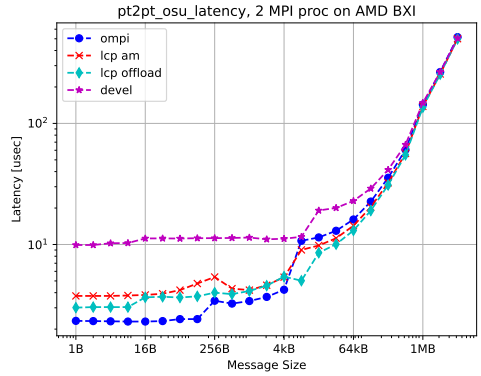
[3] `https://www.intel.com/content/dam/develop/external/us/en/documents/imb-users-guide-607091.pdf`



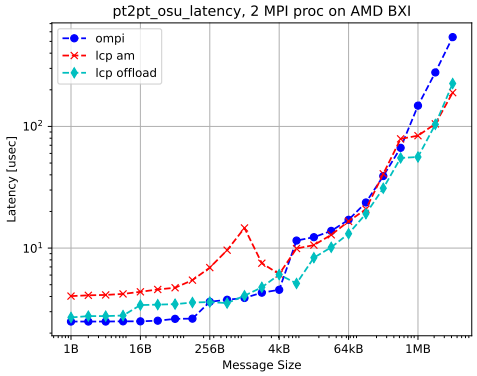
(a) OSU bandwidth with one BXI NIC.



(b) OSU bandwidth with four BXI NICs.



(c) OSU latency with one BXI NIC.



(d) OSU latency with four BXI NICs.

Figure 7: OSU with one BXI NICs (a, c) and 4 (b, d) on AMD BXI. `ompi` is the reference OpenMPI 4.1.5-Bull version, `lcp am` is our version using the AM API (see Section 2.3.2.3), `lcp offload` is the version using tag-matching offloading (see Section 2.3.2.1). Messages of size $> 1MB$ are split in fragments of 512kB and sent using data striping protocol to take advantage of the multirail feature when possible.

show consistent results compared to the previous version of MPC: for 2 MPI ranks, performance are similar while our implementation scales better on 64 MPI ranks. Moreover, bandwidth and latency still suffers for small messages ($<$ 8kB) compared to OpenMPI. Part of the explanation lies in the large overhead introduced by the upper layer of the MPC implementation with duplication of request initialization. Performance could also be improved by implementing "immediate" sends to avoid request initialization for very small messages ($< 256B$). On the other hand, results for large messages are encouraging but a more in-depth study is needed to understand such results. As a consequence, this demonstrates that developments described in Sections 2.2.3, 2.3.2 and 2.4 at least do not create overhead compared to previous reference implementation.

Figure 7 shows results on the OSU bandwidth benchmark for which we used the default configuration. This benchmark is relevant to demonstrate the multirail feature since messages are sent in batches of 64. As a consequence, with multiplexing enabled, they are scheduled successively onto the multiple NICs in a round-robin fashion. For

the OpenMPI 4.5.1-Bull version, MCA parameters `OMPI_MCA_btl_bxi_max_btls` and `OMPI_MCA_btl_bxi_max_cards` have been set appropriately to use multiple NICs.

Figure 7a shows first that both AM and TAG API from the new driver implementation now outperform our reference implementation of MPC (`devel`), especially for small messages. Moreover, it shows competitive bandwidth results compared to state-of-the-art MPI implementation. The peak bandwidth is around 10GB/s and the offloaded version has comparable performance compared to the AM version in terms of bandwidth.

On the other hand, Figure 7b demonstrates the efficiency of the multirail both for multiplexing and data striping. `lcp am` shows unexpected behavior for messages ranging from 512B to 2kB and this should be further investigated. However, our multiplexing algorithm matches performances of `ompi` for large messages (>4kB) which is promising. The data striping protocol for the offloaded version can be seen for message size $> 1MB$.

As a final comment, Figure 7c shows that the offloaded version provids better latency compared to the AM version of the BXI driver. This is due to the zero-copy mechanism provided by tag-matching offloading which thus avoid intermediary copies in the eager protocol. Moreover, other tests has been performed on other testbeds and show consistent results.

2.6 Conclusion

In the report, we provided a high level view of all the developments that have been made within the MPC framework to support the multirail feature. There are three main contributions: first, we improved MPC's software architecture of the communication layer, second we supported data striping on multiple BXI NICs using their tag-matching offloading capabilities, and third we adapted our previous development to support thread-based communications.

In the future, we plan to consolidate the performance results observed in Section 2.5. Also, we would like to take advantage of other offloading capabilities offered by the BXI NIC such as Triggered Operations to design fully-offloaded collective algorithms or using the iovec send operations for complex MPI Datatypes. We could further support BXI extending the current API of the communication module with Atomics.

3 ParaStation MPI

ParaStation Modulo is a comprehensive software suite especially designed for MSA systems. It is the selected middleware powering the DEEP projects but is also extensively used in production environments, e. g., on the JUWELS Cluster/Booster system at JSC [1] and the modular MeluXina system in Luxembourg.

The ParaStation MPI communication stack is a central pillar of ParaStation Modulo and enables efficient communication for MPI applications. Its general software architecture has been presented in Deliverable D4.3 [29]. The following sections focus on the re-design of the RMA interface of the low-level pscom library, especially with respect to its pscom4portals plugin.

3.1 Optimised Low-level RMA

This section presents the work of adding the support for one-sided communication via native RMA operations to ParaStation MPI on top of BXI. To benefit from the RMA capabilities provided by BXI, the pscom4portals plugin has been extended to leverage the native RMA support provided by the Portals 4 API for one-sided communication operations (e. g., RMA put and get).

The presented work builds upon the re-design of pscom's RMA interface in the context of the DEEP-SEA project. It enables the efficient execution of both MPI one-sided communication operations and PGAS workloads on top of a single low-level communication layer, thereby improving the composability of different programming models within MSA systems. The internal plugin interface has also been changed/extended in the context of this re-design. Therefore, the pscom4portals plugin had to be adapted accordingly. This new RMA implementation is also aimed at improving the interoperability and composability of MPI and PGAS environments. This allows efficient use of RMA operations and one-sided communications offered by MPI and PGAS programming models.

3.1.1 Architecture

The RMA support provided by pscom is implemented in a layered architecture (cf. Fig. 8). The upper layer offers interfaces such as for memory region registration, RMA communication, and RMA synchronisation, which can be directly used by higher-level communication semantics such as MPI one-sided communication or PGAS concepts. Another purpose of the upper layer is to provide software layers running on top with efficient access to the RMA capabilities of the different hardware interconnects supported by pscom. This is achieved by the lowest layer, i. e., the plugin layer. This interfaces via the internal plugin interface with the upper, hardware-independent pscom layer and implements the RMA support including memory region registration, RMA communication, and synchronisation within the plugins for the different interconnects. In the case of pscom4portals, this is realised by leveraging the one-sided communication operations offered by the Portals 4 API, e. g., `PtlMEAppend` to create memory region, `PtlPut` for RMA put communication, and event the waiting function `PtlCTWait` for synchronisation. The implementation details of native RMA support are described in the following sections.

To utilise such native RMA capabilities, the exposed memory commonly has to be registered with the Host Channel Adapter (HCA). The pscom library offers the function `pscom_mem_register` for memory region registration in the hardware-oriented plugin layer. To register its memory region, the initiator process can execute the following two steps:

1. Call this function `pscom_mem_register` and obtains a memory region handle `pscom_mem_t` and a buffer containing information regarding the memory region.

2. Send this obtained buffer to the remote process to which the memory region is to be exposed.

To have an access to the exposed memory region at initiator, the remote process can perform the following steps:

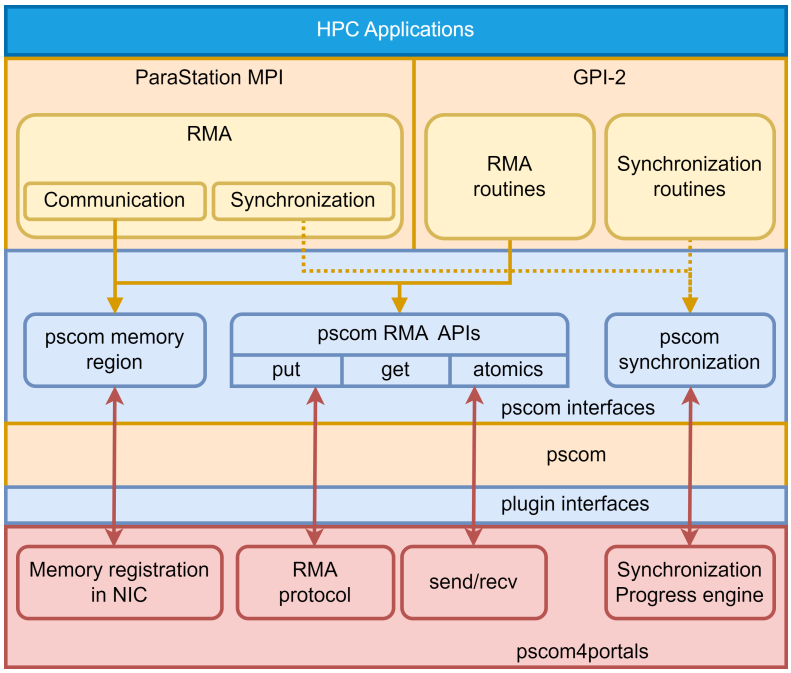


Figure 8: The layered architecture of the RMA support in pscom.

1. Receive the buffer containing the information of memory region from the initiator process
2. Generate a connection-bound remote key object (i. .e, `pscom_rkey_t`) corresponding to the exposed memory via the pscom API `pscom_rkey_generate` using the received buffer.

The generated remote key object can be then utilised for native RMA communication to the exposed memory region at the initiator. The new pscom RMA API also provides means for de-registering memory regions (`pscom_mem_deregister`), for freeing the resources associated with the remote key object (`pscom_rkey_destroy`), and for releasing the buffer containing the remote key once it has been sent to the remote process (`pscom_rkey_buffer_release`).

The communication-related part of the new pscom RMA API contains six elementary functions (cf. Lst. 3). These serve as the basis to implement higher-level APIs, e. g., as defined by the MPI standard.

```
pscom_post_rma_put(pscom_request_t *request, pscom_rkey_t *rkey,
                   int flag);
pscom_post_rma_get(pscom_request_t *request, pscom_rkey_t *rkey,
                   int flag);
pscom_post_rma_accumulate(pscom_request_t *request,
                          pscom_rkey_t *rkey, int flag);
pscom_post_rma_get_accumulate(pscom_request_t *request,
                              pscom_rkey_t *rkey, int flag);
pscom_post_rma_fetch_and_op(pscom_request_t *request,
                            pscom_rkey_t *rkey, int flag);
pscom_post_rma_compare_and_swap(pscom_request_t *request,
                                void *origin_addr,
```

```
                    void *compare_addr,
                    void *result_addr,
                    pscom_rkey_t *rkey, int flag);
```

Listing 3: RMA communication APIs offered by the pscom library.

These functions support both native RMA and RMA based on two-sided communication semantics. Native RMA not only requires memory region registration and remote key generation, but also has to fulfil certain limitations of the network hardware (e.g., data type, data length, and accumulate operation types). If native RMA is not supported by the hardware, pscom will automatically fall back to the RMA based on two-sided communication semantics. The use of the pscom's two-sided communication functions for realising the RMA functionalities requires additional information packed into the extended header of pscom messages and shifts certain processing steps to the target sides, e. g., the unpacking of complex data types, and the execution of accumulate operations. The implementation and usage details of RMA based on two-sided communication semantics can be found in Deliverable D3.2 of the DEEP-SEA project [31].

The pscom library offers interfaces to perform native RMA synchronisation by flushing individual connections (`pscom_connection_flush`) or entire sockets (`pscom_socket_flush`) comprising multiple connections. These functions guarantee that the native RMA operations, which are issued on a socket or connection before this synchronisation call, are completed at both the origin and target side. The origin side denotes the process that performs the RMA call, and the target is the process in which the memory is accessed.

Note that the current design status of the interfaces mentioned above is still in an early stage and the interfaces might be changed to match the needs of the intended users in the future.

3.1.2 Memory Registration and Remote Key Generation

Figure 9 visualises the procedure of memory region registration and remote key generation in the pscom4portals plugin layer. At the target process which exposes a memory region, a Matching List Entry (ME) with specific `match_bits` is created by calling `PtlMEAppend` during registration (i. e., the invocation of `pscom_mem_register`). The progress engine in the pscom4portals plugin is triggered to guarantee that the `PTL_EVENT_LINK` event is logged, which indicates that the ME has been appended to the Portals Table Entry (PTE). Then the specified `match_bits`, Process Identifier (PID), and Portal Table Index (PTI) are packed into the remote key buffer and returned to the user. Afterwards, this remote key buffer is sent to the remote process. The remote process can locally generate a remote key via `pscom_rkey_generate` with the remote key buffer. For this, the received data is unpacked into a remote key data structure in the pscom4portals plugin layer.

3.1.3 RMA Communication

RMA communication operations (i. e., `put`, `get`, and atomic operations) have been implemented in the pscom4portals plugin leveraging native hardware support provided by the underlying BXI hardware. Therefore, these functions map onto their Portals 4 counterparts `PtlPut`, `PtlGet`, `PtlAtomic`, `PtlFetchAtomic`, and `PtlSwap`. The PID and PTI in

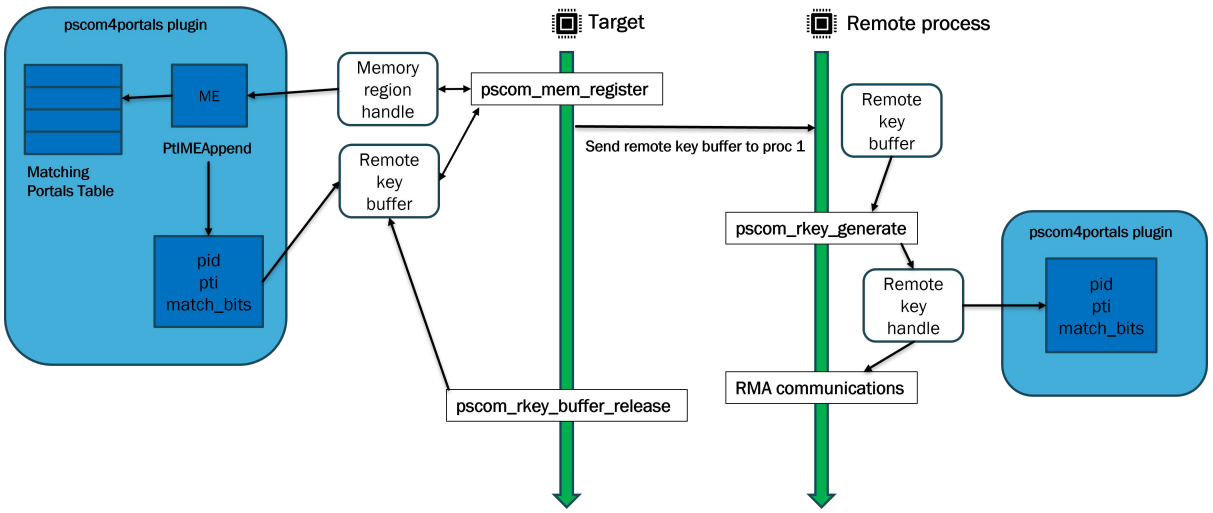


Figure 9: Procedure of memory region registration at the target and remote key generation at the remote process based on pscom4portals plugin.

the remote key object are used to specify the correct destination and `match_bits` grants the access right to the exposed memory region.

Currently, the pscom library offers new interface for checking whether native RMA is possible for a given connection. This allows software layers on top to determine whether further means for synchronisation are required. Various parameters, e. g., data type and atomic operation type, have to be passed to this function. In the pscom4portals plugin, there exist certain limitations of data type, data length, and atomic operation types due to Portals 4 API. For example, non-contiguous data types are not supported by Portals 4. These limitations are determined during the initialisation of the pscom4portals plugin, e. g., in the case of pscom4portals `PtlNIInit` will return the relevant parameters supported by the underlying hardware.

3.1.4 Synchronization

The support for socket flush and connection flush within pscom4portals is realised by using light-weight Counting Events (CEs) of Portals 4. Figure 10 shows how the RMA synchronisation is achieved in the pscom4portals plugin by using example of an RMA put request. A memory descriptor is created and bound to the whole virtual address during the initialisation of pscom4portals plugin at the origin side. An Event Queue (EQ) and a CE, where information about the operations performed on the memory descriptor are recorded, are also attached to this memory descriptor. The EQ only logs the failed events and the link events `PTL_EVENT_LINK`. When a put request `PtlPut` is initiated, an acknowledgment event is requested and the counter `Counter1` is incremented at the origin side. The CE attached to the memory descriptor will log this acknowledgment event at the origin side. The `PtlCTWait` function will block and wait for the CE (`Counter2`) to reach a given value set by `Counter1`. This guarantees that the number of complete RMA communication matches that of issued RMA communication on this socket or connection.

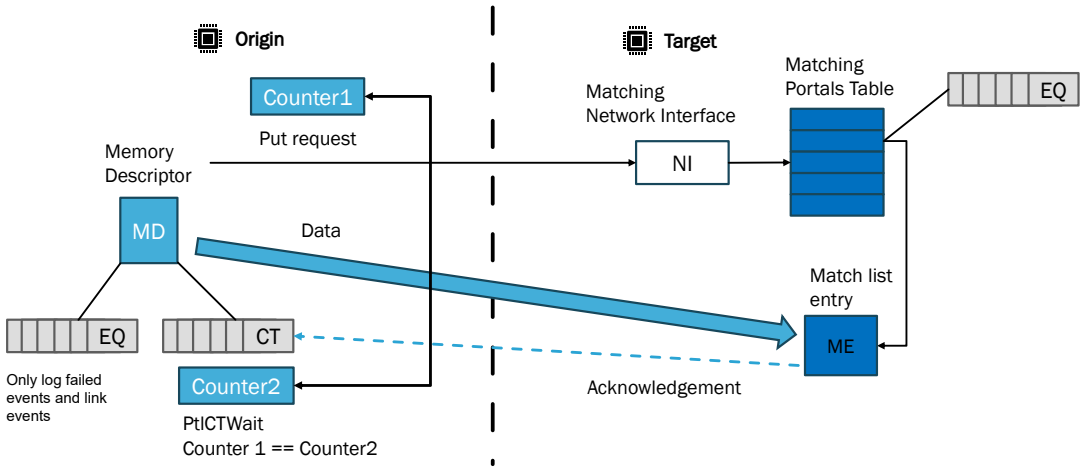


Figure 10: Mechanism of RMA synchronization based on a counting event in pscom4portals plugin using Portal 4 API.

Here are some implementation examples of RMA synchronization in ParaStation MPI. In `MPI_Win_fence`, this can be implemented by flushing sockets or connections, followed by a barrier call. General active target synchronization is implemented with an acknowledgement to the target after flushing the connection at the origin side. Passive target lock and flush can be also implemented with an acknowledgement in a similar manner as active target synchronization. The implementation of *post-start-complete-wait* and *lock-unlock* synchronization of hardware-enabled RMA on top of BXI is shown in Figure 11.

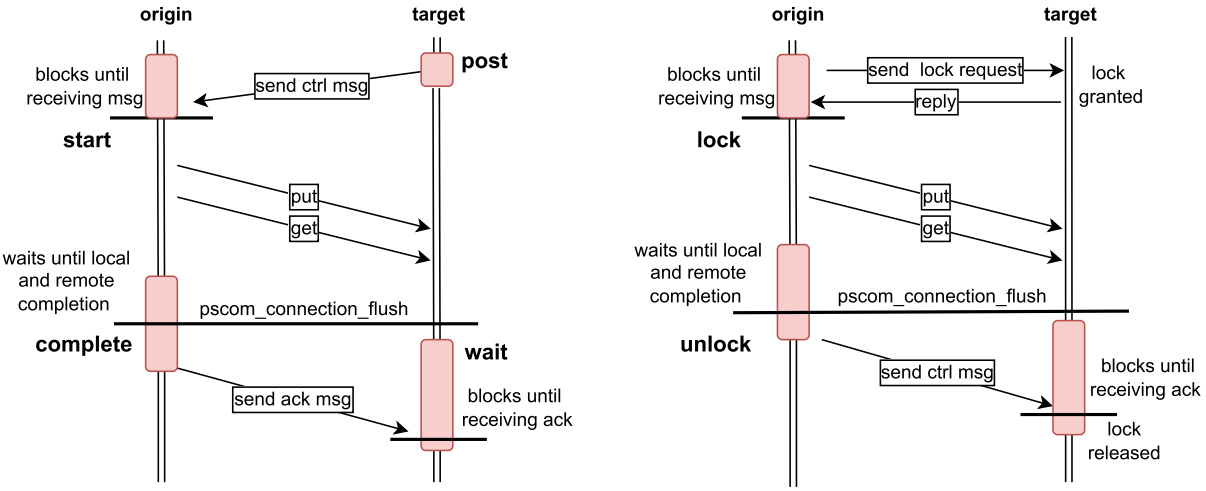
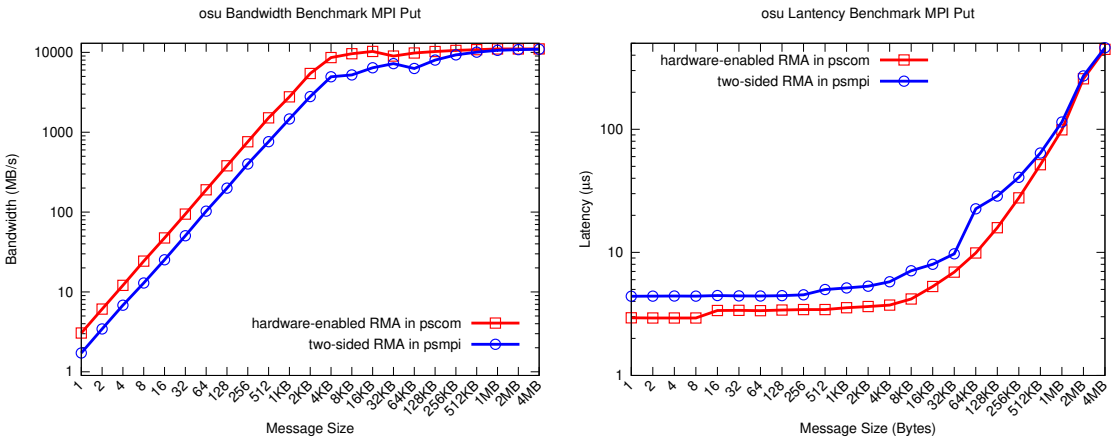


Figure 11: Implementation examples of active target synchronisation of post-start-complete-wait (left) and passive synchronization of lock-unlock (right) of hardware-accelerated RMA supported by Portals 4 API.

3.2 Evaluation

The DEEP system served as platform for conducting a preliminary evaluation of the native RMA support provided by the pscom4portals plugin. Therefore, both functionality and performance have been analysed on the MPI level. This analysis compares a new implementation of MPI one-sided communication leveraging the new RMA interface of pscom with an old implementation based on two-sided communication semantics in ParaStation MPI. Figure 12 presents the throughput and the latency of `MPI_Put` obtained by running the `osu_put_bw` and `osu_put_latency` benchmarks of the OSU benchmark suite on the DEEP system. We use their default RMA configuration, i. e., the window creation is done by calling `MPI_Win_allocate` and synchronisation is ensured by using `MPI_Win_flush`. The implementation based on the new pscom RMA API shows a higher throughput and lower latency than the old implementation based on two-sided communication semantics in ParaStation MPI. Especially for small messages (<= 1024 B), the latency of the implementation based on the new pscom RMA API is around 30 % lower than that of the old implementation based on two-sided communication semantics.

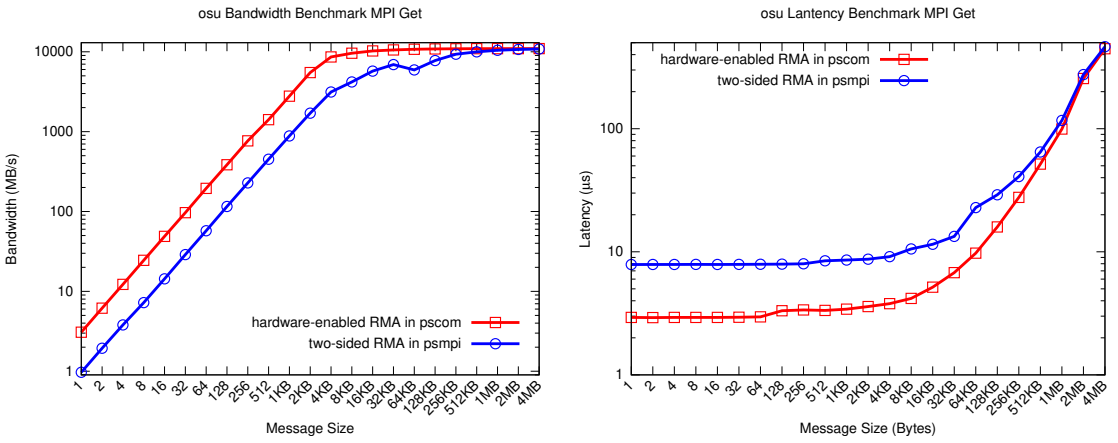


(a) (b)

Figure 12: Bandwidth (a) and latency (b) comparison of `MPI_Put` communication based on hardware acceleration and two-sided communication semantics for OSU one-sided micro benchmarks.

Figure 13 presents a similar comparison for `MPI_Get` for both the throughput and the latency. Again, the implementation base on the new pscom RMA API shows a much better performance compared with the two-sided RMA communication semantics in ParaStation MPI. The latency with the new RMA implementation is significantly reduced by approx. 60 % compared with the old implementation.

Figure 14 shows the latency evaluation of `MPI_Accumulate`. The BXI hardware supports a maximum data length for atomic operations (i. e., `PtlAtomic`) of 1024 B. Therefore, small messages (<= 1024 B) experience an important performance benefit of around 30 % latency reduction when using the new implementation of MPI one-sided as compared with the two-sided-based implementation. However, once this threshold is exceeded, pscom automatically falls back to a two-sided-based implementation of



(a) (b)

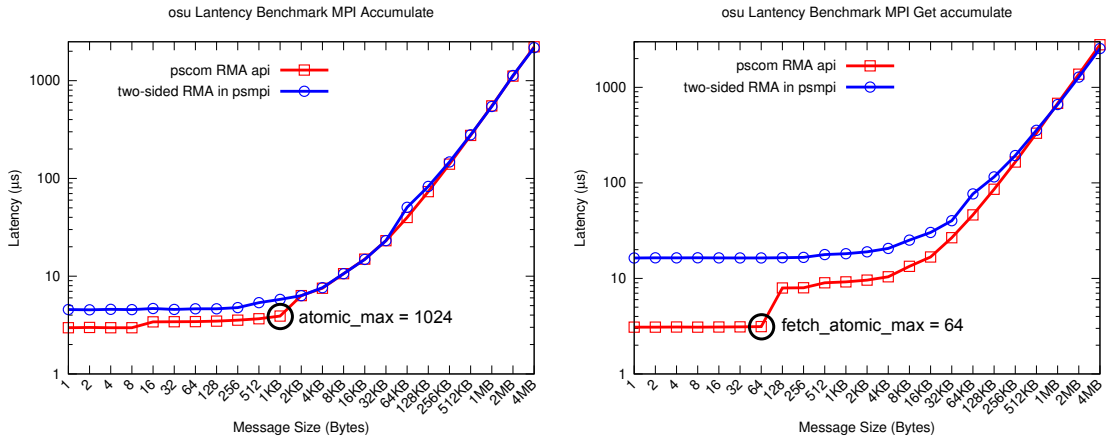
Figure 13: Bandwidth (a) and latency (b) comparison of `MPI_Get` communication based on hardware acceleration and two-sided communication semantics for OSU one-sided micro benchmarks.

atomics. Therefore, the latencies for larger messages are on par with the old implementation of MPI one-sided within ParaStation MPI. In case of `MPI_Get_accumulate`, the data length limit of `PtlFetchAtomic` is 64 B. For small messages (<= 64 B) we observe a latency reduction of approx. 80 % with the new implementation. We also see a similar increase of the `MPI_Get_accumulate` latency at this threshold (64 B), where the RMA mode is switched from hardware-accelerated RMA to RMA via two-sided semantics in pscom, such that an increase of latency is observed for the case of pscom RMA API. Figure 15 shows the latency of `MPI_Compare_and_swap` and `MPI_Fetch_and_op` respectively. The default data size of atomic operations in OSU benchmarks is set to 8 B. The window is created with `MPI_Win_allocate` and the synchronization function is `MPI_Win_flush`. The latency obtained from the new pscom RMA API is about 80 % lower than the two-sided-based implementation. This is due to an internal lock within ParaStation MPI that is required to guarantee the atomicity of the two-sided-based approach.

3.3 Transparent Network Bridging

The pscom4portals plugin in conjunction with pscom's gateway capabilities enables the transparent bridging to and from the BXI network. This way, MSA systems exhibiting a heterogeneous network landscape can run MPI workloads across multiple modules even if they use different underlying interconnection technologies including BXI.

The DEEP system (cf. Tab. 1) is an example for such a setup: besides the cluster nodes and the Extreme Scale Booster (ESB) nodes which are connected via IB, it contains a few nodes using BXI as the high-speed interconnect. This testbed therefore served a preliminary study to assess the network bridging between IB and BXI. The results of this preliminary analysis are shown in Figure 16. Therefore, one of the four BXI nodes has been equipped with an IB HCA. This node is therefore able to serve as a *gateway* node, federating the IB and the BXI fabric.



(a) (b)

Figure 14: Comparison of Latency of `MPI_Accumulate` (a) and `MPI_Get_accumulate` (b) via pscom RMA API and the two-sided RMA communication in ParaStation MPI for OSU one-sided micro benchmarks.

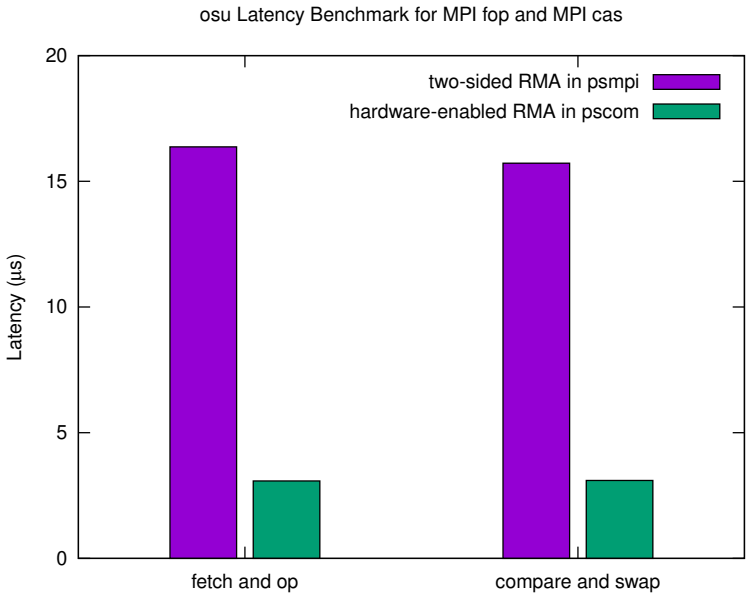


Figure 15: Latency comparison of `MPI_Fetch_and_op` and `MPI_Compare_and_swap` communication based on hardware acceleration and two-sided communication semantics for OSU one-sided micro benchmarks.

	Cluster Nodes	BXI Nodes
Architecture	Intel(R) Xeon(R) Gold 6146	Intel(R) Xeon(R) Gold 5122
Node Count	12	4
Socket Count	2	1
Memory per Node	192 GiB	48 GiB
Interconnect	ConnectX-5	BXI 1.3

Table 1: The DEEP system serving as testbed for the evaluation of ParaStation MPI's support for network bridging across IB and BXI.

The main goal of this test was to demonstrate the ability of ParaStation MPI to transparently bridge MPI traffic to/from the BXI network. Therefore, the performance figures should be considered preliminary. Further optimisations would be required to improve the network throughput via the gateway node. Especially, the plugin-specific parameters (e. g., the number and size of pre-allocated send and receive buffers) have a strong impact on the resulting performance.

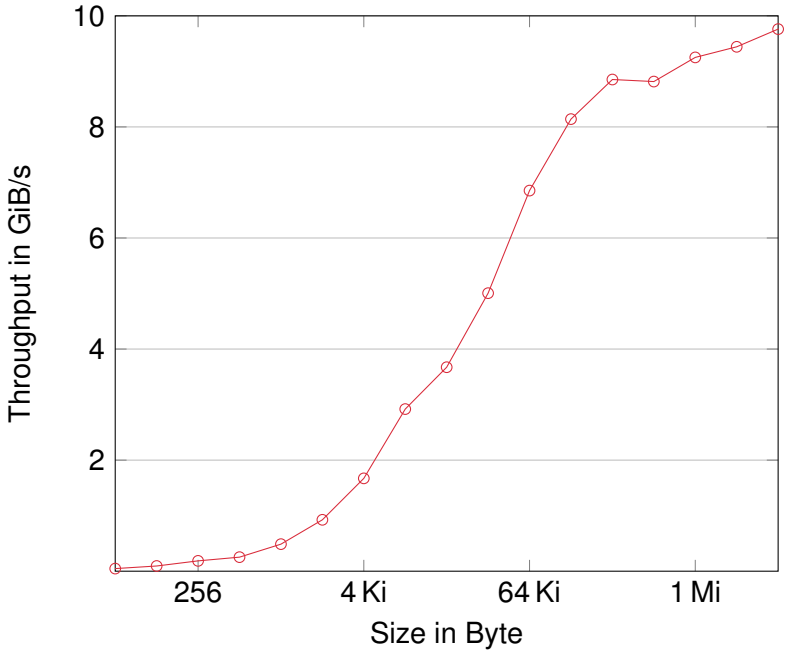


Figure 16: A throughput analysis of the network bridging between InfiniBand and BXI on the DEEP system. The preliminary results were obtained by running the `osu_bw` benchmark between a cluster node and a BXI node on the DEEP system. Performance improvements can be expected by implementing further optimisations especially w.r.t. to the tuning of plugin-specific parameters.

3.4 Conclusion

In conclusion, the newly implemented pscom RMA API using Portals 4 on top of BXI shows important performance improvements compared with the former two-sided-

based implementation in ParaStation MPI. A latency reduction of approx. 30 % for `MPI_Put` and `MPI_Accumulate` and 60 % for `MPI_Get` for small messages ($\leq$ 1024 B) with the new implementation is obtained. We also observe a significant latency reduction up to 80 % for MPI atomic operations, such as `MPI_Accumulate`, `MPI_Compare_and_swap` and `MPI_Fetch_and_op`. These developments allow for efficient implementations of MPI one-sided communication as well as PGAS programming models (e. g., the GPI-2 implementation of the Global Address Space Programming Interface (GASPI) standard) on top of pscom.

Furthermore, the preliminary evaluation of pscom's network bridging capabilities demonstrates its viability also for MSA systems using BXI among other high-speed interconnects.

4 sPIN: High-performance streaming Processing in the Network

4.1 Introduction

SmartNICs are a recent movement towards offloaded packet processing to free the CPU from packet processing and thus spend more time handling the typical computation tasks. They come in different programming models and dataflow models. Among different paradigms, sPIN [18] developed at ETH Zurich proposes network accelerators with a micro-architecture optimised for packet processing and fine-grain memory hierarchies and data movement acceleration. It offers precise control to the programmers to build high-performance networked applications that are offloaded completely to the SmartNIC.

The sPIN paradigm has been evaluated extensively with diverse networked applications [10, 5, 9], showcasing its capability of offloading complicated applications to a sPIN-based network accelerator. Up to now, however, all evaluations of sPIN took place in simulation and there lacked a real-world end-to-end demo on hardware. While simulation works well to demonstrate capabilities of the paradigm in a *synthetic* environment, an end-to-end evaluation involving all parts of the final system would uncover unforeseen design and implementation shortcomings and offer valuable insights to further improve the paradigm.

In this section of the introduction, we give a brief summary of the contributions of the work described in this deliverable. For more technical details we refer the reader to the MSc thesis of Pengcheng Xu [40], supervised by the SPCL group at ETH Zurich. An overview functional diagram of the system is shown in Figure 17.

First of all, we built *FPsPIN*, the first full-system demo of sPIN in hardware based on the PsPIN [11] implementation of sPIN and the Corundum [15] open-source Ethernet Network Interface Card (NIC). This allows fast testing of packet *handlers* (code that runs on the sPIN cluster) in comparison to the slow cycle-accurate simulator. The hardware (Section 4.2) and software (Section 4.3) components bridge the missing parts in the PsPIN prototype to allow sending and receiving of packet data from real NICs and, most importantly, completes the *host-side* programming model of sPIN. This allows development of complete sPIN applications with both the NIC-side handlers and

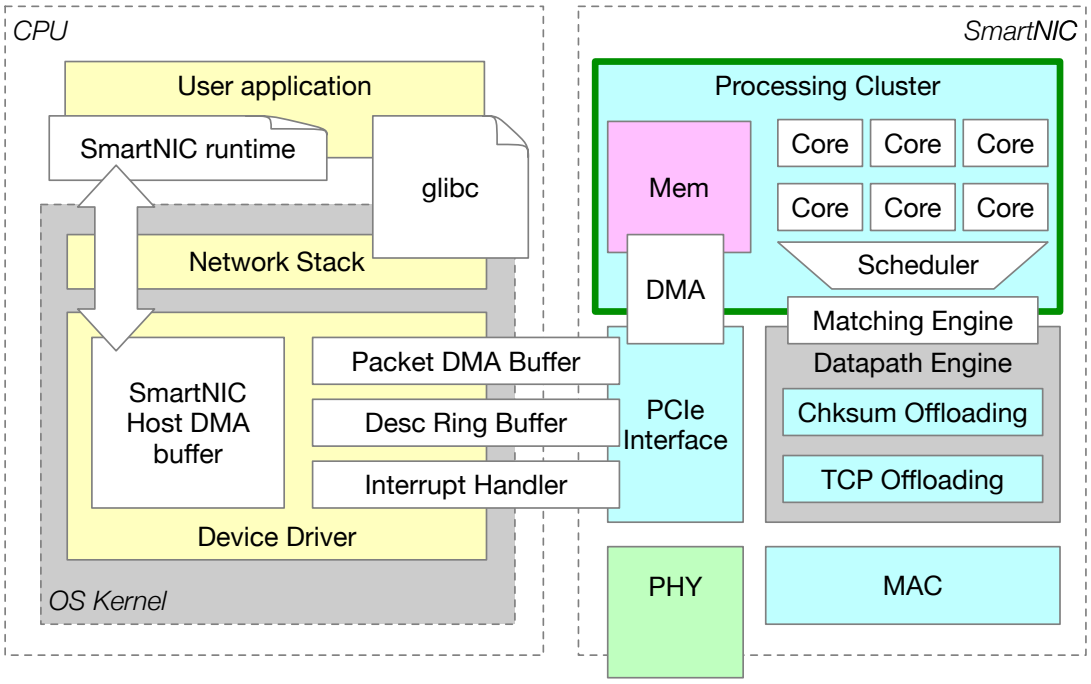


Figure 17: Overview of the complete server system, showing the software stack on the CPU and hardware components on the SmartNIC. The green box marks the existing PsPIN processing cluster available from previous work. Everything else needs to be developed, integrated and tested.

host-side application. In all, the demo system greatly facilitates development both of the sPIN platform as well as applications designed for it.

An important yet largely unexplored benefit of sPIN is the possibility of *computation/communication overlap* by offloading packet processing tasks to the SmartNIC. We implement synthetic ping-pong and file transfer benchmarks to demonstrate the e2e latency and throughput of the system in ideal situations. We port the MPI Datatypes [33] sPIN handlers [10] to the FPsPIN platform (Section 4.5.4) to demonstrate the ratio of overlapping between the computation and communication tasks, as well as interference from each other. These demonstrations show sPIN's potential of accelerating networked applications and improving efficiency, but also open up interesting research questions about the architectural design of packet processing units in SmartNICs. These demonstrations also show that the RED-SEA KPI of developing a network offload soultion that achieves 90% communication-computation overlap on sufficiently large message packing/unpacking has been fully achieved.

Last but not least, we discovered numerous shortcomings and points of improvement in the sPIN specification [18] (Section 4.4) during the development of the FPsPIN prototype system. We discuss about the issues closely with the sPIN team and work together towards a more complete and sensible specification for other implementations of the paradigm. Several of the proposed changes have already been incorporated back into the specification.

4.2 FPsPIN Hardware Implementation

PsPIN is a RISC-V-based packet processing cluster implementing the sPIN in-network-computing paradigm. However, PsPIN itself does not consist of a fully functional SmartNIC due to the lack of capability to receive and send packets; it also lacks an interface to read from and write to the system memory. The following three classes of hardware components need to be implemented to achieve full functionality of a sPIN NIC:

- the *data path*: the PsPIN cluster should be able to receive packet data from the network and send a reply back into it;
- the *control path*: the PsPIN cluster and other components should be configured from the host over various control registers and program memory (code and data); and finally,
- the *host-side DMA*: the PsPIN cluster should be able to read from and write to the main memory on the host system to establish the full sPIN programming model.

An overview of all the hardware components is shown in Figure 18. We now walk through the design and implementation of these modules in more detail.

Module Name	Description
`pspin_host_dma`	Host acdma adapter
`pspin_ingress_datapath`	Collective ingress data path wrapper
`pspin_her_gen`	Handler Execution Request (HER) generator
`pspin_ingress_dma`	Ingress DMA engine
`pspin_pkt_alloc`	Packet buffer allocator
`pspin_pkt_match`	Packet matching engine
`pspin_ctrl_regs`	Control registers adapter
`pspin_egress_dma`	Egress DMA engine

Table 2: Description of the modules shown in in Figure 18.

4.2.1 Control Path

The control path handles configuration of the PsPIN cluster as well as the various data path components *before* the actual execution of handler code on the cluster. There are three important control-path tasks to perform from the host, all of which are implemented over Corundum's slow-path 32-bit Advanced eXtensible Interface (AXI)-Lite interface with an address bus of 16 bits:

- to toggle various control registers to the PsPIN cluster and data path components;
- to read back standard output produced by PsPIN (i.e., `printf`); and
- to load program code and data onto memory in the PsPIN cluster.

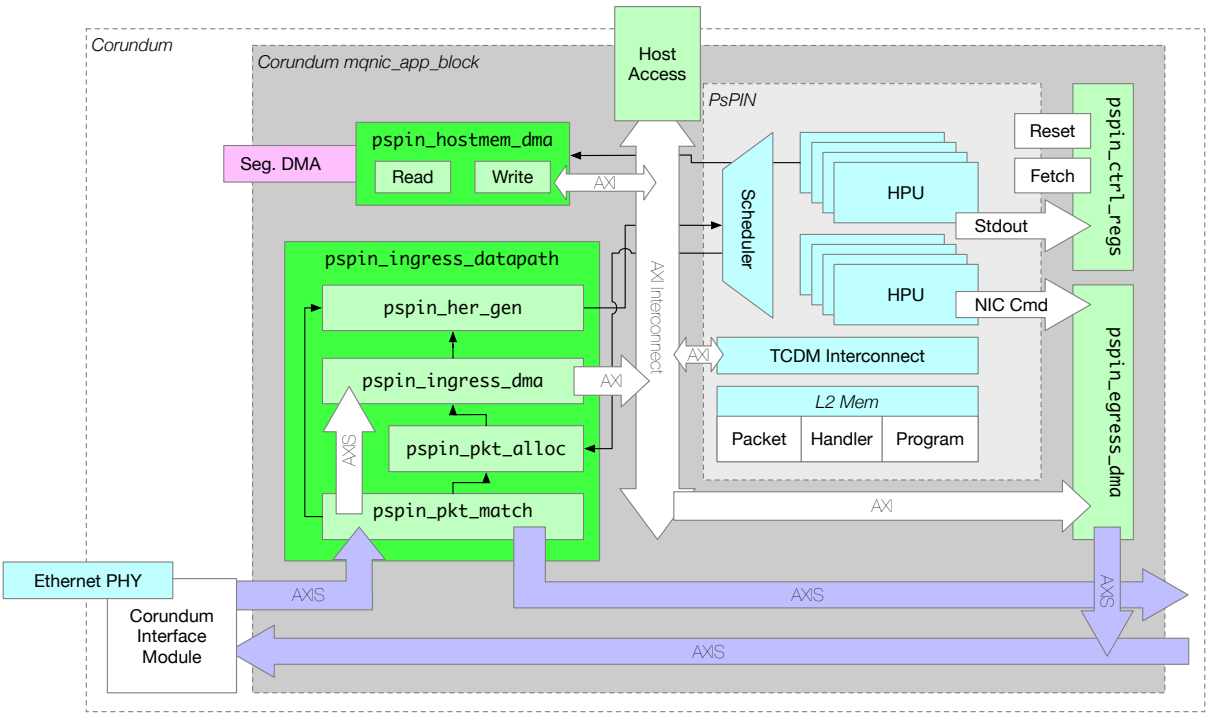


Figure 18: Overview of the FPsPIN hardware. A description of the hardware function blocks is shown in Table 2. Blocks marked in green are the modules implemented as part of this project to bridge the PsPIN cluster to Corundum.

4.2.1.1 Control registers

The control registers are configured through the `pspin_ctrl_regs` module. The module exposes an AXI-Lite slave towards the AXI-Lite interconnect and converts this into simple `valid`-guarded interfaces for PsPIN and various data path components to consume. Some signal groups have requirements on *consistency of update*, that is, the signals in the same group should always be consistent and no partial updates should be visible to the components being controlled. Checks for this requirement happens in the kernel driver. An overview of the exposed control signals from `pspin_ctrl_regs` is shown in Table 3.

Name	Direction	Description
`cl_fetch_en`	O	Fetch-enable control to PsPIN
`aux_rst`	O	Auxiliary reset for PsPIN and data path
`cl_busy`	I	Cluster busy status from PsPIN
`mpq_full`	I	Message Processing Queue (MPQ) full status bitmap
`match_*`	O	Matching engine configuration
`her_gen_*`	O	HER generator configuration
`stdout_*`	O	Standard output readback

Table 3: Overview of the control wires exported by `pspin_ctrl_regs`. The meaning of these control wires will be introduced in the coming sections.

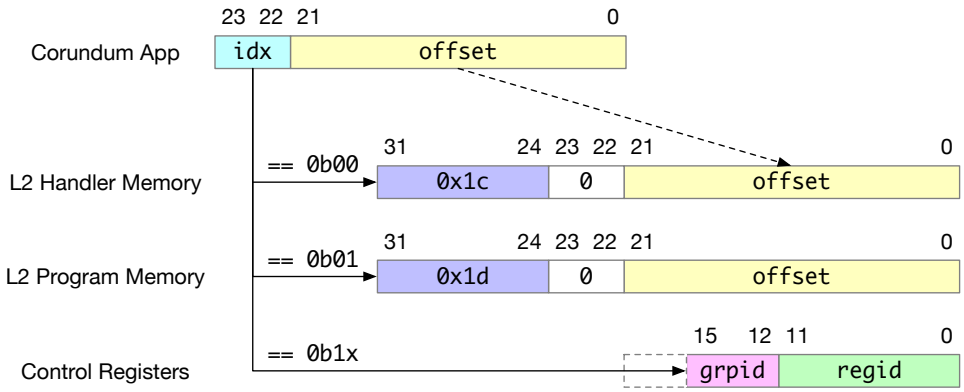


Figure 19: Address translation from the Corundum application control space. Access to the PsPIN host access space always have the top bit as 0; we use the second top bit to select the correct memory area in PsPIN. Access to the configuration registers have the top bit set to 1; the 22-bit offset is decoded as the 16-bit control register address and the top 6 bits ignored.

The control registers module is designed to allow reconfiguration during normal operation of the system. Therefore, components that take configuration data from the module are expected to have a explicit *valid* signal, if they expect consistency between multiple registers. The software that controls these register would then de-assert valid, change the registers, and then reassert valid, such that the downstream module can have a consistent configuration.

We group registers by the subsystem they control (e.g. the matching engine or the HER generator) and assign a block of address in the control register address space. We then refine these groups into subgroups that each of them control a specific field of configuration; some of the subgroups contain multiple identical register instances (e.g. for multiple rulesets in the matching engine). As shown in Figure 19, the 16-bit control register address uses the top 4 bits ("grpid") to address the register groups and the lower 12 bits ("regid") to address the subgroup and register instances. We do not explicitly define a subgroup field in the address due to different subgroup sizes across different groups.

Verilog modules that interact with the control register system are written in a template language, namely Jinja [23], that abstracts the exact register definitions away. A generator written in Python processes all Verilog templates w.r.t. the register metadata and emits the final source file ready for synthesis. Such an approach eliminates the tedious and error-prone maintenance of repetitive register definitions and proved to be crucial as the number of control wires grows. The generator also derives part of the kernel driver that later exposes these control registers as described later in Section 4.3.1.

4.2.1.2 Standard output access

To facilitate debugging of handler code on the PsPIN cluster, we implement a readback mechanism for the characters printed by the RISC-V cores. The core executes `putchar` to write characters into the `apb_stdout` module. Different cores write to separate addresses exported by the module, allow-

ing the module to demultiplex the incoming characters. The module enqueues the characters together with the source core ID in a First-In-First-Out (FIFO). The FIFO is then read out from `pspin_ctrl_regs`. To avoid introducing module ports on all levels of RTL hierarchy, we utilise the *hierarchical reference scope* [38] feature of Verilog to connect the output ports from `apb_stdout` directly. Finally, the host can read back the enqueued characters by reading out the `stdout_*` registers through the register interface, demultiplex, and store the output as logs for future inspection.

4.2.1.3 Code and data download

The code and data of the packet handler program on PsPIN need to be loaded into the *program memory* in PsPIN before we can start scheduling packets to execute on the HPUs. The program memory is accessible through the *host slave* port on the PsPIN cluster. This port also allows write to the other memory area, the *handler memory*, to allow writing either static or dynamic configuration data by the host. Together, this allows loading compiled PsPIN program images onto the cluster memory.

We implement such access by connecting the upstream AXI-Lite port from Corundum, through a AXI-Lite interconnect and a AXI-Lite to AXI4 adpater, to the host slave port. Note that the PsPIN host access address space on the host slave port is 32-bits. However, we only have a 24-bit address space from the application block control port from Corundum. Therefore, we perform a *compression* in the address space by mapping the two memory areas closer together into the application control port address space; we demonstrate this in Figure 19. The FPsPIN kernel module (Section 4.3.1) will encode the PsPIN memory accesses according to this mapping.

4.2.2 Data Path

PsPIN, being a packet processor, needs to have access to the receive and transmit paths in the NIC to function properly. We introduce in this section the design and implementation of the ingress and egress data path engines that gives PsPIN access to the packet data path.

4.2.2.1 Attach points of the data path

Corundum provides access to raw Ethernet frames over the AXI Stream interface. Three attachment points are available to the application block for reading ingress Ethernet frames out, as well as injecting egress frames:

Direct The AXI Stream interface directly after the Ethernet MACs and before most Corundum modules. The interfaces are synchronous to the MAC clock (322.265625 MHz for 100 Gbps Ethernet). This offers the lowest possible latency from the application block.

Sync The AXI Stream interface after the clock domain crossing (cdc) FIFO for each port. These interfaces are synchronous to the Corundum core clock (250 MHz). They offer comparatively low latency.

Interface The AXI Stream interface after the main packet aggregation FIFO per interface. These interfaces are per interface (instead of per port; for example a 100 Gbps interface could be split into 4 25 Gbps ports, and are the simplest to process. They are synchronous to the Corundum core clock (250 MHz).

The Field Programmable Gate Array (FPGA) board we use, as described in detail in Section 4.5.1, has two 100 Gbps interfaces; each interface can be further split up into 4 25 Gbps ports. For simplicity of implementation, we attach the PsPIN data path at the *interface* attach point, such that we don't have to multiplex traffic from different ports by ourselves.

4.2.3 Ingress

After a packet has arrived at the *interface* attach point, multiple tasks need to be done for an ingress packet before it lands in PsPIN memory and is ready for processing. We implement four separate functional blocks as follows; together they form the ingress data path module (`pspin_ingress_datapath`):

- `pspin_pkt_match`: match if the packet is to be processed by the SmartNIC cluster or to be forwarded to the normal Corundum data path;
- `pspin_pkt_alloc`: allocate buffer for the incoming packet in the L2 packet buffer, free the buffer once it finishes processing;
- `pspin_ingress_dma`: DMA write the packet data into the L2 packet buffer
- `pspin_her_gen`: generate the HER to the PsPIN cluster

We explain in detail the design of these modules. Note that common design considerations presented in Section 4.2.6 apply to these modules.

4.2.3.1 Packet matching engine

`pspin_pkt_match` exposes one AXI-Stream slave (`s_axis_nic_*`) towards the upstream packet data that comes from the application block interface in Corundum. It further exposes two AXI-Stream master ports towards the downstream packet processing logic. One of them (`m_axis_pspin_*`) forwards the matched packet data to the rest of the data path components for further processing. In addition, the module also exposes metadata for the matched packet over a ready-valid interface (`packet_meta_`) providing the downstream components with the following information:

Message ID from the sPIN lightweight messaging protocol (SLMP) packet header (see Section 4.4.1 for details of the SLMP protocol), for the HER generator

End of Message (EOM) bit as specified by the matching ruleset, for the HER generator

Ruleset ID of the matching ruleset, for the HER generator to select the correct sPIN Handler Execution Context (EXTX)

Length of the packet, for the packet buffer allocator

Since we need to count the length of the packet, the packet metadata can only be generated after that the packet has been transferred on the AXI-Stream interface. A later stage in the data path (the ingress DMA engine) will reverse this dependency by buffering the packet data.

We adopt a simple approach to define the matching rules similar to the IPTables U32 match [6]. The matching engine provides a configurable number of *rulesets*. We expose ruleset configuration to the host as control registers. Each ruleset is defined by a configurable number of *matching rules* for the *matching units*, which, each one on its own, matches against a 32-bit word of the packet and produces a boolean output. Given index I, 32-bit mask M, 32-bit start value S, and 32-bit end value E, the matching unit output is defined as:

$$\text{Output} := S \leq (\text{Packet}[4I : 4I + 3] \,\&\, M) \leq E$$

Each ruleset defines a *mode* in which the output from the matching units are combined into the match output of the ruleset. We currently implement two modes: `MATCH_AND`, which combines the match unit outputs with a logical AND; and `MATCH_OR`, for a logical OR. The module is designed such that it is easy to add another combination mode, if such a use case rises (for example an *exactly-one* combination mode). If any of the installed rulesets matched on the packet, the module marks the packet as matched for further processing in the data path. The module then sets the *ruleset ID* metadata of the packet accordingly for EXTX selection as described later when we introduce the HER generator.

The other AXI-Stream master interface (`m_axis_nic_*`) performs a *pass-through* of packets that did not match with any installed rulesets back into the regular Corundum packet data path. This allows the NIC with PsPIN attached to it to still function as a normal NIC when PsPIN is not configured. It also enables host processing of traffic that is not of interest to PsPIN, for example in handling the address resolution protocol (ARP) as described in Section 4.4.5, or when implementing an application-level control plane in the Message Passing Interface (MPI) Datatypes application as described in Section 4.5.4.

4.2.3.2 Packet buffer allocator

The packet buffer allocator takes the metadata from the matching engine and allocates a buffer for the packet in the L2 packet buffer of PsPIN. It runs the allocation algorithm based on the packet length, adds the resulting address of the allocated buffer to the packet metadata, and forwards the metadata to the DMA engine to actually write the packet into the memory. It takes in the *feedback* from PsPIN, which denotes that a packet has been processed and its buffer can be freed, to free the buffer correctly. It further outputs one statistics counter of how many packets have been dropped due to the buffer being full.

The Verilator model originally developed in the PsPIN project uses a software-based ring buffer in the simulation testbench to allocate space for incoming packets in the packet buffer. The free algorithm needs to keep a queue of out-of-order frees and is thus difficult to implement in hardware. However, most packets on the Internet and in

data center environments follow a *bimodal* distribution in size: 40% of packets are below 64 bytes and another 40% are 1500 bytes (the MTU of an Ethernet/IP network) [24, 3]. We thus take a simpler *fixed-size* allocation approach: we partition the packet buffer into two halves; in one half we make fixed 128-byte slots, and in the other half we make 1536-byte slots. We store these free slots in two separate FIFOs. We then handle allocation and free simply by popping from and pushing to the respective FIFOs. This way, we greatly simplify the hardware implementation of the allocator while not sacrificing too much buffer utilisation on internal fragmentation.

4.2.3.3 Ingress DMA

The ingress DMA module takes the allocated address and length in the packet metadata and performs a DMA transaction to write the packet data to the PsPIN NIC inbound memory port. Upon finish of the DMA request, the module forwards the packet metadata on to the HER generator in the data path, such that the packet can get scheduled on the PsPIN cluster. We use the `axi_dma_wr` module from the Corundum AXI IP library to perform the actual DMA operation.

One complication to be handled in this module is that the matching engine could only generate the packet metadata *after* transferring the packet data on the AXI-Stream bus. This is due to a dependency introduced by needing to count the length of the message. While this is handled by introducing a shallow `axis_fifo` to reverse this dependency for the DMA module, it would introduce a per-packet latency of the number of cycles it takes to transmit the packet on the AXI-Stream bus. In addition, the module has to ensure that the DMA transfer to the PsPIN packet buffer is finished before it could issue the HER to the cluster due to the current monolithic design of the PsPIN scheduler.

4.2.3.4 HER generator

Once the packet is written to the right place in the L2 packet buffer of PsPIN, the data path can now schedule the packet for processing by issuing a HER to PsPIN. Part of the information required to generate a HER comes from the packet metadata, such as the message ID and if the packet is the last in a message (*End-Of-Message*, EOM). The rest of the HER stores the address of the handler functions that the packet should be processed with, as well as the host DMA and L2 memory regions. We expose a register control interface to the host through `pspin_ctrl_regs`.

4.2.3.5 Collective ingress data path

`pspin_ingress_datapath` does not provide extra logic by itself, as it is simply an instantiation wrapper of the four data path components. It keeps the parameters in synchronisation among the data path modules and allows for one single place to pass in custom parameters. It also functions as a top module for end-to-end simulation and unit tests so that we can validate that the data path modules have consistent assumptions of how each other operates.

4.2.4 Egress

PsPIN also needs the ability to send packets into the network. This is needed to either complete a protocol by sending back acknowledgements, or transmit packets to other

nodes e.g. to implement in-network AllReduce [8] with PsPIN. The transmission of the prepared egress packet is handled by `pspin_egress_datapath`; we discuss about potential problems in preparing the outgoing packet and solutions in Section 4.4.5.

`pspin_egress_datapath` handles egress commands from PsPIN. With the Corundum IP `axi_dma_rd`, the module performs a DMA read from the packet buffer and gets an AXI-Stream bus that contains packet data. It then injects the AXI-Stream into the outbound AXI Stream of Corundum with an AXI-Stream arbiter (`axis_arb_mux`). The arbiter is wired such that the outgoing traffic from PsPIN has priority over egress traffic from the host for maximum possible throughput from PsPIN. It can also be configured to use round-robin arbitration to ensure fairness between the host and PsPIN on outgoing packets.

4.2.5 Host DMA

A feature that distinguishes the sPIN programming model from other packet processing paradigms intended for intrusion detection (IDS), for example [26], is the ability of packet handlers to read from and write to host memory. Between PsPIN and Corundum, this is enabled through the `pspin_hostmem_dma` module. The module bridges the AXI master port of the PsPIN cluster to the segmented DMA interface of Corundum [14], which takes a RAM port and a separate command bus. We utilize the AXI-Stream DMA client (`dma_client_axis_source`, `dma_client_axis_sink`) from Corundum to convert the output AXI Stream bus to AXI4 channels. For write requests from PsPIN, the module first issues a DMA command to the AXI-Stream client to capture the write data in a dual-port RAM buffer (`dma_psdpram`); it then issues a command to the Corundum DMA subsystem to DMA the data from the buffer RAM to the host memory. The read process happens in the reverse order.

There are some notable limitations in this approach, namely that the adapter is not fully AXI-compliant in multiple corner cases. We do not support irregular bursts (narrow bursts or modes other than `INCR`), as well as interleaved read requests. Unlike AXI4, the PCIe interface also does not support arbitrary *byte enable* (BE) configurations, so we also do not support these cases. While it is theoretically possible to handle all these corner cases, it would lead to very long combinatorial paths of the resulting hardware, which would then take too much engineering effort to fix. However, these limitations are acceptable in our use case, since the DMA bus master in PsPIN does not issue such requests.

One important corner case to implement correctly, however, is *unaligned writes*. As mandated by the sPIN specification and also as we later will see in Section 4.5.4, unaligned transfers are essential to some applications. While it is possible to implement unaligned transfers in software by reading the affected word first to compose and issue an aligned transfer, the extra memory read transactions (up to *two* extra reads for one unaligned write) would significantly hurt performance. Fortunately, the Corundum DMA subsystem fully supports unaligned transfers. As AXI4 expresses unaligned writes as aligned writes with strobe (byte-enable), we implement an *address recovery* procedure that calculates the original address and length from the AXI burst strobe (`WSTRB`) signal of the first and last beat in the AXI transaction. The module then issues the unaligned transfer to the client and Corundum DMA subsystem as normal.

4.2.6 Design Considerations

It is not a goal of this project to achieve the absolute highest possible performance. The hardware implementations are thus designed with the approach of the *simplest* hardware implementation possible. This means that modules with complicated logic e.g. the host DMA engine are simple state machines without pipelining. We also do not support concurrent requests, even if the protocol supports it (in the case of AXI4 on the host DMA engine). For the purpose of a full-system demo, we argue later in Section 4.5.2 that these design limitations would not impact the overall system performance.

Another limitation of the hardware performance is in the PsPIN implementation. PsPIN uses the PULP Ultra Low Power (PULP) [34] RISC-V cores and AXI infrastructure, which are originally designed for ultra-low-power Application-Specific Integrated Circuit (ASIC) platforms. This means that they are optimised for recent ASIC process nodes and thus have long critical paths, making them not suitable for FPGA operation. While some parameter tweaking allowed us to break very long critical paths e.g. single cycle bus across the entire SoC, most components need to be redesigned to reach a higher F_{max} on FPGAs.

The lengthy critical paths of PULP and thus PsPIN on FPGAs mean that without significant re-engineering, the packet processing cluster could only run at a lower frequency. This situation is further worsened by the area requirements of the original PsPIN design: the 4-cluster configuration that was used in the original PsPIN paper proved to be extremely difficult, if possible at all, to place and route on the FPGA device we are using. We thus use a 2-cluster configuration with reduced memory sizes. To further resolve the routing congestion problems, we employ the incremental implementation flow provided by Xilinx as described in Section 4.5.1.

In contrary to PsPIN, Corundum runs at 250 MHz on the target Xilinx devices. While it is possible to retarget Corundum to run at a lower frequeny, we would to have to reconfigure the clock domains and validate that the resulting design still works properly; this is a non-trivial process. Instead, we opted to *only* run the PsPIN cluster and the closely coupled data path engines at a lower frequency (40 MHz for the evaluation in this document; more about the setup in Section 4.5.1). We perform cdc on the AXI-Lite and AXI-Stream interfaces with standard IP blocks from Xilinx. We isolate timing optimisation as a separate task and keep it out of the scope this work due to time constraints of the project.

4.3 Drivers and other Software Components

As described in Section 4.2.1, the hardware design of FPsPIN exposed all slow-path control flows to the host CPU through the `pspin_ctrl_regs`. While this simplified the hardware design by allowing us to omit a dedicated *management core* on the FPGA, the job of configuring the hardware now lands on the host CPU. In addition, we also extended the handler runtime on PsPIN to support the new hardware integration. We explain in this chapter the different software components developed for FPsPIN. Three classes of software are required for the full operation of the hardware: Linux kernel

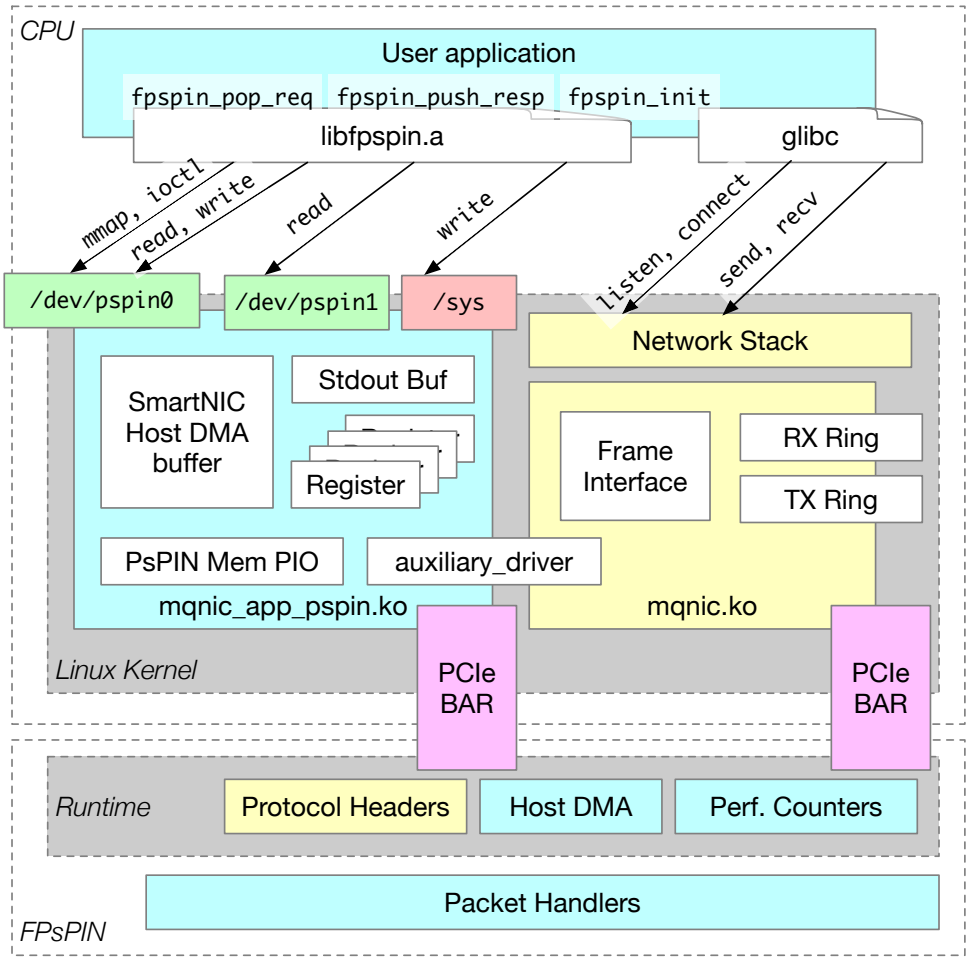


Figure 20: Overview of the software on the host. Yellow blocks denote existing software, while blue boxes show software developed in this project's scope. Note that we use only standard Unix syscalls (`read`, `write`, `mmap`, `ioctl`) between the user- and kernel-space.

modules, user-space library and utilities, and the updated handler runtime. An overview of the software landscape of FPsPIN can be seen in Figure 20.

4.3.1 CPU Kernel Modules

Multiple approaches to access to device memory on Linux exist and most of them require some degree of kernel-level support. One approach is to expose device I/O memory access (in the case of PCIe (PCIe) devices, the base address register (BAR)) to user-space through `/dev/mem` and host memory DMA access through `udmabuf` [19]. While this approach is commonly used when developing FPGA-based accelerators in embedded environments, it introduces severe security risks due to exposing direct physical memory access to the user-space and is thus limited to embedded systems.

The other approach is to have a dedicated kernel module that interfaces with existing subsystems in Linux and does not expose unconstrained physical memory read and write (other than for diagnostics purposes). The API exposed by the device driver kernel module not only greatly reduces the attack surface, but also abstracts away details of

the hardware between different revisions, facilitating development of user applications and support libraries. While writing a dedicated kernel module requires experience with kernel programming, we argue that this is a necessity in hardware development. In addition, the overhead of doing so has already been greatly reduced by Corundum from their application block driver templates. This is the approach adopted by Corundum (`mqnic.ko`) and in turn by FPsPIN (`mqnic_app_pspin.ko`).

4.3.1.1 mqnic.ko

Corundum ships a kernel driver for the complete NIC functionalities, including interactions with the Linux network stack to expose the device as Ethernet NICs for packet transmit and receive, as well as control interfaces with `ethtool` that reports link status. In addition, it also exposes a device file `/dev/mqnicX` for the user-space libraries and utilities to perform management tasks, such as online firmware upgrade and device reset control.

Corundum provides driver support for the custom application block through the *auxiliary bus* framework [2] in Linux. The framework allows splitting drivers for largely independent functionalities on the same device into different device drivers and thus different modules to allow compartmentalisation and separated operation. The main device driver registers an auxiliary *device* while the sub-component driver registers an auxiliary *driver* with the framework. In Corundum, the main driver registers the application block as an auxiliary device and exposes the *application base address* (a separate PCIe BAR) to the auxiliary driver. This allows the custom driver to access the application block BAR to interact with the hardware.

4.3.1.2 mqnic_app_pspin.ko

The driver for FPsPIN configures the PsPIN cluster and additional datapath components after they are brought out of reset. The driver exposes two device nodes, `/dev/pspin{0,1}`, as well as a selection of device registers over `sysfs` [28]. All user-space operations during configuration and normal operation happen through access to these resources using standard *system calls* (syscalls). In addition to normal operation, the kernel module checks for additional requirements imposed by the hardware and rejects requests from the user-space that violates these requirements. We explain the main functionalities of the kernel module in this section.

4.3.1.3 Control registers

The control registers from the hardware are exposed as access to the *application base address* from the Corundum auxiliary device. We use the register generator introduced in Section 4.2.1, `regs-compiler.sh`, to generate the respective sysfs node implementations; the register group and subgroup hierarchies are directly translated into *device attributes*. The generative approach keeps the driver's view of the device registers consistent with actual hardware. We implement consistency checks of data-path engines via internal flags that are kept in sync with the respective enable registers, such that only valid and consistent configurations can be latched into hardware.

By exposing the hardware registers directly to user-space through sysfs, we adopt a *user-space-centric* approach to hardware configuration. This means that most configuration logic will be implemented in a user-space library (Section 4.3.2) instead of directly baked into the kernel module. This allows more flexibility in the implementation,

since we do not need to update the kernel module as often; it acts more as a *shim* that only enforces basic safety and forwards other requests directly to the hardware. This approach also offers more protection against programming errors when implementing the configuration routines, as errors in the user-space cannot crash the kernel.

4.3.1.4 Standard output read-back

Recall that as described in Section 4.2.1, the handler processing unit (hpu)s write their standard output into a FIFO for the host CPU to read for diagnostic purposes. The FPsPIN kernel driver exposes the `/dev/pspin1` character device to the user-space. For simplicity, the raw word sequence read from the hardware FIFO is directly exposed: each 4-byte word encodes one character as well as which hpu wrote this character. A user-space script later introduced in Section 4.3.2 would de-multiplex this stream and write a log file for each hpu.

In the current design and implementation, the standard output device file is *on-demand*, meaning that data will be fetched from the FIFO only when a user-space program reads from the device file. This has the potential issue of the hpus writing too fast to overflow the FIFO, resulting in a partially lost and corrupted output buffer. An alternative design is to run a kernel *worker* (also known as a kernel thread) that continuously polls on the hardware FIFO and actively fetches the standard output data as soon as it is available. However, this would result in a constant overhead for busy polling and wouldn't be ideal if we do not care about the debug output. We thus stick to the current on-demand design.

4.3.1.5 PsPIN memory access

As part of the configuration process, the host needs to download the code and runtime data for the hpus onto NIC memory. As explained in Section 4.2.1, a technicality due to the small Corundum control port address space mandates a static address mapping when accessing PsPIN memory from the host. The kernel module implements this mapping and maintains the plain address view; requests that does not land in a valid memory area will be rejected with a SIGBUS (bus error signal in Linux) to avoid disrupting the hardware. We hide the translation technicality away and never expose the exact mapping details to the user-space.

The kernel module exposes two *flavours* of APIs to the user-space for accessing PsPIN memory, designed for different use cases. The first flavour conforms to the traditional non-buffered Unix file I/O: we implement the `open()`, `seek()`, `read()`, `write()`, and `close()` syscalls on the `/dev/pspin0` character device. Reads and writes to the device file are directly translated into reads and writes in the NIC memory region. This flavour is suitable for bulk read or write on the PsPIN memory area and would be used during program image load or debug memory dumping. It allows existing, unmodified Unix user-space utilities such as `dd` [20] to work as diagnosis tools and quick prototypes.

The second flavour is implemented as `ioctl()` over the `/dev/pspin0` device file. An *ioctl* (input/output control) is a syscall for device-specific I/O operations. The syscall allows the user-space application to pass a pointer to the kernel to read or modify, along with an *ioctl number* to denote the operation desired. We implement two *ioctls*, `PSPIN_HOST_WRITE` and `PSPIN_HOST_READ`, allowing the user-space to read and write 64-bit words in one action. This simplifies the implementation of host DMA and performance counters user-space routines (Section 4.3.2) and reduces the syscall overhead. In

comparison, the traditional Unix file I/O approach would require two separate syscalls (`seek()` and `read()` or `write()`).

4.3.1.6 Host DMA

Memory pages used for DMA on Linux have to be registered with the kernel to ensure that cache coherency and alignment requirements are fulfilled. It is also important to make sure that the memory page used for DMA are not moved by the kernel through swapping or memory compaction (through `kcompactd`). The easiest way to ensure these requirements is to have the kernel module allocate the DMA buffer through the DMA API, which takes care of these requirements automatically. We implement the `mmap()` syscall for the `/dev/pspin0` device to perform a *multi-use* DMA allocation (as opposed to *single-use*; termed as *coherent* by Linux, but does not actually imply cache coherency). We then mark the area as *uncached* and map the allocated DMA memory area into the user application address space to allow user processing of host DMA traffic.

Since we adopt a user-space-centric approach regarding the configuration registers, the user-space needs access to the physical address[4] of the mapped DMA area to write to the control registers. We implement another *ioctl* on `/dev/pspin0`, `PSPIN_HOSTDMA_QUERY`, to allow the user-space to query the physical address of the DMA area, in order to program the EXTX to the data-path engines, specifically the HER generator.

The multi-use DMA buffer allocations we use suit the purpose of a DMA buffer shared between the CPU and device over a rather long period of time. However, in the practice of implementing NIC drivers, the *single-use* allocation scheme is more commonly used and allegedly more performant due to the possibility of taking advantage of the cache. It is possible to take advantage of this approach in FPsPIN by using a separate DMA area per *message*, as opposed to the current strategy of one area per *EXTX*.

4.3.2 CPU User-Space

The user-space software for FPsPIN caters to three distinct purposes in system operation: *configuration* of the system to bring it into operative state; *runtime* that supports the host-side application to interact with the NIC; and several *utilities* to aid system-wide setup as well as to perform troubleshooting. They interact with the various facilities provided by the `mqnic_app_pspin.ko` kernel module. The user-space software shipped with FPsPIN are either packaged into a static library, `libfpspin.a`, along with the header files, or as standalone programs or scripts.

4.3.2.1 Configuration

The main configuration routine is packaged in `libfpspin.a` as one function: `fpspin_init`. It takes as input the device node exposed by the kernel (by default `/dev/pspin0`), the separately-built sPIN handlers image, the ID of the EXTX to use, and a number of rule sets for the matching engine. The user can either select existing rule sets that match against common protocols, e.g. Transmission Control Protocol and the Internet Protocol (TCP/IP) or User Datagram Protocol (UDP) over IP/Ethernet, or define their own rule sets by filling in the `fpspin_ruleset_t` struct that

[4] On a system with an *I/O memory management unit* (IOMMU) enabled, this is actually the *bus* address as seen by the DMA bus masters in the device.

contains configurations for each matching unit (review Section 4.2.3 for more details). `fpspin_init` configures all the device registers over sysfs, loads the sPIN handler image, and also allocates the host DMA area by requesting through `mmap` upon the kernel. It then fills in all necessary addresses and handles in the context variable `fpspin_ctx` and returns this to the user. All future interactions with the runtime takes the context as an argument.

After a successful return of `fpspin_init`, FPsPIN is ready for packet processing. However, in complicated applications e.g. the datatypes demo shown in Section 4.5.4, the user may wish to perform additional initialisation, e.g., loading dynamically generated data into the NIC memory. This is accomplished via the host access to NIC memory interfaces provided in `libfpspin.a`, namely `fpspin_write_memory`, allowing the host to generate the NIC memory content *in* the host application at runtime. The host application needs to take care of *relocation* so that data structures contain valid NIC pointers when they are accessed by the hpu in operation.

While the basic initialisation via `fpspin_init` programs the matching engine as the last step such that no packets can arrive at the cluster until it is fully configured, host-side user initialisation happens after the hpus have started execution. As a result, the user needs to ensure that the sPIN handlers do not start processing packets until the host initialisation process is finished, e.g. through a flag that gates all hpus from running. The exact mechanism and interface requirements are further discussed in Section 4.4.6 as a possible extension to the sPIN specification.

It is important that the host CPU should be able to perform other workloads, such as computational tasks, during packet processing in a truly *offloading* manner. The host application can overlap other workloads via multi-threading or by anticipating the inter-message gap (IPG) and polling only when there could be a message arriving. For simplicity, the current flags-based host DMA notification facility can only hold one in-flight message between the host and each hpu; this limits the duration of overlapped workloads between polling to be one IPG. The overlap can be increased by implementing a proper ring buffer for the notification, which we leave as a possible future improvement.

The host application may still need to receive and send network packets on the same interface, for example to implement the slow, non-performance-critical paths of a network protocol, like connection setup and tear-down in TCP/IP. The intended operation for this purpose is via the host network stack, either normally or through the *raw* sockets (in case of state confusion due to partially offloaded messages). The user needs to correctly configure the matching engine, so that these packets are actually delivered to the host CPU and not to PsPIN. Alternatively, if it is difficult to express the criteria in the matching rules, the user can make the handlers perform a *secondary match* and deliver such packets to the host over host DMA.

Performance measurements are important to estimate bottlenecks of packet processing. The runtime provides facility to read and clear *performance counters* exposed by the handlers. Up to 16 32-bit counters are accessible from the host application via `fpspin_get_counter` and `fpspin_clear_counter`. Each counter keep track of a total sum and iteration count of updates, enabling the calculation of an average value. The counters are updated in the packet handlers using a facility in the handler runtime.

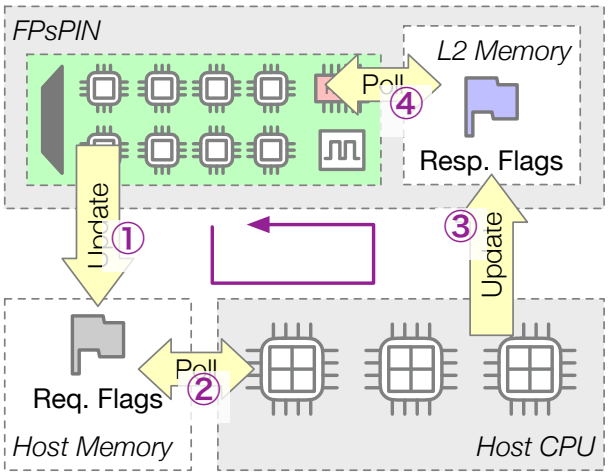


Figure 21: Simplified view of the host DMA loop, in chronological order. The hpu sends a request to the host for processing, by writing to the flag in host memory; the host polls and pops the request from local memory; the host pushes the response by writing to the flag in NIC memory; the hpu polls and pops the response from local memory.

4.3.2.2 Utilities

In addition to the `libfpspin.a` library to be statically linked into the user application, we also provide several standalone utilities that are important to the normal operation of FPsPIN. One of these is `cat_stdout.py` that reads from the log facility, `/dev/pspin1`, provided by the application kernel module. The script performs *blocking read* on the log device and demultiplexes the stream of printed characters according to the core ID. The user can specify whether to dump the log to files and if the script should remove stale logs. The script is provided separately instead of integrated into the runtime, in case of an application that does not care about the debug output from the hpus and thus does not want to waste CPU cycles to read them.

During the development and testing of handlers, it may be necessary to read or write specific memory locations in the NIC L2 memory. The `mem` utility takes a NIC address and performs a 64-bit read or write command over the `ioctl` interface provided by the kernel module. It is possible to implement a more complicated debugger protocol with memory access in this fashion; we leave this as future work.

4.3.3 Handler Runtime

The PsPIN project provided a rather comprehensive implementation of the handler-side sPIN API through the PsPIN/PULP runtime. This includes the HER and task data structures, as well as host DMA commands for the handler code to invoke. A few additions are made to accomodate new abstractions introduced by FPsPIN. One of such additions concerns packet header processing. The existing PsPIN runtime already provides C structs for interpreting headers for IP and UDP, but since FPsPIN directly receives Ethernet frames instead of the IP payloads of a lower-level messaging network layer, we added the Ethernet header structs for this situation. We also implement support for the SLMP, introduced later in Section 4.4.1, in the same manner in the FPsPIN runtime.

4.3.3.1 Performance measurements

The host-side runtime has support for reading back performance counters generated by the handler routines; these counters are updated through the handler runtime on the hpu. Each 64-bit counter consists of two 32-bit fields, the *sum* and *count*, allowing the handler logic to push a specific performance value into the counter with the `push_counter` routine. Every time the counter is incremented, the count field is incremented by one. The counters sit in L2 memory accessible to the host and are initialised to zero on cluster setup. These counters can be used to collect various statistics such as handler execution time at handler or message granularity, as well as to profile specific code areas in the handler routines.

The counter values for performance measurements are derived from the *cycles* register in PULP that is only accessible to *machine-mode*, while the handler code executes in user-mode. We extend the existing fault handler in the PsPIN runtime to handle syscalls and implement a syscall to read the cycles register. With the current naïve implementation, this syscall path takes around 100 cycles on the PULP cores, in which most of the time consumed comes from the need to save and restore all general purpose registers. While it is possible to optimise this path for a lower latency, a proper solution is to address the lack of a user-accessible time register for precise performance measurements as we will argue in Section 4.4.3. We leave the related changes to hardware as future work.

4.4 Proposed Changes to the sPIN — Lessons Learned

The process of building FPsPIN has uncovered various aspects in the sPIN specification that are important in a real-world system but left unspecified. We have reported the findings in this chapter to the sPIN team and some of them have already been incorporated back into the specification.

4.4.1 Messaging and Reliability Layer: SLMP

The sPIN specification did not impose a fixed list of underlying network protocols; instead, it specifies two *matching modes* of the underlying network. In *packet matching*, single packets are matched for processing on the packet handler in the same flow; Ethernet would be an example of a network operating in this mode. *Message matching*, on the other hand, requires the network to provide an abstraction of messages as a stream of multiple packets; they are in turn mapped onto the *head*, *packet*, and *tail* handlers for processing. Examples of a network operating in message matching mode are Remote Direct Memory Access (RDMA)-style networks such as InfiniBand or the Intel *Omni-Path Architecture* (OPA).

Although FPsPIN is built on Ethernet, which would seemingly force a *packet matching* implementation, many applications still benefit from the message-oriented abstraction sPIN offers; there would be significant hpu and memory overhead to perform flow matching and differentiate the handler code paths in software, as opposed to the MPQ-based hardware flow matching mechanism introduced in PsPIN. As a result, it is desirable to *emulate* the message abstraction on top of Ethernet. In addition, since

Ethernet is a *lossy* network[5] but traditional high-performance network applications such as MPI expect that messages do not get lost in the network, we also need guarantee on reliable delivery of messages on top.

We developed a thin messaging and reliability layer built on top of UDP/IP for FPsPIN, which we named sPIN lightweight messaging protocol (SLMP). The protocol features a 10-byte header inside the UDP payload with the following field definition:

Flags *(2 bytes)* hosts three packet-level status bits, *syn*, *ack*, and *eom*. The remaining bits are reserved for future versions of the protocol.

Message ID *(4 bytes)* unique ID of the message.

Offset *(4 bytes)* byte offset of the first payload byte in the message.

The message ID and offset fields together implement up to a message size of 4 GB, limited by the 4-byte offset field in the SLMP packet header. We believe that this is a sensible limitation and most applications would not generate messages larger; for applications that require larger messages, we could adjust the size of the offset field in a case-by-case fashion. Although the offset field maintains a order among all the segments of a message, we do not implement any in-order delivery guarantees.

A possible alternative for the offset field is to use a packet sequence number instead of a byte offset. This is a design choice made to accomodate the SLMP file transfer and MPI datatypes applications we will introduce in Section 4.5.1; these applications process incoming segments according to their byte offset in the whole stream. Therefore, by storing the offset number directly in the SLMP header, we eliminate the need to keep protocol states on the receiver, allowing a fully stateless receiver for handling the protocol.

We handle the reliability requirement through the syn and ack bits in the flags field in the SLMP header. The action rule for the receiver is simple: each packet that has a syn bit set in the header needs to be ack'ed by sending back the same header with no payload. The sender decides on what reliability mode the protocol operates in. For no guarantee at all, the sender omits the syn bit for all packets. For a guarantee of message delivery but not individual segments, the sender sets syn on the first and last packets of the message. For a guarantee of every single segment, the sender sets syn on all packets it transmits. We do not implement retransmission for SLMP at the moment for simplicity, but it should be easy to add since our acks carry the message ID and segment offset and thus would allow the sender to identify a lost segment.

4.4.2 SLMP flow control

If the sender would transmit packets too fast to the receiver, the receiver would be overwhelmed by incoming packets before it had time to process them; packets would be dropped once the receive buffer is completely filled. *Flow control* throttles the sender to make sure that the receiver is not overwhelmed. Generally speaking for maximum

[5]While RDMA over converged Ethernet (ROCE) does provide a lossless guarantee on top of Ethernet, it is not supported by Corundum at the time of this project and is not trivial to implement in hardware. We thus consider ROCE irrelevant for this discussion.

throughput, the sender needs to first fill up the receiver's buffer at a higher send rate, then lower the rate to the processing speed of the receiver to maintain the occupation rate of the receiver buffer in order to saturate the receiver's processing power.

There are two modes of flow control in SLMP, depending on the reliability configuration selected. When the sender operates without reliable packet delivery in the form of acks from the receiver, a heuristic inter-packet gap (IPG) is chosen based on the workload, receiver's capability, as well as possible buffer state reports from the receiver (see Section 4.4.3 for more details). This form of flow control requires almost no collaboration from the receiver and can be implemented fully on the sender side. However, a practical configuration would fix the IPG and thus the sending rate, which must be equal to or lower than the receiver processing speed for long term stable operation; this would under-utilises the receive buffer and result in the receiver waiting for data, hurting throughput.

A well-known mechanism that originated from TCP/IP but found its way into most flow control schemes is a *flow control window*[6]. The sender maintains a fixed window of packets that has not yet been ack'ed by the receiver and would only send when the window has free space to fit a new packet, allowing the sender to automatically throttle down to the speed of the receiver after the window is filled. Assuming constant sender and receiver speed v_{Send} and v_{Recv}, we can calculate the ideal window size S_{Wnd} for a receiver buffer size S_{Recv}:

$$t_{\text{Fill}} = \frac{S_{\text{Recv}}}{v_{\text{Send}} - v_{\text{Recv}}} \tag{1}$$

$$S_{\text{Wnd}} = t_{\text{Fill}} \cdot v_{\text{Send}} \tag{2}$$

$$= S_{\text{Recv}} \cdot \frac{v_{\text{Send}}}{v_{\text{Send}} - v_{\text{Recv}}} \tag{3}$$

The sender window approach to flow control still requires the window size to be determined in some manner. For simplicity in implementation, we assume the suitable window size is relatively fixed for each SLMP conversation and let the user specify the window size manually.

An interesting side-effect of allowing explicit control of the flow control window size is that, by setting the window size to 1 packet, the sender can serialise packet processing on the receiver side, only sending the next packet of the message after receiving ack for the previous one. While doing this severely limits the top throughput available, the serialisation guarantee is important since sPIN did not specify any concurrency control mechanism on the scheduler level (discussed later in Section 4.4.4). We use this method to avoid *packet-level parallelism* for the MPI Datatypes demo application in Section 4.5.4.

4.4.3 Telemetry

The ability to measure the performance of a system is crucial to further improve it. While the current sPIN specification formalised performance-related events to be delivered to

[6]This is called the *sliding window* in TCP/IP due to the additional in-order delivery requirement.

the host for further analysis, this is vastly different from the common practice of implementing *performance counters* for detailed system-level inspection. Therefore, we call for a general interface of host access to various implementation-defined performance counters in hardware, as well as user-controllable counters updated from the handler software.

We integrated a prototype of memory-backed general purpose counters for packet handlers in Section 4.3.2 to allow intrusive inspection of handler performance in the ping-pong demo in Section 4.5.3. For low-overhead updates to these user-controllable counters, an implementation should make accurate cycle counters available to user-level handler code. While we recognise the exposure of accurate timing information is a break of isolation, the current sPIN specification does not handle multi-tenancy yet; we leave this discussion for a possible future work that would explore operating system paradigms on sPIN.

Another compelling use case for telemetry data apart from off-line host performance analysis is for consuming in the sPIN NIC itself, better known as *introspection*. One use case would be for the scheduler to have access to the handler execution time counter for fair scheduling between multiple EXTXs.

4.4.4 Scheduler Concurrency Control

The current sPIN scheduler enforces the dependency between packets in a message w.r.t. the three categories of packet handlers: the *head* handler is guaranteed to be scheduled before all *packet* handlers, and the *tail* handler is guaranteed to be scheduled after them; the packet handlers will be scheduled in parallel if possible. In some use cases however, the packet processing routine may require serial execution; the MPI Datatypes demo application we will introduce in Section 4.5.4 is an example.

With the current sPIN specification, the only viable synchronisation primitive is *spinlocks* that would allow one hpu into the critical section for serial processing. This is far from ideal, since without an effective task switching method, all other hpus will busy-loop at the spinlock, effectively reducing the number of hpus down to one. A workaround currently used by the MPI Datatypes demo in Section 4.5.4 is to force the SLMP flow control window to 1 packet, effectively requiring ack on every packet and thus serialising packet processing. Such a workaround is still not ideal, since such a small window size would mean that the hpu would always have to idle for one rtt plus the sender latency for one packet before it can start processing the next packet. It also forces the developer to put the per-message state in the *globally-shared* L2 memory, since there is no guarantee which hpu the next packet will be scheduled to, making it impossible to use the *cluster-local* L1 memory.

We propose the addition of *core masks* to all sPIN EXTXs for fine-grained control on the locality and parallelism of sPIN handlers. With the current FPsPIN architecture, the core masks are installed into the HER generator and copied into the HER to the scheduler. The scheduler can then schedule the packet on the subset of cores specified by the core mask in the HER. This design can support the use case of serially scheduled handlers by programming a mask that contains one single core; *message-level parallelism* can be achieved by installing multiple otherwise identical EXTXs that have different core masks. Another possible use case is to specify a core

mask of all cores in one cluster to allow the shared states to be stored completely in the cluster-local L1 memory.

4.4.5 Network-layer Protocol Handling

So far, sPIN assumes that the packet handlers only process the application payloads carried by the underlying network protocol. However, as we have shown in Section 4.4.1, extra protocol-layer processing is required for a lossy underlying network like Ethernet. While in some situations protocol handling itself is the main offloaded workload (e.g. accelerating QUIC), we recognise that most of the sPIN applications care more about the actual payload instead of details in the protocol itself. Examples of such protocol-level handling include running ARP for outgoing packets, handling connection setup and teardown in TCP/IP, and sending the explicit buffer state messages in SLMP.

One approach to this issue makes use of the *bypass* feature of the matching engine. The user can configure the matching engine to pass incoming protocol control packets to the host for handling, possibly updating relevant states in the meantime. Examples where this approach would work include ARP responses, in which the host updates the IP neighbour table and sends back the ARP response, and TCP/IP connection setups and teardowns, where the host handles packets that have the syn or FIN bit set. This approach would not work very well for NIC-initiated actions (e.g. an active ARP *query* for an outgoing packet from the sPIN NIC) as well as protocol messages on the hot path (e.g. the buffer state notification of SLMP).

An alternative approach is to introduce a dedicated system-level coprocessor for handling such protocol requests. This coprocessor would handle network-layer control-path tasks, freeing the hpus from these. It would take requests from the hpus through a mailbox-like interface, e.g. to run an ARP query or setup a SLMP or TCP/IP connection with a remote endpoint. It could also run local periodic tasks such as sending back telemetry data to its link partner, or take action on specific protocol control messages forwarded from the scheduler. The coprocessor would also be crucial for potential *flow spilling* to the host when the sPIN NIC is overloaded.

4.4.6 Handler Initialisation

Complicated applications may require dynamic initialisation of application-level states on the NIC memory. However, the current sPIN specification would start scheduling packets to EXTXs as soon as they are installed, resulting in a race condition. However, since the installation of an EXTX on the host would also allocate NIC memory windows and set up memory protection, it is required before the host can actually perform initialisation.

A potential solution is to separate the *installation* and *activation* of EXTXs. The installation of an EXTX would arm all relevant hardware modules, allocate memory windows, and set up memory protection; the activation actually enables the respective matching rule on the matching engine for packets to arrive and get scheduled. The actual initialisation can happen on either the host (through NIC memory access) or as a special function in the handler image to run on a hpu.

4.4.7 Host-side Activation

So far sPIN has only been formalised for offloading packet processing tasks on a receiver; although [10] contemplated a possible sPIN-Out implementation that would allow the host to offload sending tasks e.g. in *AllReduce* to a sPIN NIC, the specification did not formalise on how such an EXTX would be defined. We propose *host-activated* EXTXs that are triggered via a special host-initiated event. With the FPsPIN architecture, this can be implemented via adding extra configuration registers to the HER generator to inject a HER whenever the host wants to invoke the EXTX. To avoid confusion, such an EXTX should always have the corresponding matching rule set to *never* such that it never gets invoked from incoming packets.

4.4.8 Alternative Host DMA Interface

sPIN specifies the host DMA facility in a very simple manner: the hpus can read from or write to the exposed flat memory window. While this abstraction is concise to implement and allows , it is not very helpful for end users of a sPIN NIC; they would have to implement their own interface on top of this facility. The fact that sPIN does not specify any memory consistency semantics on the host memory window makes any user implementations non-portable and thus impossible to work with other sPIN implementations.

We developed a simple request/response interface on top of the host DMA window in FPsPIN as we described in Section 4.3.2; the current design largely resembles a traditional *queue pair* design, allowing the hpus to post requests in the receive queue to the host and the CPU to post responses back to the hpus in the completion queue. We propose the addition of higher-level APIs for communication between the host and sPIN NIC, which would simplify user programs that make use of host DMA and improve portability.

4.5 Evaluation

In this chapter, we evaluate the software and hardware implementation of the proposed FPsPIN platform. We first describe the platform we implemented FPsPIN on. We then present an analysis of the hardware components proposed in Section 4.2 to identify potential bottlenecks in the hardware design and implementation. Finally, we demonstrate the overall functionality and performance of the system through several demo applications.

4.5.1 Experiment Setup

The experiments are done on the AMD server with the Ryzen 7 2700 CPU and the PCIe-attached Xilinx VCU1525[7] Development Kit. We run the FPGA board at 16 lanes of PCIe 3.0 clocked at 8 GT/s. A diagram of the experiment platform is shown in Figure 22. Corundum runs at its native frequency of 250 MHz on the Virtex UltraScale+

[7] `https://www.xilinx.com/products/boards-and-kits/vcu1525-a.html`

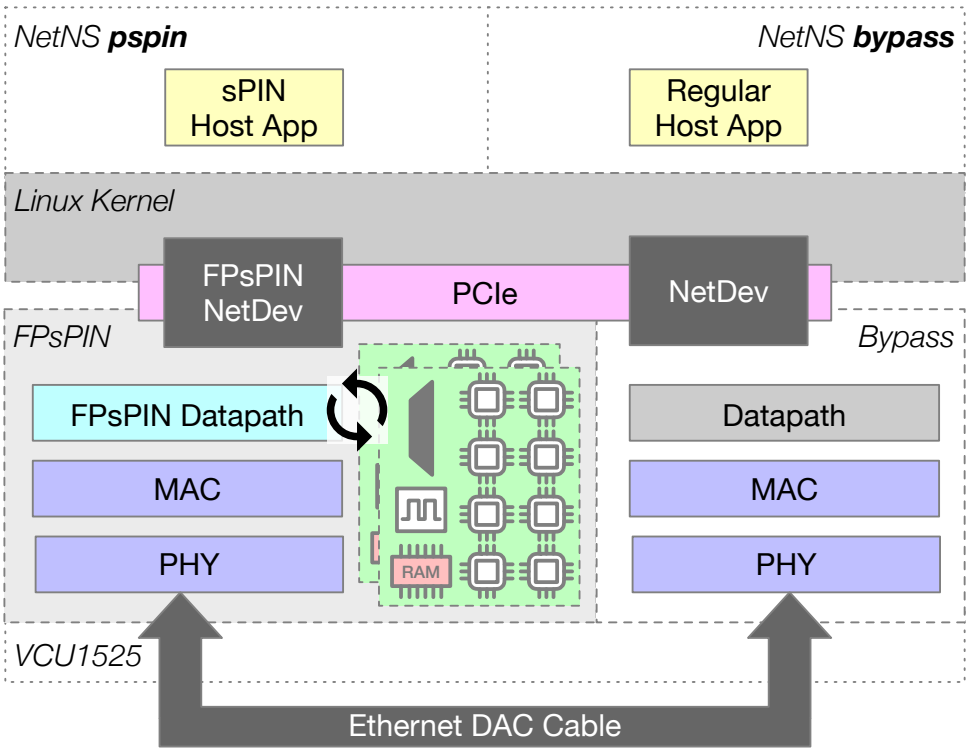


Figure 22: The experiment setup. The sPIN and non-sPIN host applications are in two separate network namespaces to prevent the direct loop-back mechanism in Linux that prevents packets from actually going through FPsPIN.

FPGA. However, we could only run the application block (FPsPIN) at 40 MHz due to the PsPIN IP not being designed for FPGAs: the PULP RISC-V cores are designed for an advanced ASIC process node and have long critical paths on FPGAs. As a result, we clock the processing cluster at 40 MHz.

4.5.1.1 Networking

The two 100 Gbps QSFP Ethernet ports on the FPGA board are attached via one direct-attached copper (DAC) cable, forming a loop-back between the two interfaces of Corundum. Since the two interfaces are present on the same Linux host, we have to isolate the two network interfaces into separate *network namespaces* to avoid a direct loop-back in software. The exact network topology of the system can be seen in Figure 22. A script, `setup-netns.sh`, automates the creation and tear-down of network namespaces and assignment of the interfaces to them.

4.5.1.2 Toolchain

We use Ubuntu 20.04.4 LTS on the host with a slightly modified Linux 5.15.0-76-generic kernel with the CMA enabled; this allows the FPsPIN kernel driver to allocate arbitrarily large contiguous DMA areas, as is required by demo applications shown later in Section 4.5.3. We use Xilinx Vivado 2020.2 to produce the FPGA bitstream, the PULP RISC-V toolchain[8] to compile the sPIN handlers, and the Ubuntu system GCC for the host-side applications.

[8] `https://github.com/pulp-platform/pulp-riscv-gnu-toolchain`

Module	Cycles	Frequency (MHz)	Latency (ns)
Matching engine	4	40	100
Allocator	0	40	0
Ingress DMA	8-70	40	200-1750
HER generator	0	40	0
Host DMA	*n/a*	250	450

Table 4: Latency estimation for various data path modules in cycles and nanoseconds. Note that as the host DMA goes over PCIe to the host DRAM, the exact latency in cycles is difficult to estimate; the latency in nanoseconds is measured on real hardware via the Integrated Logic Analyzer (ILA) on Xilinx platforms.

4.5.2 Design Analysis

To evaluate the implementation quality of the newly introduced data-path components, we estimate the theoretical latency of these components as described in Section 4.2 based on the RTL source. Table 4 shows the latency in cycles based on the state machine construction in the Verilog RTL code, the frequency, and the latency time in nanoseconds. The `pspin_ingress_dma` module has a latency linearly related to the packet size due to dependency requirements as introduced in Section 4.2.3. We show in the later sections that these latency numbers are negligible compared to other parts of the system and thus would not have a big impact on overall system performance.

Resource utilisation and timing are very important static insights into FPGA designs. While we have trimmed the original PsPIN design significantly compared to the standard configuration [11] as shown in Table 5, the design is still very hard to close timing due to congestion issues. We present in Table 6 data in resource utilisation, timing, and time taken to implement the design. To ensure that we get acceptable implementation results for each run, we employ the *incremental implementation flow* [21] from Xilinx to have the Electronic Design Automation (EDA) tool try to reuse routed nets from previous valid implementation runs. This shortens implementation time and improves the general Quality of Results (QOR) of the resulting design.

We present three e2e demo applications to showcase the real-world programmability and performance of FPsPIN. We show that it is possible to write packet-processing applications for the platform. We further characterise the performance of the platform in detail with the MPI Datatypes demo.

4.5.3 Ping-pong

4.5.3.1 Motivation We demonstrate the overall system functionality with two classic types of *ping-pong* protocols: Internet Control Message Protocol (ICMP) and UDP. With this demo, we exercise the various data-path and control-path components newly introduced in FPsPIN to show their basic functionality. In addition, we evaluate the system e2e rtt under simple packet processing workloads to compare with pure-CPU processing. We further identify rtt contributions from different actors in order to evaluate bottlenecks in the system.

Resource Category	[11]	**FPsPIN**
Clusters	4	2
#MPQ	256	16
L1 Cluster Memory	1 MiB	256 KiB
L2 Program Memory	32 KiB	32 KiB
L2 Packet Memory	4 MiB	512 KiB
L2 Handler Memory	4 MiB	1 MiB
L2 SRAM Latency (cycles)	1	1

Table 5:]
Comparison between the stock PsPIN configuration and that used in FPsPIN. The MPQ enables parallel in-flight messages; by reducing the number of queues available, we limit the number of concurrent in-flight messages to 16.

QOR Metric	Value	
lookup table (LUT)	645k	54.5%
Flip-Flop (Flip-Flop)	490k	20.7%
Block-RAM (BRAM)	1141	52.8%
Ultra RAM (URAM)	206	21.5%
worst negative slack (WNS) (ns)	-0.057	
total negative slack (TNS) (ns)	-9.945	
Impl. Time	6:11:15	

Table 6: QOR metrics of the hardware implementation of FPsPIN. The first four entries (LUT, Flip-Flop, BRAM, URAM denote key resource consumption and the percentage utilisation value on the VU9P device; WNS and TNS measure how much the design has failed timing.

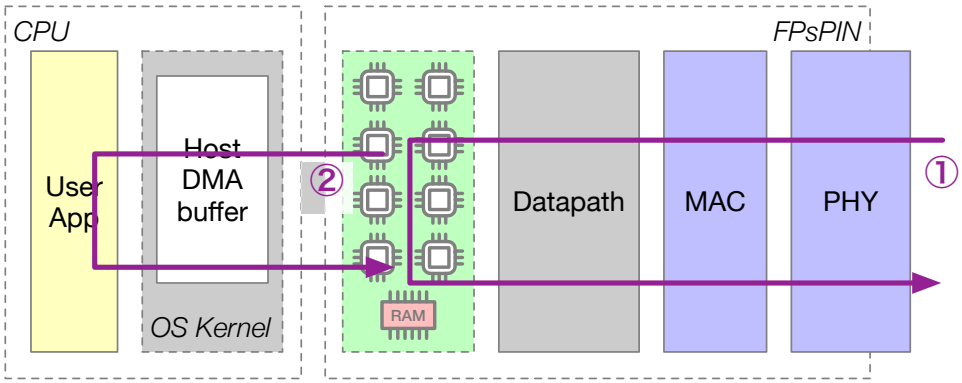


Figure 23: Workflow of the two ping-pong applications. 1) Normal operation; the cluster processes incoming ping requests and sends back responses (**FPsPIN**). 2) The cluster can optionally choose to forward data to the host application for further processing (**Host+FPsPIN**).

4.5.3.2 Experiment

We implement on FPsPIN the server to respond to client requests; the operation flow of the server is shown in Figure 23. Both protocols operate in the same way that the client sends a request packet and the server sends back a response. The server needs to swap the source and destination addresses in the Ethernet, IP, and UDP headers, and recalculate relevant checksums. For each protocol, we implement three different modes of operation:

- the *baseline* a.k.a. host-only case (**Host**): all processing on the CPU by setting the FPsPIN matching engine to bypass mode;
- the *FPsPIN-only* case (**FPsPIN**): FPsPIN does all the packet processing (header processing and checksum calculation);
- the *combined* case (**Host+FPsPIN**): FPsPIN swaps the addresses in the headers and the host CPU calculates checksums.

We use the `ping` utility from *iputils* [22] for ICMP and `dgping` from the *stping* suite [25] for UDP. For **Host** mode, we use the responder in the Linux kernel for ICMP and the user-space `dgpingd` for UDP. For both the **FPsPIN** and **Host+FPsPIN** modes, we use the same naive IP checksum algorithm implementation.

An important difference between the UDP and ICMP ping protocols is that ICMP requires the *entire payload* to be included in the calculation of the checksum field, while UDP only specifies an *optional* checksum of the *UDP header*; we omit this header checksum in our UDP ping server implementation on FPsPIN. This difference between the two protocols impacts both the server and client implementation, but is especially significant for the server since the rtt measurements taken at the client do not include packet preparation and checksum validation time on the client side.

For both ICMP and UDP and the three modes of operation, we measure the e2e rtt of the ping-pong process from the client by running the ping program 20 times, taking 100 measurements in each iteration. This would take into consideration any possible interference between the ping client and server that would result in variance in the

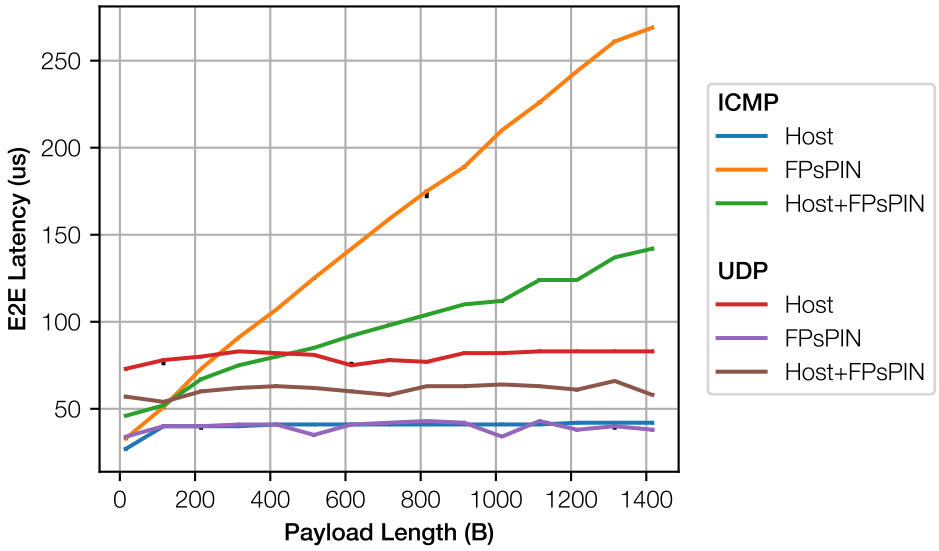


Figure 24: e2e rtt of both protocols across the three setups. We plot the median and 95% confidence interval of the rtt of 20 measurements for each configuration.

measurement, as well as to allow caches to warm up. We plot this e2e rtt with their medians and 95% confidence intervals calculated with the bootstrap method [12] in Figure 24. In addition in cases that involve FPsPIN, we measure cycle counts in the handler code to time the mean handler execution time and latency of host processing. We plot a breakdown of the e2e rtt in these cases in Figure 25. Please note that the high overhead designated as *Syscall* is due to the lack of a cycle-count register accessible to user-space handler code; we discussed this situation and possible solutions in Section 4.3.2 and Section 4.4.3.

4.5.3.3 Data interpretation

A key observation we make is that both **FPsPIN** and **Host+FPsPIN** performed significantly better than **Host** for UDP, with a largest latency advantage of *50 µs* in **FPsPIN** mode. This is mainly due to the packet data in FPsPIN modes not having to go through PCIe to get DMA'ed to host memory, go through the Linux UDP network stack, and context switch to user-mode to reach the UDP responder. The ICMP responder in **Host**, in comparison, runs in the Linux kernel and thus does not have the overhead from the full UDP stack and context-switch to user-mode. This overhead can be confirmed with a comparison between UDP and ICMP in **Host** mode, showing a difference of 41 us. As we see in Figure 25 in **Host+FPsPIN** for UDP, our system reliably achieves a 20 us rtt advantage over the baseline case even with the added latency from host processing. In addition, we notice that in all four cases with FPsPIN the *syscall* category occupies 10-20 us in the rtt. As we have previously explained in Section 4.3.3, this added latency is due to a lack of user-mode accessible cycle counters and should be easily fixed in future work.

We notice a big divergence in the course of e2e rtt w.r.t. payload size between ICMP and UDP in Figure 24: in the two modes that involve FPsPIN for ICMP, the rtt increases almost linearly with the payload size; while in the **Host** mode for ICMP as

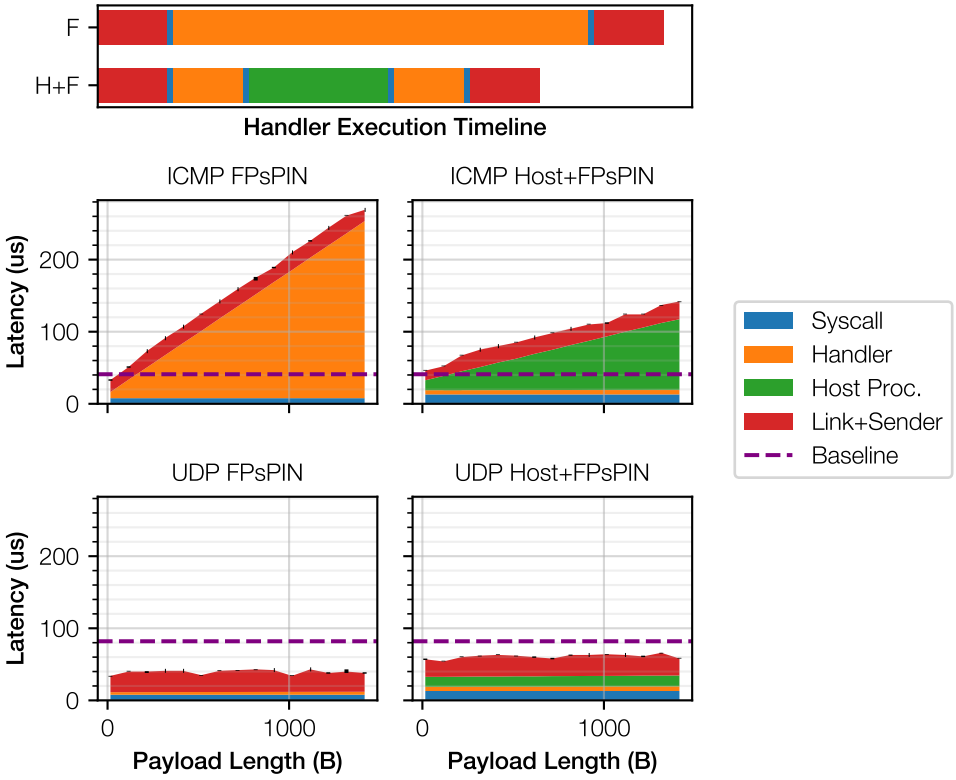


Figure 25: Breakdown of the e2e rtt into different categories. The timeline at the top shows the time-series relationship between the various time components (not to scale): *Syscall*, time spent reading the cycles counter from trapping into M-mode; *Handler*, time spent executing the packet handlers, excluding waiting for host; *Host Proc.*, time spent waiting for host DMA and processing on the CPU; and *Sender*, time spent on the Ethernet wire and client side. *Baseline* marks the median e2e rtt in **Host** mode.

well as all three modes for UDP, the rtt remains relatively constant. This reflects the difference in checksum calculation between the ICMP and UDP ping protocols, showing that checksum calculation time is a significant component of the ICMP response rtt. A comparison between the **Host+FPsPIN** and **Host** modes for ICMP in Figure 25 reveals that the Linux kernel's ICMP responder uses extremely optimised code paths, highlighting room of improvement in our IP checksum algorithm.

We further observe that the lower frequency that PULP cores in FPsPIN run at have a significant impact on packet processing latency. This is confirmed by Figure 25 between *Handler* on FPsPIN and *Host Proc.* on Host+FPsPIN for ICMP. It shows that a single core on FPsPIN is *only* 2.8x slower than a single CPU core, a gap way smaller than the actual performance difference between these cores (40 MHz vs 3.4 GHz). Part of this small performance gap comes from the fact that the host CPU performance in checksum calculation is far from ideal due to the CPU always issuing uncached requests to the host DRAM as a result of a lack of cache-coherency over PCIe. In addition, the host processing category also includes one PCIe rtt between the host CPU and FPsPIN and polling latency, both of which are not present on FPsPIN.

4.5.3.4 Conclusion

The rtt advantage from FPsPIN against the CPU-only **Host** case shows that FPsPIN allows packet processing with lower latency, thanks to its proximity to the data and lack of context switch overheads. The ICMP cases show that FPsPIN still has plenty of potential for higher performance in packet processing from a faster core built for FPGAs, optimised code that reduces handler execution time, and domain-specific accelerators for compute-heavy workloads like checksum calculation.

4.5.4 MPI datatypes

4.5.4.1 Motivation

Apart from synthetic benchmarks like the ping-pong demo we showed in the previous section, we also need to demonstrate the ability of FPsPIN to run real-world sPIN workloads. MPI Datatypes are a popular mechanism for exchanging custom messages over the MPI paradigm commonly used in parallel computing. On the sender side, the datatypes subsystem in MPI *serialises* the custom message with non-contiguous memory blocks (in other words, *holes* in between) into a contiguous *streaming* buffer for transmission on the network; on the receiver side, MPI *deserialises* the contiguous message back into the non-contiguous messages for the user application.

Previous work on sPIN has ported the *MPICH dataloop*-based single-threaded implementation of MPI Datatypes to sPIN handlers that run on a simulator-based platform [10]. By porting these existing sPIN handlers to FPsPIN, we characterise the throughput of sPIN workloads on FPsPIN with different levels of handler complexity on the platform. In addition, we showcase the ability of FPsPIN to achieve almost perfect *computation/communication overlap* with a compute-heavy CPU workload that runs simultaneously.

4.5.4.2 Experiment

We port the handlers in [10] to the FPsPIN platform. A major difference between the original target platform and FPsPIN is the underlying network

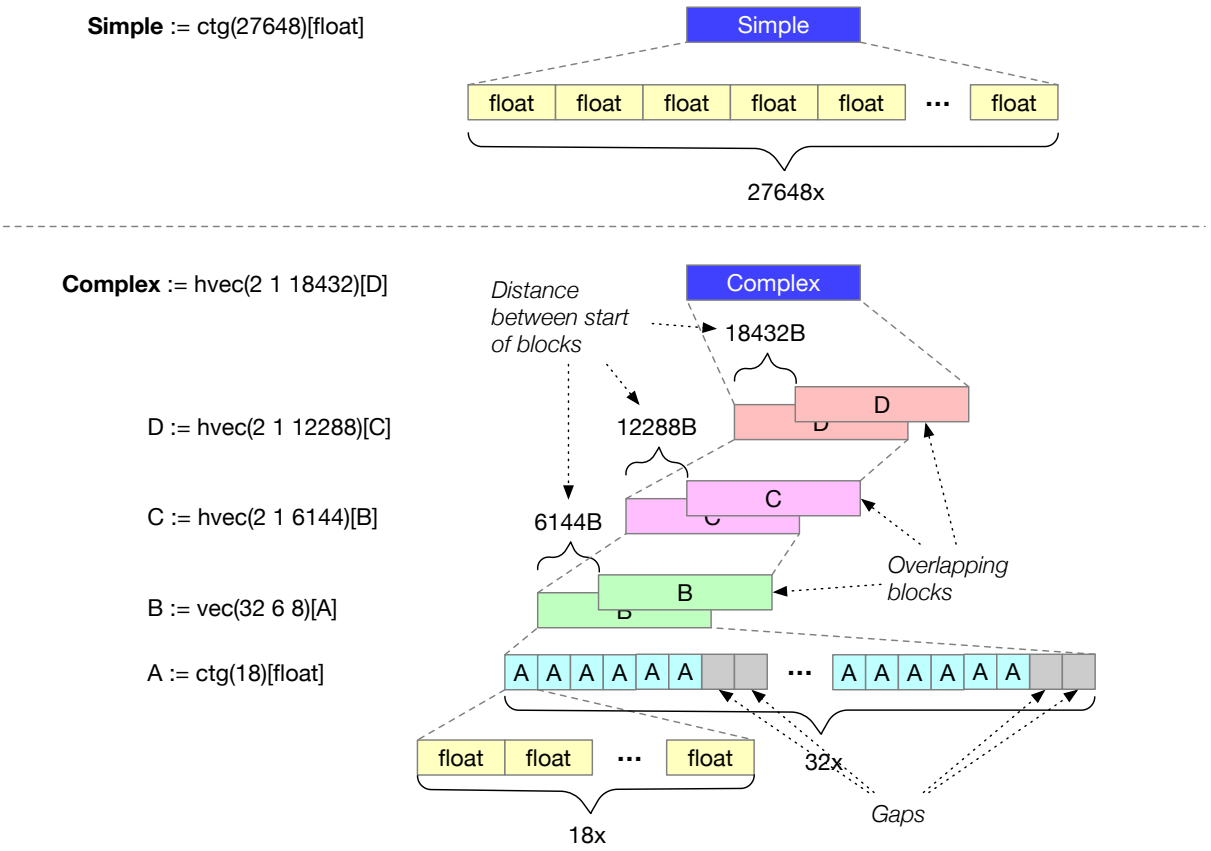


Figure 26: Structure of the two datatypes, **Simple** and **Complex**, used for evaluating the datatypes handlers; each layer builds an intermediate nested datatype until we get the final type. `ctg`: contiguous vector; `vec`: vector with stride in elements; `hvec`: vector with stride in bytes.

layer: the handlers were designed for InfiniBand-style networks that offer a *reliable message transport* with arbitrary length support, while FPsPIN runs on top of lossy Ethernet;

Due to the single-threaded nature of the *dataloop* implementation of MPI Datatypes, we are unable to implement *packet-level parallelism* and are thus forced to use a sender window size of 1 packet to ensure serialised packet processing. To make use of all the 16 hpus on FPsPIN, we implement *message-level parallelism*, sending multiple messages in parallel. The handler function stores all per-message states in the shared L2 handler memory. We evaluate the e2e bandwidth of two different datatype handlers on FPsPIN in comparison to the reference MPICH datatypes implementation on CPU with varying degrees of parallelism in Figure 27. The structures of the two datatype workloads, denoted as **Simple** and **Complex**, are shown in Figure 26.

To demonstrate the *computation/communication overlap* capability of FPsPIN, we run double-precision gemm from OpenBLAS [39] on the CPU to simulate a compute-heavy workload that runs simultaneously with the datatypes deserialisation on FPsPIN. Since the CPU fetches notifications from FPsPIN in poll mode (Section 4.3.2), it is the best to poll as few times as possible to avoid wasting CPU cycles that could otherwise be running the computation workload. For a meaningful evaluation of computation/com-

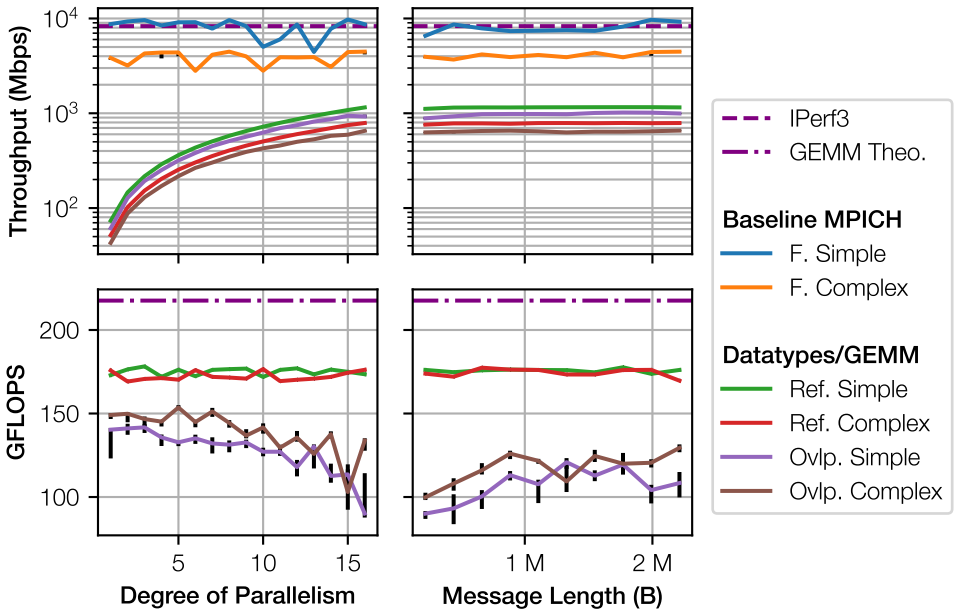


Figure 27: Comparison of e2e datatypes and the GEMM (gemm) throughput in diverse parallelism and message length setups. *Ref.* denotes the reference case where the workload is not overlapped with the other; *Ovlp.* denotes the overlapped case. We also plot the throughput of the MPICH reference CPU implementation as baseline for comparison. We plot the mean value of 20 measurements in each setup; the error bars in black show the 95% confidence interval.

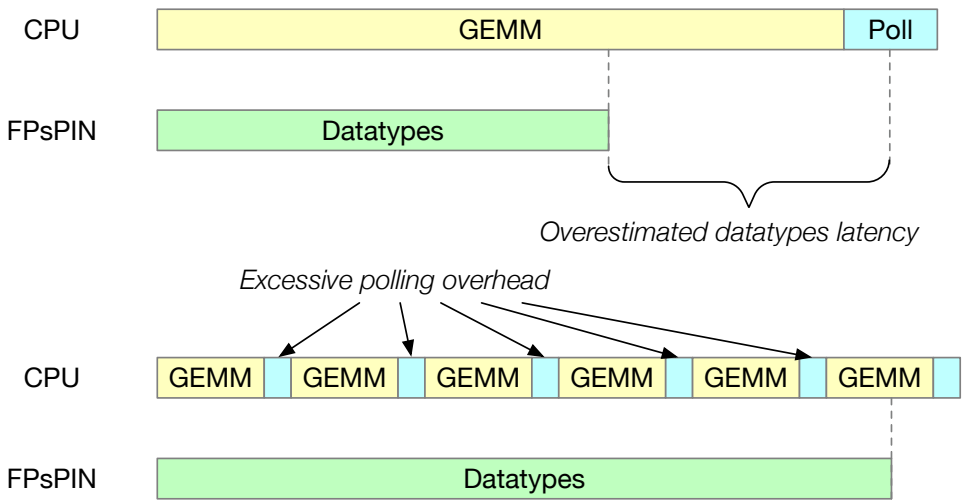


Figure 28: Two possible situations of overlap between gemm and datatypes processing. The top case overestimates the latency of datatypes processing due to not polling at a high-enough frequency; the bottom case imposes excessive overhead on the gemm workload due to polling too frequently.

munication overlap, we *tune* the gemm size for a *balanced* setup between computation and communication, i.e. to minimise both datatypes latency and polling overhead. We show in Figure 28 the two opposites of polling frequency, both of which hurts the performance of either the datatypes or gemm. By tuning the size of a single invocation of gemm, we find the sweet spot to balance between these two situations. Following [17], we define the *overlap ratio* as follows:

$$r_{\text{Overlap}} = \frac{T_{\text{GEMM}}}{T_{\text{GEMM}} + T_{\text{Poll}}}$$

We plot the overlap ratio and polling overhead from two datatypes in Figure 29.

In order to have correlated timing measurements between datatypes processing and gemm, we measure the time elapsed for both workloads on the receiver side. Since the host application does not get a notification until the datatypes transfer is finished, we introduce a Ready-to-send (RTS) signal from the receiver to the sender: the receiver application sends RTS to the sender and starts the gemm workload on CPU. We take the time between the notification from FPsPIN and the RTS as the elapsed time for the datatypes workload.

4.5.4.3 Data interpretation

We report the peak throughput of the two datatypes under different message-level paralellism and message length setups as we have shown in Figure 27. The highest throughput are achieved at:

- **Simple**: **1162.4 Mbps** with message length of 1944 kB and 16 hpus
- **Complex**: **801.2 Mbps** with message length of 1080 kB and 16 hpus

The throughput gain per extra hpu utilised remains relatively constant at around 73 Mbps for **Simple** and 50 Mbps for **Complex** thanks to the *message-level parallelism* mechanism. We believe that this difference comes from the fact that the **Complex** dataloop

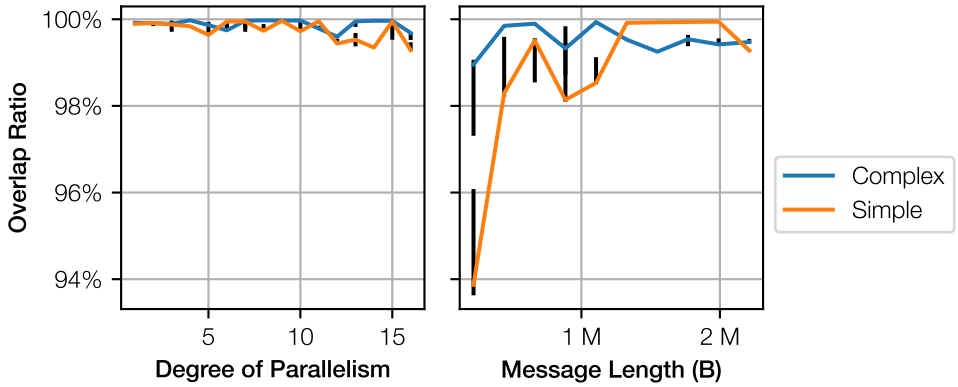


Figure 29: Overlap ratio of the two data types as presented in Figure 26. The left plot shows the ratio with a fixed message size of 2160 KB across different parallelism settings; the right plot shows the ratio with a fixed parallelism of 16 hpus across different message sizes. For each configuration, we plot the median values and 95% confidence intervals across 20 samples.

representation is more complicated and takes more handler time to process compared to **Simple**, manifesting as a lower throughput.

We note in addition that despite the fact that the **Simple** datatype has practically the most simple structure as we have shown in Figure 26, the e2e throughput of the datatype is still way lower than the IPerf3 throughput of around 8 Gbps. We conjecture that the MPI dataloop software implementation contributed significantly for a very big overhead in packet handling, resulting in the overall low throughput.

We further observe that the gemm workload suffers from a moderate slow-down from 20 % to 30 % across different parallelism settings when overlapped with datatypes processing, as compared to the reference case. We believe that this slow-down mainly comes from the memory-intensive nature of gemm, since overlapped datatypes processing would also compete for CPU memory bandwidth through host DMA. We conjecture that the slow-down would be less significant for a more *compute-bound* CPU workload. A similar effect of main memory bandwidth competition can also be observed through a comparison of datatypes processing throughput between the reference and overlapped cases. We recognise that the overlapping ratio stays relatively stable across different degree of parallelism and message sizes.

Figure 29 showed a stable overlap ratio of over 99 % for all parallelism configurations. We also observe that for shorter messages the overlap ratio drops to as low as 94 % due to the short message not allowing longer gemm workloads within the required number of polls. This is further confirmed by a comparison of the overlap ratio between the **Simple** and **Complex** datatypes, showing that shorter datatypes have lower overlap ratio across all message sizes.

4.6 Conclusion

We confirm that it is possible to implement and run complicated packet handlers such as the datatypes handler on FPsPIN. The throughput result leaves much to be desired

compared to the base throughput from IPerf3, mainly due to the limitation from the MPI dataloop implementation and the low frequency at which the HPUs are clocked on the FPGA.

Despite the suboptimal throughput results, we demonstrated that FPsPIN allows applications to reliably achieve almost perfect *communication/computation overlap* for sufficiently long messages through overlapping datatypes procesing with a synthetic gemm workload. This successfully shows off the offloading capabilities of FPsPIN and **achieves the RED-SEA KPI 5** in which we hoped to see 90 % overlap for sufficiently large messages, when in fact we show 92 % or more percent of overlap for all tested message sizes.

5 Conclusion

This deliverable deals with two important aspects regarding communication in HPC systems. First, it introduces improvements to the two MPI implementations MPC and ParaStation MPI. Second, it presents with sPIN a micro-architecture for network accelerators that is optimised for offloading network processing tasks to the NIC.

The developments of the multirail within MPC achieved during this project improved the potential bandwidth, especially for large messages, by a factor proportional to the number of available NICs showing almost linear scaling. This feature has been implemented through a full rewrite of previous communication layer inside MPC in order to improve its design and its stability. Other developments have concerned the support for thread-based communication within the MPC framework and we showed at least that new developments did not add overhead. Tests have been performed on the Inti supercomputer to confirm the performance improvements that we expected.

The pscom4portals plugin of pscom enables efficient MPI communication over BXI. Its adaptation to the new RMA MPI of pscom further improves the support for this high-performance interconnect by providing the software layers running on top with a more direct access to the hardware's RMA capabilities. Using this interface in the upper layers of ParaStation MPI for the implementation of MPI one-sided communication results in significant performance improvements compared with the two-sided-based implementation of the corresponding interface. This way, MPI application codes that are in particular suitable for this communication scheme do not only benefit from its semantics but also from an improved communication performance. Additionally, this deliverable presents a preliminary analysis of pscom's network bridging support in MSA systems including BXI. The test setup on the DEEP system shows that bridging of MPI traffic across IB and BXI is possible at a throughput close to limits imposed by the hardware.

With FPsPIN, ETH Zurich presents the first full-system prototype implementation of the sPIN micro-architecture in hardware. This facilitates the development of the sPIN ecosystem, including the platform itself and applications designed for it. One of the central benefits of the sPIN architectures could be demonstrated with this prototype: it is possible to implement and run packet handlers undertaking tasks usually conducted by the host CPU. Therefore, the MPICH dataloop datatype engine has been ported to sPIN by implementing the corresponding handlers. This way, real overlap of communication and computation becomes possible as the datatype (de-)serialisation can now be executed concurrently to the application's main computation phase. The presented implementation fully achieves KPI 5, which promised a 90% overlap ratio for large-enough messages. In fact an overlap of 92% or more has been demonstrated for all tested message sizes.

Acronyms and abbreviations

A

AM *(Active Message)* 14, 15, 17, 19, 20
API *(Application Programming Interface)* 7, 11–15, 17, 19–22, 24, 26, 27, 29, 67
ARP *(address resolution protocol)* 37, 51
ASIC *(Application-Specific Integrated Circuit)* 40, 53
AXI *(Advanced eXtensible Interface)* 32, 33, 35–40

B

BAR *(base address register)* 41, 42
BRAM *(Block-RAM)* 55
BXI *(BullSequana eXascale Interconnect)* 6–8, 11, 14, 15, 19–21, 23, 26, 27, 29, 30, 65

C

cdc *(clock domain crossing)* clock domain crossing 35, 40
CE *(Counting Event)* 24
CEA *(Commissariat à l'énergie atomique et aux énergies alternatives)* Commissariat à l'énergie atomique et aux énergies alternatives, France 7, 8
CMA contiguous memory allocator 53
CPU *(Central Processing Unit)* 6, 8, 14, 15

D

DAC *(direct-attached copper)* 53
DEEP-SEA DEEP – Software for Exascale Architectures 6, 7, 21
DMA Direct Memory Access 2–4, 8, 14, 32, 36–41, 43–46, 52–54, 57, 63

E

e2e end-to-end 31, 54, 56–58, 60, 61, 63
EDA *(Electronic Design Automation)* 54
EOM *(End of Message)* 36
EQ *(Event Queue)* 24
ESB *(Extreme Scale Booster)* A module of the DEEP system with highly energy-efficient many-core processors as booster nodes but a reduced amount of memory per core at high bandwidth. 27
EXTX *(sPIN Handler Execution Context)* 36, 37, 44, 50–52

F

FIFO *(First-In-First-Out)* 35, 36, 38, 43
Flip-Flop *(Flip-Flop)* 55

FPGA *(Field Programmable Gate Array)* 36, 40, 41, 52–54, 59

G

GASPI *(Global Address Space Programming Interface)* A programming model for one-sided and asynchronous communication between computing nodes. 30
gemm *(GEMM)* A matrix matrix multiplication. 60–64

H

HCA *(Host Channel Adapter)* 21, 27
HER *(Handler Execution Request)* 32–34, 36–38, 44, 46, 50, 52
HPC *(High Performance Computing)* 7, 8, 12, 65, 67
hpu *(handler processing unit)* a core which executes spin handlers 43, 45–47, 50–52, 60, 62, 63

I

IB *(InfiniBand)* A networking communication standard for HPC clusters 7, 27, 29, 65
ICMP *(Internet Control Message Protocol)* 54, 56, 57, 59
ILA *(Integrated Logic Analyzer)* 54
IPG *(inter-message gap)* 45
IPG *(inter-packet gap)* 49

J

JSC *(Jülich Supercomputing Centre)* Jülich Supercomputing Centre GmbH, Jülich, Germany 7

L

LUT *(lookup table)* 55

M

ME *(Matching List Entry)* 23
MPC *(Multi-Processor Computing)* The MPC framework regroups an MPI, an Open Multi-Processing (OpenMP), and a pthread implementation in the same software, for better interoperability 6–8, 11, 15–17, 20, 65
MPI *(Message-Passing Interface)* An API specification typically used in parallel programs that allows processes to communicate with one another by sending and receiving messages 6–8, 12, 13, 15, 20–22, 26, 27, 29, 30, 65, 67, 68
MPI *(Message Passing Interface)* Message Passing Interface 3, 31, 37, 48–50, 54, 59, 60, 63, 64
MPQ *(Message Processing Queue)* Message Processing Queue 33, 47, 55
MSA *(Modular Supercomputer Architecture)* 7, 20, 21, 27, 30, 65
MTU *(Maximum Transfer Unit)* 11, 38

N

NIC *(Network Interface Card)* 6–8, 11–17, 19, 20, 65
NIC *(Network Interface Card)* 30–32, 35, 37, 38, 42–46, 50–52

O

OpenMP *(Open Multi-Processing)* Application programming interface that support multi-platform shared memory multiprocessing 67

P

P2P *(Point-to-point)* 4, 8, 10, 11
ParaStation Software for cluster management and control developed by ParTec 68
ParaStation Modulo HPC middleware and management stack for modular supercomputing developed by ParTec 20, 68
ParaStation MPI The MPI runtime environment of ParaStation Modulo developed by ParTec. 6, 7, 20, 21, 26–30, 65
ParTec ParTec AG, Munich, Germany. 7, 68
PCIe *(PCIe)* 41, 42, 52, 54, 57, 59
PGAS *(Partitioned Global Address Space)* 7, 21, 30
PID *(Process Identifier)* 23
pscom The low-level communication layer of ParaStation MPI 6, 7, 20–23, 26, 27, 29, 30, 65
PTE *(Portals Table Entry)* 23
PTI *(Portal Table Index)* 23
PULP *(PULP Ultra Low Power)* 40, 47, 53, 59

Q

QOR *(Quality of Results)* 54, 55

R

RDMA *(Remote Direct Memory Access)* 47
RED-SEA Network Solution for Exascale Architecturesfor Exascale Architectures Project 7
RMA *(Remote Memory Access)* 6, 7, 9, 12, 20–27, 29, 65
ROCE *(RDMA over converged Ethernet)* 48
RTL Register Transfer Level 35, 54
RTS *(Ready To Send)* 12
RTS *(Ready-to-send)* 62
rtt round trip time 50, 54, 56–59

S

T

U

W

References

[1] Damian Alvarez. “JUWELS Cluster and Booster: Exascale Pathfinder with Modular Supercomputing Architecture at Juelich Supercomputing Centre”. In: *Journal of large-scale research facilities* 7 (2021), A183. DOI: 10.17815/jlsrf-7-183.

[2] *Auxiliary Bus — The Linux Kernel documentation*. URL: https://www.kernel.org/doc/html/next/driver-api/auxiliary_bus.html (visited on 08/18/2023).

[3] Theophilus Benson et al. “Understanding data center traffic characteristics”. en. In: *Proceedings of the 1st ACM workshop on Research on enterprise networking*. Barcelona Spain: ACM, Aug. 2009, pp. 65–72. ISBN: 978-1-60558-443-0. DOI: 10.1145/1592681.1592692. URL: https://dl.acm.org/doi/10.1145/1592681.1592692 (visited on 08/04/2023).

[4] Ronald Brightwell et al. “The Portals 4.3 Network Programming Interface”. In: (June 2022). DOI: 10.2172/1875218. URL: https://www.osti.gov/biblio/1875218.

[5] Shiyi Cao, Salvatore Di Girolamo, and Torsten Hoefler. “Accelerating Data Serialization/Deserialization Protocols with In-Network Compute”. en. In: *2022 IEEE/ACM International Workshop on Exascale MPI (ExaMPI)*. Dallas, TX, USA: IEEE, Nov. 2022, pp. 22–30. ISBN: 978-1-66546-341-6. DOI: 10.1109/ExaMPI56604.2022.00008. URL: https://ieeexplore.ieee.org/document/10027020/ (visited on 08/09/2023).

[6] Don Cohen and William Stearns. *IPTables U32 Match Tutorial*. URL: http://www.stearns.org/doc/iptables-u32.current.html (visited on 08/04/2023).

[7] Salvador Coll et al. “Using Multirail Networks in High-Performance Clusters”. In: *In Proceedings of the 2001 IEEE International Conference on Cluster Computing*. IEEE Computer Society, 2001, pp. 15–24.

[8] Daniele De Sensi et al. “Flare: flexible in-network allreduce”. en. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. St. Louis Missouri: ACM, Nov. 2021, pp. 1–16. ISBN: 978-1-4503-8442-1. DOI: 10.1145/3458817.3476178. URL: https://dl.acm.org/doi/10.1145/3458817.3476178 (visited on 01/17/2022).

[9] Salvatore Di Girolamo et al. “Building Blocks for Network-Accelerated Distributed File Systems”. en. In: *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*. Dallas, TX, USA: IEEE, Nov. 2022, pp. 1–14. ISBN: 978-1-66545-444-5. DOI: 10.1109/SC41404.2022.00015. URL: https://ieeexplore.ieee.org/document/10046100/ (visited on 08/09/2023).

[10] Salvatore Di Girolamo et al. “Network-accelerated non-contiguous memory transfers”. en. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Denver Colorado: ACM, Nov. 2019, pp. 1–14. ISBN: 978-1-4503-6229-0. DOI: 10.1145/3295500.3356189. URL: https://dl.acm.org/doi/10.1145/3295500.3356189 (visited on 01/16/2022).

[11] Salvatore Di Girolamo et al. “PsPIN: A high-performance low-power architecture for flexible in-network compute”. In: *arXiv:2010.03536 [cs]* (June 2021). arXiv: 2010.03536. URL: http://arxiv.org/abs/2010.03536 (visited on 10/01/2021).

[12] Thomas J. DiCiccio and Bradley Efron. “Bootstrap confidence intervals”. In: *Statistical Science* 11.3 (Sept. 1996). Publisher: Institute of Mathematical Statistics, pp. 189–228. ISSN: 0883-4237, 2168-8745. DOI: 10.1214/ss/1032280214. URL: https://projecteuclid.org/journals/statistical-science/volume-11/issue-3/Bootstrap-confidence-intervals/10.1214/ss/1032280214.full (visited on 08/31/2023).

[13] Thorsten Von Eicken et al. *Active Messages: a Mechanism for Integrated Communication and Computation*. 1992.

[14] Alex Forencich. *Verilog PCI Express Components Readme*. original-date: 2019-01-08T05:28:51Z. Aug. 2023. URL: https://github.com/alexforencich/verilog-pcie (visited on 08/04/2023).

[15] Alex Forencich et al. “Corundum: An Open-Source 100-Gbps Nic”. en. In: *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. Fayetteville, AR, USA: IEEE, May 2020, pp. 38–46. ISBN: 978-1-72815-803-7. DOI: 10.1109/FCCM48280.2020.00015. URL: https://ieeexplore.ieee.org/document/9114811/ (visited on 01/18/2022).

[16] Edgar Gabriel et al. “Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation”. In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Ed. by Dieter Kranzlmüller, Péter Kacsuk, and Jack Dongarra. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 97–104. ISBN: 978-3-540-30218-6.

[17] Torsten Hoefler, Andrew Lumsdaine, and Wolfgang Rehm. “Implementation and performance analysis of non-blocking collective operations for MPI”. en. In: *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*. Reno Nevada: ACM, Nov. 2007, pp. 1–10. ISBN: 978-1-59593-764-3. DOI: 10.1145/1362622.1362692. URL: https://dl.acm.org/doi/10.1145/1362622.1362692 (visited on 09/10/2023).

[18] Torsten Hoefler et al. “sPIN: High-performance streaming Processing in the Network”. en. In: *arXiv:1709.05483 [cs]* (Oct. 2017). arXiv: 1709.05483. URL: http://arxiv.org/abs/1709.05483 (visited on 12/25/2020).

[19] KAWAZOME Ichiro. *u-dma-buf(User space mappable DMA Buffer)*. original-date: 2015-07-13T09:17:35Z. Aug. 2023. URL: https://github.com/ikwzm/udmabuf (visited on 08/18/2023).

[20] “IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(TM)) Base Specifications, Issue 7”. In: *IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)* (Jan. 2018). Conference Name: IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008), pp. 1–3951. DOI: 10.1109/IEEESTD.2018.8277153.

[21] *Incremental Implementation • Vivado Design Suite User Guide: Implementation (UG904) • Reader • AMD Adaptive Computing Documentation Portal*. URL: https://docs.xilinx.com/r/2021.1-English/ug904-vivado-implementation/Incremental-Implementation (visited on 08/28/2023).

[22] *iputils/iputils*. original-date: 2014-04-18T12:14:53Z. Aug. 2023. URL: https://github.com/iputils/iputils (visited on 08/31/2023).

[23] *Jinja — Jinja Documentation (3.1.x)*. URL: https://jinja.palletsprojects.com/en/3.1.x/ (visited on 08/22/2023).

[24] Wolfgang John and Sven Tafvelin. "Analysis of internet backbone traffic and header anomalies observed". en. In: *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*. San Diego California USA: ACM, Oct. 2007, pp. 111–116. ISBN: 978-1-59593-908-1. DOI: 10.1145/1298306.1298321. URL: https://dl.acm.org/doi/10.1145/1298306.1298321 (visited on 08/04/2023).

[25] Flavel Katherine. *stping/dgping: TCP and UDP ping*. original-date: 2020-03-30T02:25:36Z. Aug. 2023. URL: https://github.com/katef/stping (visited on 08/31/2023).

[26] Moein Khazraee et al. "Rosebud: Making FPGA-Accelerated Middlebox Development More Pleasant". en. In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. Vancouver BC Canada: ACM, Mar. 2023, pp. 586–605. ISBN: 978-1-4503-9918-0. DOI: 10.1145/3582016.3582067. URL: https://dl.acm.org/doi/10.1145/3582016.3582067 (visited on 06/05/2023).

[27] Jiuxing Liu et al. *MPI over InfiniBand: Early Experiences*. Tech. rep. 2003.

[28] Patrick Mochel and Mike Murphy. *sysfs - The filesystem for exporting kernel objects*. Aug. 2011. URL: https://www.kernel.org/doc/Documentation/filesystems/sysfs.txt (visited on 08/21/2023).

[29] Gilles Moreau et al. *RED-SEA Deliverable 4.3: Planned MPI-related optimizations*. Tech. rep. Sept. 2022.

[30] Marc Pérache, Hervé Jourdren, and Raymond Namyst. "MPC: A Unified Parallel Runtime for Clusters of NUMA Machines". In: *Proceedings of the 14th International Euro-Par Conference on Parallel Processing*. Euro-Par '08. Las Palmas de Gran Canaria, Spain: Springer-Verlag, 2008, pp. 78–88. ISBN: 978-3-540-85450-0. DOI: 10.1007/978-3-540-85451-7_9. URL: http://dx.doi.org/10.1007/978-3-540-85451-7_9.

[31] Simon Pickartz and Manolis Marazakis. *DEEP-SEA Deliverable 3.2: Complete System Software Implementation*. Tech. rep. Mar. 2023.

[32] S. Pai Raikar et al. "Designing Network Failover and Recovery in MPI for Multi-Rail InfiniBand Clusters". In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. 2012, pp. 1160–1167. DOI: 10.1109/IPDPSW.2012.142.

[33] Robert Ross et al. “Processing MPI Datatypes Outside MPI”. en. In: *Recent Advances in Parallel Virtual Machine and Message Passing Interface*. Ed. by Matti Ropo, Jan Westerholm, and Jack Dongarra. Vol. 5759. Series Title: Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 42–53. ISBN: 978-3-642-03769-6 978-3-642-03770-2. DOI: 10.1007/978-3-642-03770-2_11. URL: http://link.springer.com/10.1007/978-3-642-03770-2_11 (visited on 07/11/2023).

[34] Davide Rossi et al. “PULP: A parallel ultra low power platform for next generation IoT applications”. In: *2015 IEEE Hot Chips 27 Symposium (HCS)*. Aug. 2015, pp. 1–39. DOI: 10.1109/HOTCHIPS.2015.7477325.

[35] Whit Schonbein et al. “Measuring Multithreaded Message Matching Misery”. In: *Euro-Par 2018: Parallel Processing: 24th International Conference on Parallel and Distributed Computing, Turin, Italy, August 27 - 31, 2018, Proceedings*. Turin, Italy: Springer-Verlag, 2018, pp. 480–491. ISBN: 978-3-319-96982-4. DOI: 10.1007/978-3-319-96983-1_34. URL: https://doi.org/10.1007/978-3-319-96983-1_34.

[36] Pavel Shamis et al. “UCX: An Open Source Framework for HPC Network APIs and Beyond.” In: *Hot Interconnects*. IEEE Computer Society, 2015, pp. 40–43. ISBN: 978-1-4673-9160-3. URL: http://dblp.uni-trier.de/db/conf/hoti/hoti2015.html#ShamisVLBHIDSGL15.

[37] Sayantan Sur et al. “RDMA read based rendezvous protocol for MPI over InfiniBand: design alternatives and benefits”. In: *In PPoPP ’06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles*. 2006, pp. 32–39.

[38] *Verilog Hierarchical Reference Scope*. en-GB. URL: https://www.chipverify.com/verilog/verilog-hierarchical-reference-scope (visited on 08/04/2023).

[39] Zhang Xianyi, Wang Qian, and Zhang Yunquan. “Model-driven Level 3 BLAS Performance Optimization on Loongson 3A Processor”. In: *2012 IEEE 18th International Conference on Parallel and Distributed Systems*. ISSN: 1521-9097. Dec. 2012, pp. 684–691. DOI: 10.1109/ICPADS.2012.97.

[40] Pengcheng Xu. “Full-System Evaluation of the sPIN In-Network-Compute Architecture”. In: (2023). URL: https://doi.org/10.3929/ethz-b-000637676.

