Bring the BitCODE - Moving Compute and Data in Distributed Heterogeneous Systems

1st Wenbin Lu§
Stony Brook University
Stony Brook, NY
wenbin.lu@stonybrook.edu

2nd Luis E. Peña§
Arm Research
Austin, TX
luis.epena@arm.com

3rd Pavel Shamis§
Arm Research
Austin, TX
pavel.shamis@arm.com

4th Valentin Churavy§
MIT
Cambridge, MA
vchuravy@mit.edu

5th Barbara Chapman
Stony Brook University
Stony Brook, NY
barbara.chapman@stonybrook.edu

6th Steve Poole
Los Alamos National Laboratory
Los Alamos, NM
swpoole@lanl.gov

2022 IEEE International Conference on Cluster Computing (CLUSTER) | 978-1-6654-9856-2/22/$31.00 ©2022 IEEE | DOI: 10.1109/CLUSTER51413.2022.00017

***Abstract*—In this paper, we present a framework for moving compute and data between processing elements in a distributed heterogeneous system. The implementation of the framework is based on the LLVM compiler toolchain combined with the UCX communication framework. The framework can generate binary machine code or LLVM bitcode for multiple CPU architectures and move the code to remote machines while dynamically optimizing and linking the code on the target platform. The remotely injected code can recursively propagate itself to other remote machines or generate new code.**

The goal of this paper is threefold: (a) to present an architecture and implementation of the framework that provides essential infrastructure to program a new class of disaggregated systems wherein heterogeneous programming elements such as compute nodes and data processing units (DPUs) are distributed across the system, (b) to demonstrate how the framework can be integrated with modern, high-level programming languages such as Julia, and (c) to demonstrate and evaluate a new class of eXtended Remote Direct Memory Access (X-RDMA) communication operations that are enabled by this framework.

To evaluate the capabilities of the framework, we used a cluster with Fujitsu CPUs and heterogeneous cluster with Intel CPUs and BlueField-2 DPUs interconnected using high-performance RDMA fabric. We demonstrated an X-RDMA pointer chase application that outperforms an RDMA GET-based implementation by 70% and is as fast as Active Messages, but does not require function predeployment on remote platforms.

***Index Terms*—Distributed Systems, SmartNIC, DPU, Programming Models, HPC**

I. Introduction

As Moore's law runs out of steam, hyperscalers are looking for every possible opportunity to maximize the efficiency of their datacenters. Their pursuit for efficiency, having now pushed the limits of the SoC, has moved the system architecture towards disaggregation using low-latency, high-bandwidth interconnects including PCIe Gen6, CXL [1], CCIX [2], and converged and custom Ethernet. Such disaggregation comes at a cost of reduced efficiency due to data movement overheads and increased security complexity. This reduction has caused a perfect storm for the emergence of data processing units (DPUs), computational storage devices (CSDs), and other custom accelerators. The basic goals of these devices are (a) to move the compute closer to the data in order to increase efficiency and (b) to enhance security through isolation.

DPU solutions have been adopted by multiple hyperscalers and they are increasingly seen in datacenter networks. CSDs are the next frontier of innovation at the disaggregated data center. Despite having the potential for rich programmable capabilities, both these kinds of devices are primarily used as fixed-function accelerators with very limited control from the application on the host. The application cores in these devices are currently limited due to the software stack not being well established for this domain; the hardware and software are functional and deployed at scale, but in terms of application maturity and adoption, both are in their nascent stages and reminiscent of the early days of GPGPU compute.

Intel has developed the IPDK [3] framework which bundles container with multiple existing software packages like Open Virtual Switch (OVS) [4] that can be programmed using the P4 programming language [5]. Nvidia's DOCA [6] package also bundles a rich set of software packages ranging from high-level Snort [7], OVS, and NVMe virtualization to low-level abstraction like rdma-core [8] and DPDK [9]. In both cases the software has to be predeployed such that DPDK exposes fixed-function offload and acceleration. As a result, host based applications do not have direct access to the compute resources on DPUs.

We believe there are several challenges contributing to the limited exposure of compute resources to the host applications:

- Security constraints – one of the main use-cases for current DPUs in the datacenter is the security-control-point and hypervisor offload for bare-metal instances. Therefore, deployment of user application codes side-by-side with infrastructure management codes (verified and trusted) raise concerns about the impact on system security. In this paper we do not focus on addressing security concerns, instead relying on existing hardware [10] and software confidential compute [11] [12] efforts.

§Equal contribution

2168-9253/22/$31.00 ©2022 IEEE
DOI 10.1109/CLUSTER51413.2022.00017

- Lack of standardized interfaces for offloading compute to Arm SoC subsystem and embedded accelerators – this is the main challenge that we aim to address with the *Three-Chains* framework. It is important to note that *Three-Chains* is a path-finding project that aims to demonstrate how such interfaces could be implemented and to evaluate potential trade-offs.

In order to democratize programmability of this new class of disaggregated systems and directly expose programming elements in DPUs and other accelerators to application layers we developed *Three-Chains*. The *Three-Chains* framework is implemented from scratch, and it is inspired by *Two-Chains* [13]. It utilizes the idea of remote binary code injection while introducing new capabilities and bringing portability and performance to new levels. *Three-Chains* provides infrastructure for moving functions within a heterogeneous computing environment, caching, remote dynamic linking, and remote recursive invocation. We decided to name it *Three-Chains* because it extended original toolchain with LLVM [14]. *Three-Chains* leverages the LLVM framework and intermediate representation (IR) for portable function representation, just-in-time (JIT) compilation, and execution.

Using LLVM IR, the framework implements a multi-architectural function representation which we call `fat-bitcode`, as well as a separate binary function representation. One of the benefits of using `fat-bitcode` beyond portability is that the code gets optimized for the target micro-architecture. Since LLVM is used by multiple programming languages as a backend, *Three-Chains* thus also supports multiple programming languages. For this paper we focused on the C and Julia programming languages [15] as representative languages for both low- and high-level programming environments.

Leveraging LLVM, *Three-Chains* has full support for distributed linking wherein functions get linked on the target machine. As a result, *Three-Chains* functions can interact with external libraries including UCX itself. This also means that *Three-Chains*'s remotely injected functions can recursively inject functions to other processing elements in the system. Such a function can propagate itself to a remote machine and potentially dynamically select new functions for further remote injections. Using functions in this way, we have implemented a novel type of one-sided network remote operations hereafter called eXtended Remote Direct Memory Access (X-RDMA), in which RDMA operations trigger recursively executed code inside the network with processing elements. In order to minimize the overhead associated with moving code over the fabric, we have also implemented a caching mechanism that substantially reduces the size of the network messages and avoids the JIT compilation already-seen code.

The key contributions of the paper are:

- High-performance framework for moving and executing complex functions using either binary representation or LLVM's intermediate representation
- Network protocols and caching algorithms for efficient code delivery and multi-architecture code representation
- Implementation of dynamic linking for remote injected codes using LLVM's ORC-JIT API
- Evaluation of overheads for remote function communication, deployment (JIT vs binary), caching, and invocation
- Extension of an in-process JIT compiled language (Julia) with remote function injection, linking, and execution
- Introduction of X-RDMA operations, and a demonstration in the form of pointer-chase algorithms to DPUs and host processing

II. Background

There are multiple projects and programming models that implement high-performance RPC message-like semantics [16] [17] [18]. The Three-Chains project was developed from scratch while leveraging concepts from earlier *Two-Chains* research [13] [19]. The primary difference between our work and the previous state of the art is that (a) our implementation moves the code and data while supporting multiple code representations and (b) it leverages JIT techniques to support optimizations for heterogeneous systems.

Below, we highlight some of the projects that aim to solve a similar class of problems. These projects are GASNet [20], Google's Snap microkernel [21], Charm++ [22], CHAMELEON [23], FaRM [24] and Julia [25], [26]. GASNet is a communication library used by the HPC community as a communication backend for programming languages like UCP, Chapel, and others. GASNet provides APIs for registering and invoking active messages, but those require code presence on the remote machines. This differs from our approach where the code is communicated over the network and dynamically linked and optimized on the target platform without requiring compile-time registration. Google's Snap Microkernel project provides a platform for remote procedure calls in the context of network functionality distribution but it functions as an OS extension whereas *Three-Chains* is implemented as a user space library. Charm++ is a distributed programming model that defines distributed C++ objects with a unified logical view and the ability to call methods on those objects. It implements a task scheduler, automatic object migrations, a run-time for task launch, communication channels, among other features. The CHAMELEON framework uses compiler `#pragma` and runtime APIs to define OpenMP tasks as entities that can be moved around the distributed system using its built-in task scheduler. Conceptually, CHAMELEON's approach is similar to Charm++, and our framework can serve both programming models as a foundational abstraction for object/task migration. The FaRM project implements a distributed computing platform for distributed shared memory programming. It uses RDMA network for remote object manipulation, but unlike our work it does not provide functionality for moving compute. In *Three-Chains*, the Julia programming language is used to demonstrate how to integrate with high-level programming models and languages. Julia provides a remote procedure call (RPC) library, called `Distributed.jl`, in the standard library. It allows the user to send unoptimized functions to a remote process, and the function is then optimized and

compiled on the remote process; in contrast, *Three-Chains* moves optimized bitcode emitted for the target architecture. Furthermore `Distributed.jl` does not take advantage of RDMA networks. LLVM's ORC JIT supports a remote executor protocol that allows for JIT compilation from one process into the other as well as executing remotely compiled functions; in contrast, *Three-Chains* provides a mechanism for propagation of bitcode allowing functions to propagate across a cluster.

III. Design and Implementation

The primary problem statement that *Three-Chains* addresses is considering heterogeneous distributed system design, like the one that can be found in modern datacenters with GPUs, DPUs, and other processing elements, how one can deploy and execute application-defined functions. We believe that for a software framework to address the above problem, it has to provide the following functionality:

- A workflow and APIs for moving code and data.
- An infrastructure for a dynamic linking of dependencies on the target platform.
- Support for invoking remote or local functions.
- Support for multiple CPU architectures.

It is important to point out that this is very different from RPC and Active Messages communication semantics, which explicitly assume that the target procedure call is present on the remote machine with all its dependencies.

A. Three-Chains Workflow

The *Three-Chains* framework provides a set of C APIs that defines how to create and move executable code in a distributed system. *Three-Chains* uses the UCX [27] communication framework for moving code and data around the system. An executable piece of code in *Three-Chains* is called an *ifunc*, which stands for "injected function". In addition to the *ifunc* code, the user can also send a contiguous chunk of memory, called payload, from the source process to the target process. The payload contains data that is needed by the *ifunc* when it is executing on the target process. The *Three-Chains* API is implemented as an extension of the UCP interface in UCX, and it is inspired by API design described here [19]. Due to space constraints, we do not include the API description here and focus on the workflow for application codes.

A brief description of the Three-Chains workflow is shown in Figure 1 and is described as follows. The application developer creates an *ifunc* library that implements an entry function (main) and several other functions required by the *Three-Chains* framework. Once the *ifunc* is processed using the *Three-Chains* toolchain, the generated files should be placed in a directory that can be located by *Three-Chains*. Then, in the application, the *ifunc* library is registered using its name (e.g., "foo"), and a handle is returned to the application. With this handle, the user can create and send *ifunc* messages of type "foo" to a target process element.

The target process should use an UCX *ifunc* polling function to check the message buffer and handle incoming *ifunc*

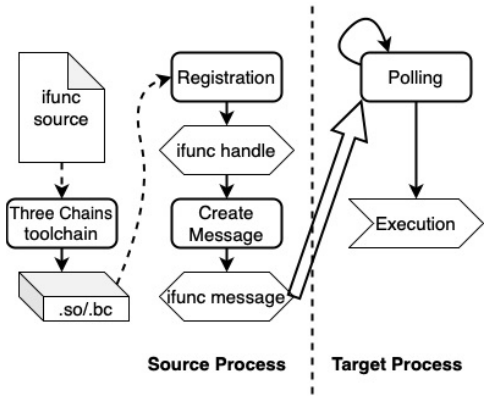

Fig. 1: *Three-Chains* workflow

messages. Once a message has arrived, the polling function will load the *ifunc* code to the target process's address space and invoke the entry function with a pointer to the payload, and a user-defined target pointer. Ideally, the target processes should setup a daemon thread that polls the message buffers periodically. The dynamic nature of *ifunc*s allows adding new functionalities to the target process without recompiling the application and restarting it.

B. The Challenges of the Binary-Based ifunc Implementation

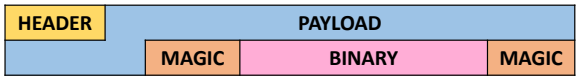

Fig. 2: Message frame of binary-based *ifuncs*. HEADER carries information that describes that type and format of the message, MAGIC field is used to discover and identify message delivery, BINARY fields correspond to binary code.

The original implementation of the *ifunc* concept is based on a standard ELF binary library format. When the user registers an *ifunc*, the runtime loads the `.so` file using `dlopen()`. To send a message, the `.text` and `.data` sections of the *ifunc* dynamic library are copied directly from the dynamic library and packed into a contiguous buffer along with the payload and some other metadata, as shown in Figure 2.

The major challenge of this approach is that run-time symbol resolution must be performed again on the target process, i.e., one has to do remote dynamic linking. In our previous work [19], the remote dynamic linking is implemented by redirecting all Global Offset Table (GOT) references to a special location, which will contain the GOT reconstructed by the target process's runtime, and so all symbols will resolve to the correct address. The reconstruction requires the presence of a dynamic library `.so` file on the target process's file system, and the use of `ld.so` linker to perform relocation of the symbols. If the *ifunc* does not use external symbols, then it is defined as "pure", and we skip GOT patching and go straight to execution. The details of the binary-based implementation can be found in [19].

The above approach has several issues. First, the binary code of the *ifunc* dynamic library is instruction set architecture (ISA)-specific, which means the `.so` file for a x86_64 host cannot be used to send *ifunc* messages to an Arm-based SmartNIC. This issue could be solved by setting up cross-compilation toolchain to compile the *ifunc* for different ISAs/μArchs and select the appropriate one when creating an *ifunc* message. Function-multiversioning could even pack implementations of the same function specialized for different μArchs into the same binary, so we can have *ifunc* specialized for different μArch of the same ISA, in a single `.so` file. But overall, solutions to this problem tend to greatly increase the complexity of both the *Three-Chains* toolchain and the user's workflow.

The second issue is that, patching the assembly code for GOT redirections and remote dynamic linking must be implemented separately for all supported ISAs (x86, Arm, PowerPC, RISC-V, etc.), which makes the framework complicated and error-prone. For example, if the *ifunc* library uses OpenMP to do multi-threaded computation and is compiled by GCC, the compiler-generated work-sharing functions need special treatment to be relocated correctly. Another example is that older versions of GCC compiles C11 atomic operations provided by `<stdatomic.h>` to special outlined functions. Bodies of these functions are inserted by the linker, not visible in GCC's assembly output. Therefore, the assembly patcher could not change how they are relocated and the *ifunc* crashes. The issues above originate from the fact that *Two-Chains* [13] is sending compiled machine code that is already linked. If we instead go to the other end of the spectrum, send C source code from the source process, and compile it on the target, it would be as portable as it gets, but the overhead in terms of compilation time and size of the dependencies (headers, C compiler) would be substantial.

C. The New Approach - Bring the Bitcode

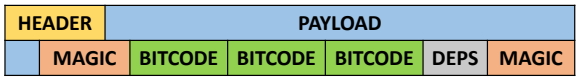

Fig. 3: Message frame of bitcode-based *ifuncs*. `HEADER` carries information that describes that type and format of the message, `MAGIC` field is used to discover and identify message delivery, `BITCODE` fields correspond to `fat-bitcode`, and `DEPS` describe dependencies for the bitcode.

LLVM is a popular framework for building compilers, it has been ported to many different ISAs and is highly modular. LLVM provides a unified intermediate representation (IR) that is capable to represent the semantics of high-level programming languages, as well as a mature back-end compiler that transforms the IR into efficient machine code. Today, many language implementations target the LLVM IR so that they can focus on front-end optimizations while still getting good portability across different ISAs. Clang, Julia, Rust, and Haskell are among the most successful LLVM-based language implementations.

One key advantage of LLVM for this project is that it is natively a cross-compiler, which means that generating IR/machine code for different ISAs can be as simple as passing a single different flag to the same compiler program. Since symbol resolution and linking happen downstream of LLVM IR generation, compiling the source code to IR only needs the header files of the dependencies, sans the special linkers and C runtime files required by GCC cross-compilation. This feature of LLVM makes it very suitable for *Three-Chains*, which needs to run on heterogeneous systems that have CPUs of different ISAs. Lastly, LLVM provides JIT compilation for its IR through the ORC interface (ORC-JIT). ORC-JIT allows the user to compile, link, and execute LLVM IR entirely in the memory, without having to save the generated machine code to the disk and load it later.

For these reasons, we have decided to implement a new *ifunc* backend based on LLVM. Instead of shipping binary code loaded from a dynamic library, we ship the binary representation of the *ifunc* library's LLVM IR, referred to as the bitcode. When the target process receives a bitcode-based *ifunc*, the Three-Chains runtime will use LLVM ORC-JIT to compile the bitcode into machine code. This compilation step does take some time, but the generated machine code is stored in a LLVM internal buffer, which stays alive until the *ifunc* is de-registered. This means that subsequent *ifunc* messages of the same type do not need to repeat the compilation step and will be executed without delay.

Symbol resolution is also handled by LLVM in this new implementation. While the LLVM IR does not contain information about shared library dependencies, ORC-JIT does provide an interface to load shared libraries and perform runtime symbol resolutions. In this new implementation, the user should provide a text file `foo.deps` that lists all the dynamic library needed by *ifunc* foo. For example, OpenMP applications need `libomp.so` and OpenSSL applications generally need `libcrypto.so`. This list of shared libraries will also be shipped to the target process so that the *Three-Chains* runtime can load them before invoking the main function of the *ifunc*.

Since LLVM IR is ISA-dependent, an *ifunc* message should contain bitcode for all the ISAs it intends to run on. Therefore, the Three-Chains toolchain will generate bitcode files for all the targets supported by the toolchain's Clang compiler. These `.bc` files are available on the source process and are identified by its target triple (e.g. `x86_64-pc-linux-gnu`). When the *ifunc* is registered, all the bitcode files will be packed into a bitcode archive and for the rest of the paper we reference it as a `fat-bitcode`. The `fat-bitcode` is shipped with the payload and list of bitcode dependencies, as shown in Figure 3.

The target process extracts the bitcode that matches its local target architecture and compiles it with LLVM ORC-JIT. If we know the exact μArch of the target process's CPU and would like to optimize for it, we could provide this information to Clang when we build the *ifunc* bitcode. Then, on the target process, ORC-JIT can instruct LLVM code-generation back-end to emit machine code specialized for the μArch

of the CPU it is running on. In our experiments, ORC-JIT running on Fujitsu A64FX was able to produce ARM LSE atomic instructions and SVE vector instructions, using bitcode generated on an Intel Xeon CPU. Doing the experiment the other way around, we have successfully generated x86 atomic instructions and AVX2 vector instructions. This means that for performance critical parts of the *ifunc* that benefit from vectorization and/or atomics, the bitcode implementation can reach the same level of performance as binary-based *ifuncs*.

One additional advantage of bringing in LLVM is that we can support *ifunc* libraries written in other high-level languages like Julia, as more and more programming languages are using the LLVM compiler framework. We will discuss the Julia integration in Section III-E.

However, the LLVM bitcode based implementation does have a drawback: it pulls in a portion of LLVM as a dependency. Although this portion is much smaller than a full C compiler, it still is a 57 MiB file for the shared library version or increases the size of the application binary by around 23 MiB if linked statically. If the target system does not have a modern C++ compiler that could build LLVM, or is extremely space-constrained, this could be an issue.

D. Code Execution and Caching

When the user creates an *ifunc* message, the *Three-Chains* runtime packs the message header, the payload, and the binary or bitcode into a single continuous block of memory, called the message frame, as shown in Figures 2 and 3. The receiving process of the *ifunc* message first registers the code section of the message to obtain an in-memory executable, and then invokes the entry function with the payload and the target pointer.

While this process is relatively straightforward, the size of the binary or bitcode shipped in an *ifunc* message is a non-trivial source of communication overhead. For example, even if the *ifunc* is as simple as increasing a counter on the target process, compiling the *ifunc* library with high optimization levels like `-O3` can increase the size of the shipped binary code from 65 bytes to 90 bytes, and the increase is even more significant for more complex libraries. The overhead is particularly obvious for the new bitcode implementation, as it must ship a `fat-bitcode` that supports multiple ISAs of the participating processes. For the previously mentioned counter-increasing *ifunc*, its bitcode archive that supports both x86_64 and AArch64 processors is around 5 KiB. Shipping such a large amount of extra data could have a significant negative impact on communication latency and message rate for small messages, even on high-performance RDMA interconnects.

To address this issue, we have added code caching mechanism for both *ifunc* implementations. To keep the simplicity of the API, all user-created *ifunc* messages still contain both code and data, and the UCX runtime will perform caching fully transparent to the user. The process is demonstrated in Figure 4.

For both the binary and the bitcode implementation, the code is placed at the last section of the *ifunc* message, in between two signals (`MAGIC`) bytes. The idea here is that we always construct a full message, but skip sending the code section if we know the target process has already seen this type of *ifunc* at least once. Note that we do not need two separate calls to UCX communication routines to send the entire message, the message frame is contiguous in memory, and we control what to send by simply passing different message size arguments to the UCP PUT interface.

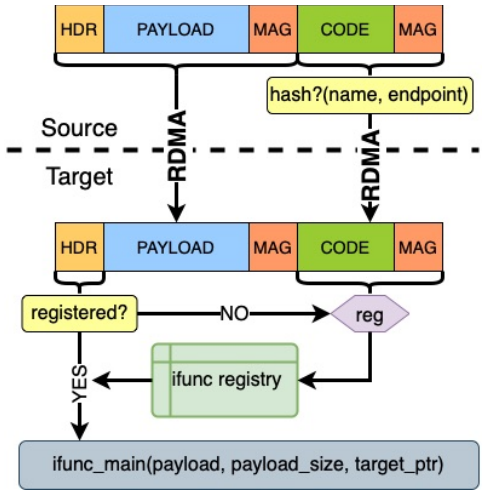

Fig. 4: *ifunc* caching mechanism

When the target process receives an *ifunc* message, the *Three-Chains* runtime first looks at the message header and see if this *ifunc* has already been registered on the local process. If this is the first time this process receives an *ifunc* of this type, then we assume we have received a full message that contains the code section. The runtime will then automatically register this *ifunc* and copy the code section to a side buffer for execution. For binary-based *ifuncs*, the side buffer will have proper alignment and execution privilege, and the runtime will patch its GOT before starting the execution. For bitcode-based *ifunc*s, the runtime will save the bitcode archive to the side buffer, create a LLVM ORC-JIT instance with the bitcode that matches the local process's target architecture, and start execution.

If the target process has already registered the *ifunc* before, then it assumes the message was truncated and so only waits for the delivery of the payload section. For both binary and bitcode *ifunc*s, we have stored the pointer to their `main` functions during registration, so the runtime invokes the *ifunc*'s `main` function immediately once the payload is fully delivered.

When the source process sends an *ifunc* message, the *Three-Chains* runtime first checks a hash table to see if it has sent an *ifunc* message of this particular type to the specified UCP endpoint before. If not, then the endpoint is added to the hash table and the entire message is sent. If the UCP endpoint is already in the hash table, we know the target has already cached the code for this type of *ifunc*, so the *Three-Chains* runtime will only send the message up to the second last signal byte, skipping the code section and the trailer signal byte. This

hash table lookup adds a bit of overhead, but it's very effective in reducing the message size and improving performance. Note that the *ifunc* message is never modified in this process, as the user might want to send it to another process later.

E. Integration with Julia Language

The Julia programming language uses LLVM for its JIT ahead-of-time compilation approach. It currently has only limited tooling for static compilation and very limited support for cross-compilation.

Since Julia is a dynamic programming language there exists uncertainty about the function call-graph of a program. Julia uses an abstract-interpretation based approach where on the first call of a method with a given type-signature, the types of variables are inferred, and the call-graph is (if possible) resolved ahead of time. If a user method contains uncertainty – colloquially referred to as type-instability – a dynamic call is inserted. Dynamic calls will be resolved at runtime through a dynamic dispatch, where the runtime variables are examined, and the correct method is selected.

We take advantage of the GPU compiler infrastructure (`GPUCompiler.jl` [25], [28]) that was developed to allow the user to offload Julia functions to GPGPU accelerators. `GPUCompiler.jl` collects the statically reachable call-graph of a Julia function into one LLVM IR module. It disallows dynamic dispatch, as well as accessing of runtime managed data such as global variables or Julia runtime functions. This provides us with a subset of the Julia language that is highly performant and can be optimized well with LLVM.

We then pass LLVM IR module generated by `GPUCompiler.jl` to *Three-Chains* and use *ifunc* polling function to invoke them on the target processing unit. One problem that arose was that Julia does not currently support cross-compilation and for *Three-Chains* we compile the *ifunc* on the target systems and cache the resulting bitcode as a file.

In future work we could relax the limitation on runtime access and allow full-fledged support for global variables, memory allocation (GC) and the task runtime, but this and improved cross-compilation both necessitate improvements to the Julia compiler.

The Julia implementation builds on top of `UCX.jl`, a set of Julia bindings for UCX, and we developed new bindings for the API introduced by our framework. Besides the use of `GPUCompiler.jl` to exfiltrate LLVM IR to be used on the target machines, the workflow is identical to the code flow described in Sections III-C and III-D. This allows our framework to consume both *ifuncs* written in Julia or C, by either a Julia or C based applications.

IV. Experimental Design

A. Benchmark Design

To further characterize our *Three-Chains* framework, we created two benchmarks that use the *ifunc* API. One of these is the Target-Side Increment (TSI) *ifunc*, and the other one is the Distributed Adaptive Pointer Chasing (DAPC) miniapp. Each one of the benchmarks implements three different modes of code execution: Active Message, *ifunc* with binary code representation, and *ifunc* with bitcode code representation. The Active Message mode assumes that the binary code is already compiled and present on the target machine. The Active Message request only transfers payload data and an index pointing to the function in a pointer table. The Active Message mode is only used as an evaluation baseline to calculate the overheads of transmitting and executing *ifuncs*. The *ifunc* binary and bitcode representations are implemented as described in Sections III-B and III-C.

B. Target-Side Increment (TSI)

To measure the overheads of *ifunc* transmission, JIT compilation, and execution to a target system, we implemented a simple benchmark that measures message rate and latency for an *ifunc* that just increases a counter on a target machine. The increment function is injected to a remote machine with a 1-byte payload and 5159 bytes of bitcode. Using this benchmark, we measured the baseline overheads for moving compute and data to a remote system.

C. Distributed Adaptive Pointer Chasing (DAPC)

To provide a more realistic evaluation of the *ifunc*, we developed code that represents a new class of eXtended Remote Direct Memory Access (X-RDMA) operations. The basic idea behind X-RDMA is an implementation of a complex RDMA operation where a user injects user-defined code and data into a remote processing element on a target DPU or server. The injection operation can modify remote memory and issue new remote memory operations using *ifunc*. To demonstrate these new capabilities, we implemented the Distributed Adaptive Pointer Chasing (DAPC) miniapp. DAPC implements a workload that resembles pointer chasing across multiple distributed machines. In our implementation, the pointer table is evenly distributed across the servers and the pointer chasing part of the algorithm is implemented using the *Three-Chains* framework. Each step of the pointer chase the algorithm has the flexibility of calling itself recursively, sending itself as an *ifunc* or creating another *ifunc* with new logic. As a result, we call this algorithm adaptive as it changes what it does as it proceeds based on context and input parameters. The data (or pointer tables) are evenly spread among the server machines into shards of the same size and the entries are indexed using the server number first. These machines are waiting for incoming *ifuncs*. When the client machine is ready to start a pointer chase operation, it creates a X-RDMA *Chaser* operation using an *ifunc* with the following fields:

- *Address*: the first element to be accessed
- *Depth*: the depth of the pointer chase
- *Destination*: the node ID of the client/requester

When the *ifunc* message is ready to be sent, the client sends it to *any* of the servers. When a server receives a *Chaser ifunc*, it calls it, passing local information like the address of the pointer table. The *ifunc* determines if it is running in the server where the entry is located. If it is running in the wrong server, it forwards an X-RDMA *Chaser ifunc* to the correct server.

Once running in the right place, the *ifunc* loads the entry. If this entry is the final one in the pointer chase, the *ifunc* sends the result back to the client. If this entry is the *Address* of the next entry, then the *ifunc* determines if the entry is local or remote. If the entry is local, the *ifunc* calls itself recursively. Otherwise, the *ifunc* is forwarded to the correct server.

To return the result to the requester, we created the simple X-RDMA *ReturnResult* operation. All it does is return the result to the requester. In the case of an X-RDMA deployment on a DPU, all the *ifunc* waiting and recursive *ifunc* invocations are completely offloaded to the Arm cores on the DPU.

D. Get-Based Pointer Chasing (GBPC)

The DAPC operation has a *baseline* mode, the Get-Based Pointer Chasing (GBPC). This mode performs the pointer chase using UCX GET semantics to remotely load the entries from the servers. It uses an iterative approach to issue remote UCP GETs until the final entry of the pointer chase is found. The code for this approach is simpler than the *ifunc* code, but the client must do all the work. GBPC is implemented as a special case on the DAPC code.

E. Test Configurations

We evaluate DAPC and GBPC by performing a parameter sweep across *depth* and *number of servers*. This allows us to study the performance characteristics of *ifunc* on several regimes. For DAPC, the partitioning of the data is refined as the number of servers increases, and, thus, the fraction of cross-server communication for DAPC rises. We implemented the setup both in C and Julia. The Julia implementation was kept as close as possible to the original C implementation.

F. Testbed Platforms

For our evaluation we used two different clusters. The first one is the Ookami HPE Cluster at Stony Brook University. The Ookami cluster is based on the Apollo 80 HPE platform, and the system has 174 Fujitsu A64FX FX700 compute nodes based on the ARMv8 CPU architecture. The A64FX SoC has a total of 48 cores clocked to 1.8/2.0 GHz with 32GB HBM memory. The cluster is connected with InfiniBand network using Mellanox ConnectX-6 100 Gb/sec dual-port host channel adapters. The system is running CentOS Linux 8.4.2105 with Linux Kernel 4.18.0-305.25.1.el8_4.aarch64, GCC 11.3.0, and MOFED 5.4-3.1.0.0 The second system is the Thor cluster at the HPC Advisory Council. Thor is based on Dell PowerEdge R730 platform with Dual Socket Intel Xeon 16-core CPUs E5-2697A clocked to 2.60 GHz. The system has 36 nodes each equipped an with Arm-based (Cortex-A72) NVIDIA BlueField-2 (BF2) 100Gb/s InfiniBand DPU adapter. The system is running Red Hat Enterprise Linux 8.5 with kernel 4.18.0-348.12.2.el8_5.x86_64, GCC 8.5.0, and MOFED 5.5-1.0.3.2. On all systems we used *Three-Chains* implementation based on LLVM 13.0.1, Julia 1.8.0-beta3, and OpenUCX master (`508e766d`). It is important to point out that we got *Three-Chains* framework running on multiple other systems, including computational storage devices connected over Ethernet network. For this paper we selected what we believe are the most interesting systems.

V. Experimental Evaluation and Analysis

A. Bitcode ifunc Overhead Breakdown

One of the first questions we wanted to answer was what the overheads were of using our *Three-Chains* framework to send and run *ifuncs*. To measure these overheads, we used the kernel described in Sec. IV-B as the code we wanted to execute on the target system. We wrote a benchmark that measures the latencies of running the TSI kernel in Active Message, cached *ifunc* and uncached *ifunc* modes between a pair of systems. In addition to measuring the latency of running the TSI code, we measured the latency of sending cached *ifuncs* (26 bytes), Active Messages (33 bytes), and uncached *ifuncs* (5185 bytes). With all these measurements, we estimate what the overheads of each of the steps of sending and running an *ifunc*. The four main steps of issuing an *ifunc* to a target system are:

- Transmission. The source sends the bitcode and data using a `Put` operation.
- Lookup. The target checks if the bitcode has already been JIT compiled by LLVM and cached by *Three-Chains*. If not cached by *Three-Chains*, the target checks if the bitcode has already been JIT compiled by LLVM.
- JIT compilation. If the bitcode is not already JIT compiled and cached, the target's LLVM JITs the bitcode and caches the binary generated. This step performs the dynamic linking of dependencies.
- Execution. The target runs the JIT compiled binary.

We used the following equations to generate the overhead tables below. In these equations, we abbreviated Active Message to *(*AM), cached *ifunc* to *C*, and uncached *ifunc* to *U*. The *total*, *trans*, *JIT*, and *L+E* abbreviations correspond to total, transmission, JIT, and lookup-and-execution overheads, respectively.

$$U_{total} = U_{trans} + U_{JIT} + U_{L+E} \quad (1)$$

$$C_{total} = C_{trans} + C_{L+E} \quad (2)$$

$$AM_{total} = AM_{trans} + AM_{L+E} \quad (3)$$

It is important to point out that LLVM's ORC-JIT caches observed code symbols. Because of this internal caching, we observed no overhead for JIT compilation in the benchmarks that re-send the same *ifunc*. When *Three-Chains* invokes JIT operation for the *ifunc*, LLVM has to do minimal work since it looks up the *ifunc* from previous JIT invocations. To get an accurate estimate of JIT overheads without LLMV ORC caching we created a separate benchmark to measure the JIT compilation duration without the caching and listed this time in the table (JIT), but did not add it to the total time.

On Table I, we list how long each step takes for all three modes when tested on Ookami machines. The Active Message and cached bitcode columns do not include a value for JIT compilation because these operations do not need to perform JIT compilation. We estimated that JIT compilation for the

Stage	Active Message	Uncached Bitcode	Cached Bitcode
Lookup+Exec	0.08 µs	0.10 µs	0.05 µs
JIT	N/A	(6.59 ms)	N/A
Transmission	2.50 µs	5.02 µs	2.62 µs
Total	2.58 µs	5.12 µs	2.67 µs

TABLE I: Ookami TSI overhead breakdown

Stage	Active Message	JIT compiled Bitcode	Cached Bitcode
Lookup+Exec	0.01 µs	0.04 µs	0.01 µs
JIT	N/A	(4.50 ms)	N/A
Transmission	1.87 µs	3.45 µs	1.85 µs
Total	1.88 µs	3.49 µs	1.86 µs

TABLE II: Thor BF2 TSI overhead breakdown

uncached bitcode took approximately 6.59 ms, completely dominating the transmission and execution times. We can also observe the toll sending bitcode takes on transmission time. The cached *ifunc* message is just 26B, while the bitcode sent with the uncached version adds 5,159 bytes, making the message 5,185 bytes long. Sending the uncached bitcode takes almost double the time it takes to send the cached *ifunc*, taking 2.67 µs and 5.12 µs, respectively. Overall, the uncached bitcode takes 91% longer to get sent and executed.

On the Ookami system, the cached *ifuncs* perform pretty well, with message transmission and execution taking about 3% longer than Active Messages.

Table II shows the time breakdown for each mode when tested between two BF2 systems from the Thor cluster. In this system, JIT compilation the TSI code takes 4.50 ms. Sending the larger uncached *ifunc* message takes 86% longer than sending the cached version, and the uncached version takes 88% longer to finish end-to-end.

In this configuration, the Active Message and cached *ifunc* take essentially the same time to execute once received. Since the cached *ifunc* message is slightly smaller than the Active Message (26B versus 33B), the transmission time favors the cached *ifunc* (about 1% faster).

Table III shows the data collected and calculated from the Thor Xeon systems. TSI JIT compilation takes 0.83 ms. In this configuration, not sending the *ifunc* code provides an overall speedup of 2.3×. These systems launch the cached JIT compiled code 2% faster than they launch the Active Message code.

Our main take away from these experiments is that JIT compilation incurs an expensive one-time cost. Since our TSI kernel only increments a memory value, it executes so quickly, that its execution time is negligible. The dominating cost of running this simple *ifunc* is sending it to the target.

In this section, we do not include binary *ifunc* numbers because the effect of caching on execution is negligible since the code arrives ready to be executed. The only benefit of caching for binary *ifuncs* is on the transmission side, where the uncached version of TSI is 75 bytes compared to the 26-byte cached version.

Stage	Active Message	JIT compiled Bitcode	Cached Bitcode
Lookup+Exec	0.01 µs	0.01 µs	0.02 µs
JIT	N/A	(0.83 ms)	N/A
Transmission	1.55 µs	3.58 µs	1.51 µs
Total	1.56 µs	3.59 µs	1.53 µs

TABLE III: Thor Xeon TSI overhead breakdown

Method	Latency	Speedup	Message Rate	Speedup
Active Message	2.58 µs	-3.29%	1,320,000 msg/sec	26.44%
Cached Bitcode	2.67 µs		1,669,000 msg/sec	
Uncached Bitcode	5.12 µs	91.39%	405,300 msg/sec	311.79%
Cached Bitcode	2.67 µs		1,669,000 msg/sec	

TABLE IV: Ookami TSI latencies and message rates

Method	Latency	Speedup	Message Rate	Speedup
Active Message	1.88 µs	0.86%	974,000 msg/sec	34.60%
Cached Bitcode	1.87 µs		1,311,000 msg/sec	
Uncached Bitcode	3.49 µs	87.73%	417,300 msg/sec	214.16%
Cached Bitcode	1.87 µs		1,311,000 msg/sec	

TABLE V: Thor BF2 TSI latencies and message rates

B. Bitcode ifunc Latency and Message Rate Overheads

As part of investigating the overheads of our bitcode *ifuncs*, we also tested the message rate achievable when sending and executing as many TSI kernels as possible from one machine to another. We tested these using Active Message, cached *ifunc* and uncached *ifunc*. We listed these results in the following tables, alongside the latency results from subsection V-A.

Table IV, Table V, and Table VI show the results of the latency and message rate of executing the TSI kernel using Active Messages, uncached *ifunc* and cached *ifunc*. Caching the JIT compiled code drastically improves latency and message rate for all systems tested. Depending on the configuration, we observed latency reductions of 84%-144% and message rate improvements of 214%-311%.

When comparing cached *ifuncs* with Active Messages, *ifuncs* consistently exhibit a higher message rate (8%-34%). In the Thor configurations (Table V and Table VI), cached *ifunc* latency is better than that of Active Message. Only in the Ookami configuration (Table IV) we observe a worse latency for *ifuncs*.

Overall, cached *ifuncs* exhibit between slightly worse (up to 3% higher latency) to significantly better performance (up to 34% higher message rate) than Active Messages. *ifuncs* provide the dynamic programmability that Active Messages cannot.

C. Pointer Chase Performance Depth Sweep

To better understand the performance of *ifuncs* in a more realistic scenario, we tested the Distributed Adaptive Pointer Chasing (DAPC) *ifunc* implementation (described in

Method	Latency	Speedup	Message Rate	Speedup
Active Message	1.56 µs	2.30%	6,754,000 msg/sec	8.11%
Cached Bitcode	1.53 µs		7,302,000 msg/sec	
Uncached Bitcode	3.59 µs	135.54%	2,037,000 msg/sec	258.47%
Cached Bitcode	1.53 µs		7,302,000 msg/sec	

TABLE VI: Thor Xeon TSI latencies and message rates

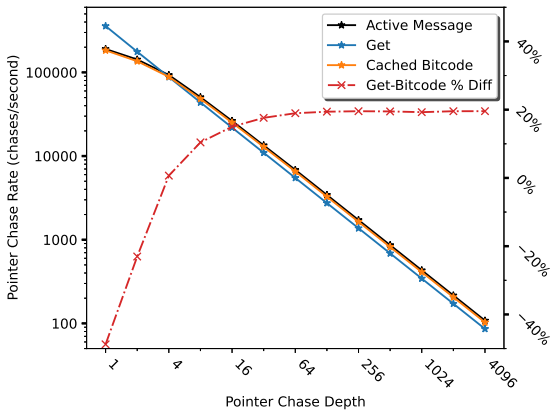

Fig. 5: Thor 32-Server; **C/C++** (Xeon Client and BF2 Servers): Distributed Adaptive Pointer Chasing (DAPC)

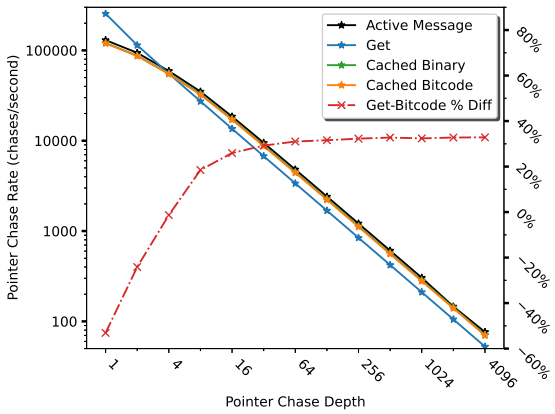

Fig. 6: Ookami 64-Server; **C/C++**: Distributed Adaptive Pointer Chasing (DAPC)

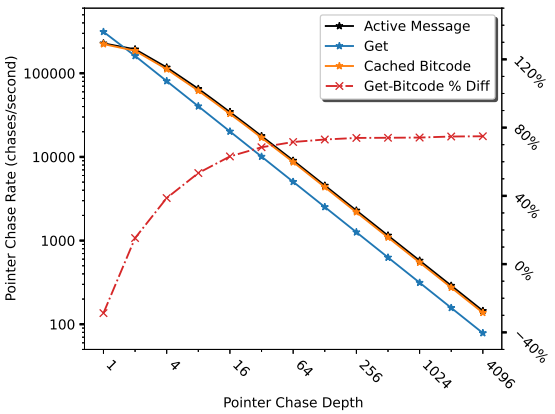

Fig. 7: Thor 16-Server; **C/C++** (Xeon Client and Servers): Distributed Adaptive Pointer Chasing (DAPC)

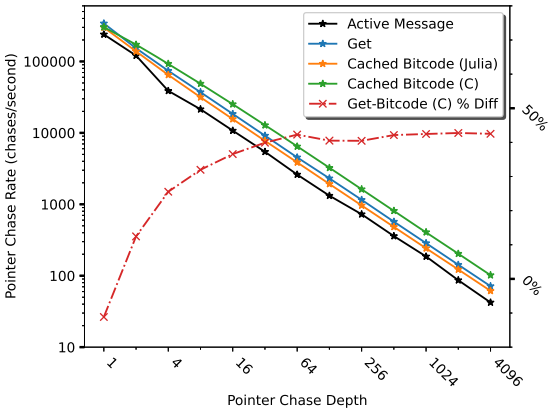

Fig. 8: Thor 32-Server; **Julia** (Xeon Client and BF2 Servers): Distributed Adaptive Pointer Chasing (DAPC)

Sec. IV-C) and compared it to the Active Message implementation and to Get-Based Pointer Chasing (GBPC). We ran a range of pointer chasing depths, from 1, scaling in powers of 2, all the way to 4096. For these sweep tests, we ran on the Ookami and Thor clusters using a variety of configurations. For the sake of space, we only present a very small subset of the results obtained, but these are representative of the trends we observed across all configurations tested.

Figures 5, 6, 7, and 8 show how many pointer chases can complete per second for all our system and combination of C/C++ and Julia implementations. For the DAPC C-based implementation we observe nearly identical behavior. The cached bitcode *ifunc* implementation performs on pair with Active Messages and cached binary implementation on Arm-based platforms. The bitcode implementation also outperforms GBPC (Get) implementation by up to 20% on Thor with BF2 Servers (Figure 5), by up to 30% on the Ookami system (Figure 6), and by up to 75% on the Thor with Xeon servers (Figure 7).

As depth grows, the gap between each of the lines stabilizes to a constant value. The gap between each line pair represents the constant speed difference between pointer lookup and traversing using each one of the methods.

D. Pointer Chase Performance Scalability

We used the data collected to understand what happens to pointer chase rate when scaling out to more servers. We took the 4096-depth datapoints for all modes tested and plotted them against number of servers.

Figures 9, 10, 11, and 12 show how pointer chase rate scales with the number of servers. All three C/C++ configurations shown exhibit similar trends. The GBPC (Get) line remains relatively flat regardless of the number of servers. This is expected because each lookup performs a get operation that returns to the client and the number of lookups is tied to the depth. For Active Message and *ifunc* bitcode, the code only issues a network operation when the value of the next lookup resides in a different server. Due to this, the more servers the system has, the more network operations will occur. These network operations are slower than the recursive call used when the next lookup resides locally, leading to a lower chase

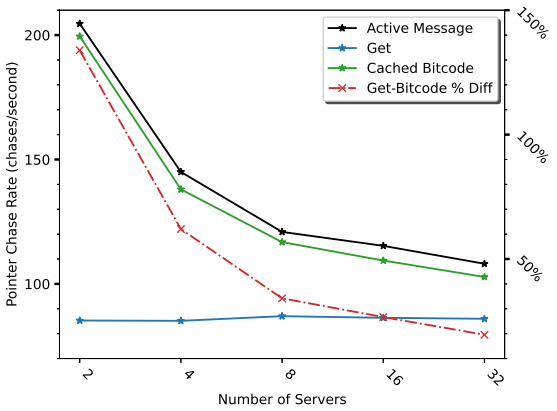

Fig. 9: Thor 4096-Chase-Depth; **C/C++** (Xeon Client and BF2 Servers): Distributed Adaptive Pointer Chasing (DAPC)

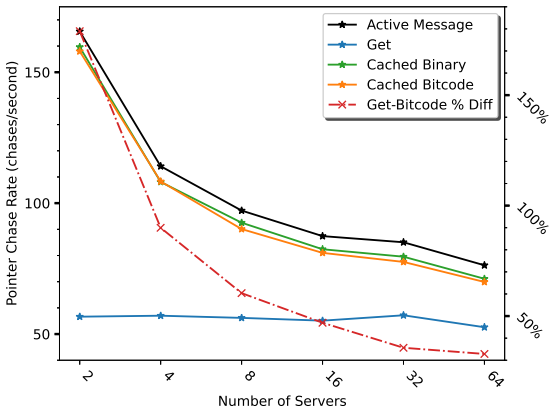

Fig. 10: Ookami 4096-Chase-Depth; **C/C++**: Distributed Adaptive Pointer Chasing (DAPC)

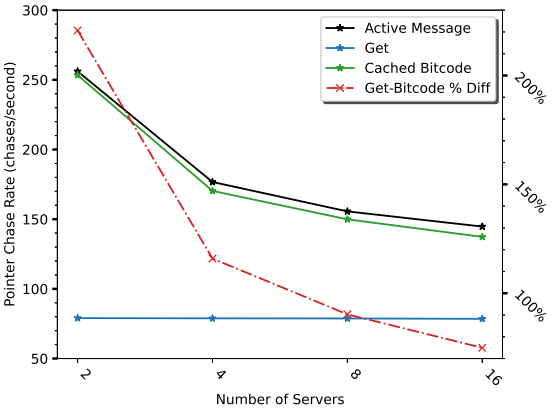

Fig. 11: Thor 4096-Chase-Depth; **C/C++** (Xeon Client and Servers): Distributed Adaptive Pointer Chasing (DAPC)

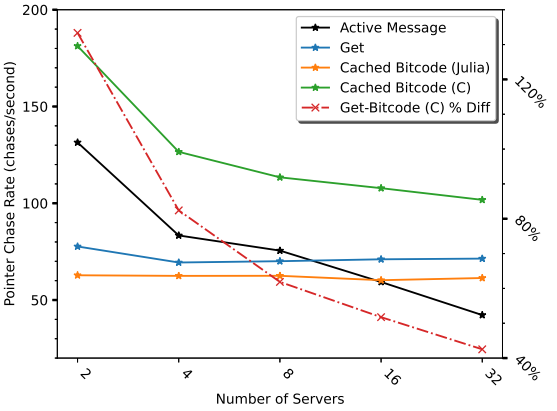

Fig. 12: Thor 4096-Chase-Depth; **Julia** (Xeon Client and BF2 Servers): Distributed Adaptive Pointer Chasing (DAPC)

rate when there are more servers. As we saw previously, the Active Message mode performs between 3% and 7% better than the *ifunc* Bitcode mode.

The Julia configuration (Figure 12) has similar behavior to the C/C++ configurations with some differences worth highlighting. In particular the performance of bitcode generated from Julia is surprisingly constant across the scaling experiment and warrants further investigation. In addition, Julia driving the bitcode generated from C is demonstrating excellent performance.

VI. Conclusions

In this work we described design of the *Three-Chains* framework and demonstrated a performance evaluation using microbenchmarks and the X-RDMA pointer chase application. The framework provides the infrastructure for moving code in binary or LLVM bitcode representation. The framework was evaluated on the Ookami Fujitsu system and the Thor system with Intel CPUs and BlueField-2 DPUs. We showed that our framework outperforms a classical RDMA-GET pointer chase implementation. We confirmed that the overhead of the framework is comparable to that of Active Message semantics while providing a greater level of flexibility and programmability. We also demonstrated the framework's integration with C/C++ and Julia-based applications. While the performance for our Julia implementation is not yet optimal, this highlights a couple of possibilities. Firstly, code written in Julia can make use of high-performance *ifuncs* generated from C to allow for dynamic code execution with high performance. Secondly, Julia as an *ifunc* will require a deeper investigation into the performance issues, but can enable the usage of high-level language features, Julia's code generation capabilities as well as use of the rich libraries available in Julia. In particular, we could conceive the usage of machine-learning and online-statistics libraries written in Julia for data processing on DPUs.

The *Three-Chains* framework and all the benchmarks used for this paper are open source and available on GitHub [29].

VII. Acknowledgments

The authors would like to thank the LANL for their continued support of this project. In addition, we would like thank

Jon Hermes, Alexandre Ferreira, and Eric Van Hensbergen for their review of the paper and code.

We thank Gilad Shainer, David Cho, and HPC Advisory Council for providing access to Thor system. The authors would also thank Stony Brook Research Computing and Cyberinfrastructure, and the Institute for Advanced Computational Science at Stony Brook University for access to the Ookami computing system, which was made possible by a $5M NSF grant (#1927880). We gratefully acknowledge funding from NSF (grants OAC-1835443, OAC-2103804, AGS-1835860, and AGS-1835881), DARPA under agreement number HR0011-20-9-0016 (PaPPa). This research was made possible by the generosity of Eric and Wendy Schmidt by recommendation of the Schmidt Futures program, by the Paul G. Allen Family Foundation, Charles Trimble, Audi Environmental Foundation. This material is based upon work supported by the Department of Energy, National Nuclear Security Administration under Award Number DE-NA0003965. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation herein.

REFERENCES

[1] "Compute express link," https://cxl.com, 2022.
[2] "Ccix consortium," https://www.ccixconsortium.com, 2022.
[3] "Infrastructure programmer development kit," https://ipdk.io, 2022.
[4] "Open vswitch," https://www.openvswitch.org, 2022.
[5] P. Bosshart *et al.*, "P4: Programming protocol-independent packet processors," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 87–95, 2014.
[6] "Nvidia doca," https://developer.nvidia.com/networking/doca, 2022.
[7] "Snort," https://www.snort.org, 2022.
[8] "Rdma core," https://github.com/linux-rdma/rdma-core, 2022.
[9] "Data plane development kit," https://www.dpdk.org, 2022.
[10] "Arm confidential compute architecture," https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture, 2022.
[11] "Veracruz: privacy-preserving collaborative compute," https://github.com/veracruz-project/veracruz, 2022.
[12] "Project oak," https://github.com/project-oak/oak, 2022.
[13] M. Grodowitz, L. E. Peña, C. Dunham, D. Zhong, P. Shamis, and S. Poole, "Two-chains: High performance framework for function injection and execution," in *(To appear in) 2021 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2021, https://arxiv.org/abs/2108.02253.
[14] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis and transformation," in *CGO*, San Jose, CA, USA, Mar 2004, pp. 75–88.
[15] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *SIAM review*, vol. 59, no. 1, pp. 65–98, 2017. Available: https://doi.org/10.1137/141000671
[16] Y. Chen, Y. Lu, and J. Shu, "Scalable rdma rpc on reliable connection with efficient resource sharing," in *Proceedings of the Fourteenth EuroSys Conference 2019*, 2019, pp. 1–14.
[17] A. Kalia, M. Kaminsky, and D. Andersen, "Datacenter {RPCs} can be general and fast," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, 2019, pp. 1–16.
[18] T. Li, H. Shi, and X. Lu, "Hatrpc: hint-accelerated thrift rpc over rdma," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2021, pp. 1–14.
[19] L. E. Peña, W. Lu, P. Shamis, and S. Poole, "Ucx programming interface for remote function injection and invocation," *arXiv preprint arXiv:2110.06292*, 2021.
[20] D. Bonachea and P. H. Hargrove, "Gasnet-ex: A high-performance, portable communication library for exascale," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2018, pp. 138–158, https://bytebucket.org/berkeleylab/upcxx/wiki/pubs/gasnet-ex-lcpc18-6da6911-tech.pdf.
[21] M. Marty *et al.*, "Snap: A microkernel approach to host networking," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 399–413, http://pages.cs.wisc.edu/~yxy/cs839-s20/papers/snap.pdf. Available: https://doi.org/10.1145/3341301.3359657
[22] B. Acun *et al.*, "Parallel programming with migratable objects: Charm++ in practice," in *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2014, pp. 647–658. Available: http://charm.cs.illinois.edu/newPapers/14-07/paper.pdf
[23] J. Klinkenberg, P. Samfass, M. Bader, C. Terboven, and M. S. Müller, "Chameleon: Reactive load balancing for hybrid mpi+openmp task-parallel applications," *Journal of Parallel and Distributed Computing*, vol. 138, pp. 55 – 64, 2020. Available: http://www.sciencedirect.com/science/article/pii/S0743731519305180
[24] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, "Farm: Fast remote memory," in *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, 2014, pp. 401–414.
[25] T. Besard, V. Churavy, A. Edelman, and B. D. Sutter, "Rapid software prototyping for heterogeneous and distributed platforms," *Advances in engineering software*, vol. 132, pp. 29–46, Jun. 2019.
[26] A. Rizvi and K. C. Hale, "A look at communication-intensive performance in julia," 2021.
[27] P. Shamis *et al.*, "Ucx: an open source framework for hpc network apis and beyond," in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. IEEE, 2015, pp. 40–43.
[28] T. Besard, C. Foket, and B. De Sutter, "Effective extensible programming: Unleashing julia on GPUs," *IEEE Trans. Parallel Distrib. Syst.*, no. 4, pp. 827–841, Dec. 2017.
[29] "Three-Chains GitHub," https://github.com/openucx/three-chains, 2022.