New Directions in Cloud Programming

Alvin Cheung Natacha Crooks Joseph M. Hellerstein Mae Milano
{akcheung,ncrooks,hellerstein,mpmilano}@berkeley.edu
UC Berkeley

ABSTRACT

Nearly twenty years after the launch of AWS, it remains difficult for most developers to harness the enormous potential of the cloud. In this paper we lay out an agenda for a new generation of cloud programming research aimed at bringing research ideas to programmers in an evolutionary fashion. Key to our approach is a separation of distributed programs into a PACT of four facets: Program semantics, Availablity, Consistency and Targets of optimization. We propose to migrate developers gradually to PACT programming by lifting familiar code into our more declarative level of abstraction. We then propose a multi-stage compiler that emits human-readable code at each stage that can be hand-tuned by developers seeking more control. Our agenda raises numerous research challenges across multiple areas including language design, query optimization, transactions, distributed consistency, compilers and program synthesis.

1 INTRODUCTION

It is easy to take the public clouds for granted, but we have barely scratched the surface of their potential. These are the largest computing platforms ever assembled, and among the easiest to access. Prior generations of architectural revolutions led to programming models that unlocked their potential: minicomputers led to C and the UNIX shell, personal computers to graphical "low-code" programming via LabView and Hypercard, smartphones to Android and Swift. To date, the cloud has yet to inspire a programming environment that exposes the inherent power of the platform.

Initial commercial efforts at a programmable cloud have started to take wing recently in the form of "serverless" Functions-as-a-Service. FaaS offerings allow developers to write sequential code and upload it to the cloud, where it is executed in an independent, replicated fashion at whatever scale of workload it attracts. First-generation FaaS systems have well-documented limitations [46], which are being addressed by newer prototypes with more advanced FaaS designs (e.g., [8, 81]). But fundamentally, even "FaaS done right" is a low-level assembly language for the cloud, a simple infrastructure for launching sequential code, a UDF framework without a programming model to host it.

As cloud programming matures, it seems inevitable that it will depart from traditional sequential programming. The cloud is a massive, globe-spanning distributed computer made up of heterogeneous multicore machines. Parallelism abounds at all scales, and the distributed systems challenges of non-deterministic network interleavings and partial failures exist at most of those scales. Creative programmers are held back by the need to account for these complexities using legacy sequential programming models originally designed for single-processor machines.

We need a programming environment that addresses these complexities directly, but without requiring programmers to radically change behavior. The next generation of technology should *evolutionize* the way developers program: allow them to address distributed concerns gradually, working with the assistance of new automation technologies, but retaining the ability to manually override automated decisions over time.

1.1 A New PACT for Cloud Programming

Moving forward, we envision decoupling cloud programming into four separate concerns, each with an independent language facet: Program semantics, Availability, Consistency and Targets for optimization (PACT).

Program Semantics: Lift and Support. A programmer's primary goal is to specify the intended functionality of their program. Few programmers can correctly write down their program semantics in a sequential language while also accounting for parallel interleavings, message reordering, partial failures and dynamically autoscaling deployment. This kind of "hand-crafted" distributed programming is akin to assembly language for the cloud.

Declarative specifications offer a very different solution, shielding the programmer from implementation and deployment details. Declarative programming environments for distributed computing have emerged in academia and industry over the past decade [11, 41, 65], but adoption of these "revolutionary" approaches has been limited. Moving forward, we advocate an evolutionary *Lift and Support* approach: given a program specification written in a familiar style, automatically lift as much as possible to a higher-level declarative

11th Annual Conference on Innovative Data Systems Research (CIDR '21), January 10-13, 2021, Chaminade, USA

Intermediate Representation (IR) used by the compiler, and encapsulate what remains in UDFs (i.e., FaaS Functions).

Availability Specification. Availability is one of the key advantages of the cloud. Cloud vendors offer hardware and networking to deploy services redundantly across multiple relatively independent failure domains. Traditionally, though, developers have had to craft custom solutions to ensure that their code and deployments take advantage of this redundancy efficiently and correctly. Availability protocols are frequently interleaved into program logic in ways that make them tricky to test and evolve. We envision a declarative facet here as well, allowing programmers to specify the availability they wish to offer independent from their program semantics. A compiler stage must then synthesize code to provide that availability guarantee efficiently.

Consistency Guarantees. Many of the hardest challenges of distributed programming involve consistency guarantees. "Sophisticated" distributed programs are often salted with programmer-designed mechanisms to maintain consistency. We advocate a programming environment that separates consistency specifications into a first-class program facet, separated from the basic functionality. A compiler stage can then generate custom code to guarantee that clients see the desired consistency subject to availability guarantees. Disentangling consistency invariants from code makes two things explicit: the desired common-case sequential semantics, and the relaxations of those semantics that are to be tolerated in the distributed setting. This faceting makes it easier for compilers to guarantee correctness and achieve efficiency, it allows enforcement across compositions of multiple distributed libraries, and allows developers to easily understand and modify the consistency guarantees of their code.

Targets for Dynamic Optimization. In the modern cloud, code is not just compiled; it must be deployed as a well-configured service across multiple machines. It also must be able to redeploy itself dynamically—*autoscale*—to work efficiently as workloads grow and shrink by orders of magnitude, from a single multicore box to a datacenter to the globe. We believe cloud frameworks inevitably must lighten this load for general-purpose developers. We envision an environment where programmers can specify multi-objective performance targets for execution, e.g., tradeoffs between billing costs, latency and availability. From there, a number of implementation and deployment decisions must be made. This includes compilation logic like choosing the right data structures and algorithms for "local," sequential code fragments, as well as protocols for message-passing for distributed functionality. It also includes the partitioning, replication and placement of code and data across machines with potentially heterogeneous resources. Finally, the binary executables we generate need to include dynamic runtime logic that monitors and adapts the deployment in the face of shifting workloads.

For all these facets, we envision a *gradual* approach to bring programmers on board in an evolutionary manner. Today's developers should be able to get initial success by writing simple familiar programs, and entrusting everything else to a compiler. In turn, this compiler should generate human-centric code in well-documented internal languages, suitable for eventual refinement by programmers. As a start, we believe an initial compiler should be able to achieve performance and cost at the level of FaaS offerings that users tolerate today [46], with the full functionality of PACT programming. Programmers can then improve the generated programs incrementally by modifying the lower-level facets or "hinting" the compiler via constraints.

1.2 Sources of Inspiration and Confidence

Our goals for the next generation of cloud programming are ambitious, but work over the last decade gives us confidence that we can take significant strides in this direction. A number of ideas from the past decade inform our approach:

Monotonic Distributed Programming. Monotonicity—the property that a program's output grows with its input—has emerged as a key foundation for efficient, available distributed programs [45]. The roots of this idea go back to Helland and Campbell's crystallization of coordination-free distributed design patterns as ACID 2.0: Associative, Commutative, Idempotent and Distributed [43]. Subsequently, CRDTs [78] were proposed as data types with ACI methods, observing that the ACI properties are those of join-semilattices: algebraic structures that grow monotonically. The connection between monotonicity and order-independence turns out to be fundamental. The CALM Theorem [15, 44] proved that programs produce deterministic outcomes without coordination *if and only if* they are monotonic. Hence monotonic code can run coordination-free without any need for locking, barriers, commit, consensus, etc. At the same time, our Bloom language [11, 28] adopted declarative logic programming for distributed computing, with a focus on a monotonic core, and coordination only for non-monotone expressions. Various monotonic distributed language proposals have followed [53, 63, 68]. Monotonic design patterns have led to clean versions of complex distributed applications like collaborative editing [83], and high-performance, consistency-rich autoscaling systems like the Anna KVS [85].

Dataflow and Reactive Programming. Much of the code in a distributed application involves data that flows between machines, and event-handling at endpoints. Distributed dataflow is a notable success story in parallel and distributed computing, from its roots in 1980s parallel databases [35] through to recent work on richer models like Timely Dataflow [61] and efforts to autoscale dataflow in the cloud [51]. For event handling, reactive programming libraries like React.js [23] and Rx [62] provide a different dataflow model for handling events and mutating state. Given these successes and our experience with dataflow backends for low-latency settings [9, 58, 59] we are optimistic that a combination of dataflow and reactivity would provide a good general-purpose runtime target for services and protocols in the cloud. We are also encouraged by the general popularity of libraries like Spark and React.js—evidence that advanced programmers will be willing to customize low-level IR code in that style.

Faceted Languages. The success of LLVM [54] has popularized the idea of multi-stage compilation with explicit internal representation (IR) languages. We are inspired by the success of faceted languages and separation of concerns in systems design, with examples such as the model-view-controller design pattern for building user interfaces [38], the three-tier architecture for web applications [36, 75], and domain-specific languages such as Halide for image processing pipelines [71]. Dissecting an application into facets enables the compiler designer and runtime developer to choose different algorithms to translate and execute different parts of the application. For instance, Halide decouples algorithm specification from execution strategy, but keeps both as syntactic constructs for either programmer control or compiler autotuning. This decoupling has led Halide to outperform expert hand-tuned code that took far longer to develop, and its outputs are now used in commercial image processing software. Image processing is particularly inspiring, given its requirements for highly optimized code including parallelism and locality in combinations of CPUs and GPUs.

Verified Lifting. Program synthesis is one of the most influential and promising practical breakthroughs in modern programming systems research [42]. Verified lifting is a technique we developed that uses program synthesis to formulate code translation as *code search*. We have applied verified lifting to translate code across different domains, e.g., translating imperative Java to declarative SQL [26] and functional Spark [4, 5], translating imperative C to CUDA kernels to Halide [6] and to hardware description languages [80]. Our translated Halide code is now shipping in commercial products. Verified Lifting cannot handle arbitrary sequential code, but our Lift and Support approach should allow us to use it as a powerful programmer aid.

Client-Centric and Mixed Consistency. Within the enormous literature on consistency and isolation, two recent thrusts are of particular note here. Our recent work on *client-centric* consistency steps away from traditional low-level histories to offer guarantees about what could be observed by a calling client. This has led to a new understanding of the

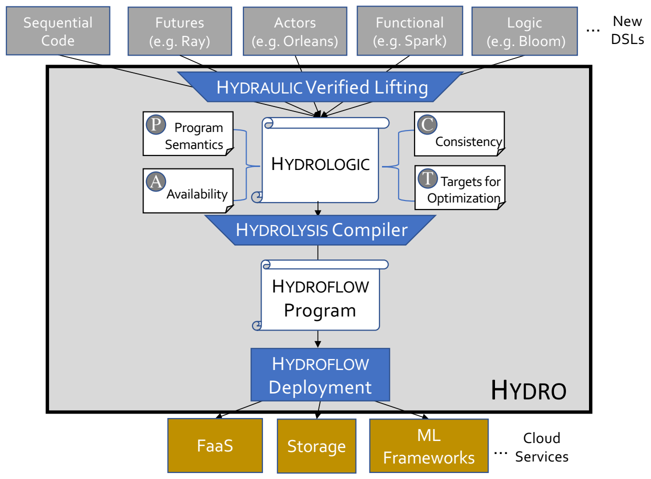


Figure 1: The Hydro stack.

connections between transactional isolation and distributed consistency guarantees [29]. Another theme across a number of our recent results is the *composition* of services that offer different consistency guarantees [31, 67, 68]. The composition of multiple services with different consistency guarantees is a signature of modern cloud computing that needs to be brought more explicitly into programming frameworks.

1.3 Outline

In this paper we elaborate on our vision for cloud-centric programming technologies. We are exploring our ideas by building a new language stack called Hydro that we introduce next. The development of Hydro is part of our methodology, but we believe that the problems we are addressing can inform other efforts towards a more programmable cloud.

In Section 2 we provide a high-level overview of the Hydro stack and a scenario that we use as a running example in the paper. In Section 3 we present our ideas for HydroLogic's program semantics facet, and in Section 4 we back up and explore the challenge of lifting from multiple distributed programming paradigms into HydroLogic. Section 5 discusses the data model of HydroLogic and our ability to use program synthesis to automatically choose data representations to meet performance goals. Section 6 sketches our first explicitly distributed language facet: control over Availability, while Section 7 covers the challenges in the correlated facet of Consistency. In Section 8 we discuss lowering HydroLogic to the corresponding HydroFlow algebra on a single node. Finally, in Section 9 we address the distributed aspects of optimizing and deploying HydroFlow, subject to multi-objective goals for cost and performance.

2 HYDRO'S LANGUAGES

Hydro consists of a faceted, three-stage compiler that takes programs in one or more distributed DSLs and compiles them to run on a low-level, autoscaling distributed deployment of local programs in the Hydroflow runtime.

To bring a distributed library or DSL to the Hydro platform, we need to lift it to Hydro's declarative Intermediate Representation (IR) language—HydroLogic. Hence our first stage is the Hydraulic Verified Lifting facility (Section 4), which automates that lifting as much as it can, encapsulating whatever logic remains in UDFs.

2.1 The Declarative Layer: HydroLogic

The HydroLogic IR (Section 3) is itself multifaceted as seen in Figure 1. The core of the IR allows *Program Semantics* Ⓟ to be captured in a declarative fashion, without recourse to implementation details regarding deployment or other physical optimizations. For fragments of program logic that fail to lift to HydroLogic, the language supports legacy sequential code via UDFs executed inline or asynchronously in a FaaS style. The *Availability* facet Ⓐ allows a programmer to ensure that each network endpoint in the application can remain available in the face of f failures across specified failure domains (VMs, data centers, availability zones, etc.) In the *Consistency* facet Ⓒ we allow users to specify their desires for replica consistency and transactional properties; we use the term "consistency" to cover both. Specifically, we allow service endpoints to specify the consistency semantics that senders can expect to see. The final high-level *Targets for Optimization* facet Ⓣ allows the developer to specify multiple objectives for performance, including latency distributions, billing costs, downtime tolerance, etc.

Given a specification of the above facets, we have enough information to compile an executable cloud deployment. HydroLogic's first three facets identify a finite space of satisfying distributed programs, and the fourth provides performance objectives for optimization in that space.

2.2 Compilation of HydroLogic

The next challenge is to compile a HydroLogic specification into an executable program deployable in the cloud. Rather than generating binaries to be deployed directly on different cloud platforms, we will instead compile HydroLogic specifications into programs written against APIs exposed by the Hydroflow runtime (to be discussed in Section 2.3). Doing so allows experienced developers to fine-tune different aspects of a deployment while simplifying code generation. We are currently designing the Hydroflow APIs; we envision them to cover different primitives that can be used to implement the HydroLogic facets, such as:

- The choice of data structures for collection types and concrete physical implementations (e.g., join algorithm) to implement the semantics facet running as a local data flow on a single node.
- Partitioning ("sharding") strategies for data and flows among multiple nodes, based on the data model facet.
- Mechanisms that together form the isolation and replica consistency protocols specific to the application.
- Scheduling and coordination primitives to execute data flows across multiple nodes, such as spawning and terminating Hydroflow threads on VMs.
- Monitoring hooks inserted into each local data flow to trigger adaptive reoptimization as needed during execution.

These primitives cover a lot of design ground, and we are still exploring their design. A natural initial approach is to provide a finite set of choices as different API calls, and combine API calls into libraries that provide similar functionalities for the compiler or developer to invoke (e.g., different data partitioning mechanisms). We imagine that the Hydrolysis compiler will analyze multiple facets to determine which APIs to invoke for a given application, for instance combining the program semantics and targets for optimization facets to determine which data structures and physical implementations to use. In subsequent sections we illustrate one or more choices for each. Readers familiar with these areas will hopefully begin to see the larger optimization space we envision, by substituting in prior work in databases, dataflow systems, and distributed storage. Note also that many of these features can be bootstrapped in HydroLogic, e.g., by adapting prior work on distributed logic for transaction and consensus protocols [9, 10], query compilation [27], and distributed data structures [59].

We are designing a compiler called Hydrolysis to take a HydroLogic specification and generate programs to be executed on the Hydroflow runtime. As mentioned, our initial goal for Hydrolysis is to guarantee correctness while meeting the performance of deployments on commercial FaaS pipelines. Our next goal is to explore different compilation strategies for Hydrolysis, ranging from syntax-directed, cost-driven translation (similar to a typical SQL optimizer), to utilizing program synthesis and machine learning for compilation. The faceted design of HydroLogic makes it easy to explore this space: each facet can be compiled independently using (a combination of) different strategies, and the generated code can then be combined and further optimized with low-level, whole program transformation passes.

2.3 The Executable Layer: Hydroflow

The Hydroflow runtime (Section 8) is a strongly-typed single-node flow runtime implemented in Rust. It subsumes

Simple COVID-19 Tracker Pseudocode

```
table people(pid, country, contacts, covid, vaccinated)
var vaccine_count

def add_person(pid):
  people.add(pid, {}, false, false)

def add_contact(id1, id2):
  people[id1].contacts.add(people[id2])
  people[id2].contacts.add(people[id1])

# transitive closure of contacts
def trace(start_id):
  return people.contacts*

def diagnosed(pid):
  people[pid].covid = true
  alert(p for p in trace(pid))

def likelihood(pid):
  return covid_predict(people[pid])

def vaccinate(pid):
  people[pid].vaccinated = True
  vaccine_count--
```

Figure 2: A simple COVID-19 tracking application.

ideas from both the dataflow engines common in data processing, and the reactive programming engines more commonly used in event-driven UI programming. HYDROFLOW provides an event-driven, flow-based execution model, with operators that produce and consume various types including collections (sets, relations, tensors, etc.), lattices (counters, vector clocks, etc.) and traditional mutable scalar variables.

HYDROFLOW executes within a transducer network [15] (Section 3.1). This event model allows for very high efficiency: as in the high-performance Anna KVS [85], all state is thread local and HYDROFLOW does not require any locks, atomics, or other coordination for its own execution. Another advantage of the transducer model is the clean temporal semantics. As discussed in Section 3.1, all state updates are deferred to end-of-tick and applied atomically, so that handlers do not experience race conditions within a tick. Non-deterministic ordering arises only via explicit asynchronous messages.

2.4 A Running Example

As a running example, we start with a simplified backend for a COVID-19 tracking app. We assume a front-end application that generates pairwise contact traces, allows medical organizations to report positive diagnoses, and alerts users to the risk of infection. Sequential pseudocode is in Figure 2.

The application logic starts with basic code to add an entry to the set `people`. The `add_contact` function records the arrival of a new contact pair in the `contacts` list of both `people` involved. The utility function `trace` returns the transitive closure of a person's contacts. Upon diagnosis, the `diagnosed` function updates the state and sends an alert to the app for every person transitively in contact. Next up is the `likelihood` function, which allows recipients of an alert to synchronously invoke an imported black-box ML model `covid_predict`, which returns a likelihood that the virus propagated to them through the contact graph.

Our final function allocates a vaccine from inventory to a particular person. We will revisit this example shortly, lifted into HYDROLOGIC.

3 THE PROGRAM SEMANTICS FACET

In HYDRO, our "evolutionary" approach is to accept programs written in sequential code or legacy distributed frameworks like actors and futures. In a best-effort fashion, we lift these programs into a higher-level Internal Representation (IR) language called HYDROLOGIC. Over time we envision a desire among some programmers for a more "revolutionary" approach involving user-friendly syntax that maps fairly directly to HYDROLOGIC or HYDROFLOW and their more optimizable constructs. The IR syntax we present here is preliminary and designed for exposition; we leave the full design of HYDROLOGIC syntax for future work.

We want our IR to be a target that is *optimizable*, *general* and *programmer-friendly*. In the next few sections we introduce the IR and the ways in which it is amenable to distributed optimizations. In Appendix A we demonstrate generality by showing how various distributed computing models can compile to HYDROLOGIC.

Figure 3 shows our running example in a Pythonic version of HYDROLOGIC. The data model is presented in lines 1 through 5, discussed further in Section 5. The program semantics (Section 3.1) are specified in lines 7 through 35, with the consistency facet (Section 7) declared inline for the handler at Line 31 that does not use the default of `eventual`. Availability (Section 6) and Target facets (Section 9) appear at the end of the example.

3.1 HYDROLOGIC Semantics

HYDROLOGIC's program semantics begin with its event loop, which is based on the transducer model in Bloom [11]. HYDROLOGIC's event loop considers the current *snapshot* of program state, which includes any new inbound messages to be handled. Each iteration ("tick") of the loop uses the developer's program specification to compute new results from the snapshot, and atomically updates state at the end of the tick. All computation within the tick is done to fixpoint. The snapshot and fixpoint semantics together ensure that the results of a tick are independent of the order in which statements appear in the program.

Simple COVID-19 Tracker App in Pythonic HydroLogic

```
1  class Person: (pid: int, country: string,
2          contacts: Set(&Person), covid: bool, vaccinated: bool,
3          key=pid, partition=country)
4  table people: Person
5  var vaccine_count: int

7  on add_person(pid: int):
8    people.merge(Person(pid)) # monotonic mutation
9    return OK

11 on add_contact(p: Person, p1: Person):
12   p.contacts.merge(p1) # monotonic mutation
13   p1.contacts.merge(p) # monotonic mutation
14   return OK

16 query transitive(p: Person, p1: Person): # monotonic query
17     {(p, p1) for p in people for p1 in p.contacts}
18     {(p, p2) for (p, p1) in transitive for p2 in p1.contacts}

20 on trace(p: Person):
21   return (p2 for (p, p2) in transitive(p, _)

23 on diagnosed(pid: int):
24   people[pid].covid.merge(true) # monotonic mutation
25   send alert(p: Person) {p for p in trace(pid)}

27 from covid_xmission_model import covid_predict
28 on likelihood(pid: int):
29   return covid_predict(people[pid])

31 on vaccinate(pid: int, consistency={serializable;
32                vaccine_count >= 0; people.has_key(pid)}):
33   people[pid].vaccinated.merge(True) # monotonic mutation
34   vaccine_count := vaccine_count - 1 # NON-monotonic mutation
35   return OK

37 availability:
38     default: { domain = AZ, failures = 2 }
39     likelihood: { domain = AZ, failures = 1 }

41 target:
42     default: { latency = 100ms, cost = 0.01units }
43     likelihood: { processor = GPU, cost = 0.1units }
```

Figure 3: A simple COVID-19 tracking application in a Pythonic HydroLogic syntax. Each **on** handler has faceted specifications of consistency, availability and deployment either in the definition (as is done here with consistency specs) or defined in a separate block.

The notion of endpoints and events should be familiar to developers of microservices or actors. Unlike microservices, actors or Bloom, HydroLogic's application semantics provide a simple "single-node" model—a global view of state, and a single event loop providing a single sequence (clock) of iterations (ticks). This single-node metaphor is part of the facet's declarative nature—it ignores issues of data placement, replication, message passing, distributed time and consistency, deferring them to separable facets of the stack.

Basic statements in HydroLogic's program semantics facet come in a few forms:

— *Queries* derive information from the current snapshot. Queries are named and referenceable, like SQL views, and defined over various lattice types, including relational tables. Line 17 represents a simple query returning pairs of `Person`s, the second of whom is a contact in the first. As in Datalog, multiple queries can have the same name, implicitly defining a merge of results across them. Lines 17 and 18 are an example, defining the base case and inductive case, respectively, for graph transitive closure[1]. A query q may have the same name as a data variable q, in which case the contents of data variable q are implicitly included in the query result.

— *Mutations* are requests to modify data variables based on the current contents of the snapshot. Following the transducer model, mutations are deferred until the end of a clock "tick"—they become visible together, atomically, once the tick completes. Mutations take three forms. A lattice merge mutation as in lines 8,12,13, or 33 monotonically "merges in" the lattice value of its argument. The traditional bare assignment operator `:=`, as in line 34 represents an arbitrary, likely non-monotonic update. A query q with the same name as a data variable q implicitly replaces (mutates) q at end of tick; this mutation is monotonic iff the query is monotonic.

— *Handlers* begin with the keyword **on**, and model reactions to messages. Seen within the confines of a tick, though, a handler is simply syntactic sugar for Hydro statements mapped over a *mailbox* of messages corresponding to the handler's name. The body of a handler is a collection of HydroLogic statements, each quantified by the particular message being mapped. For example, the `add_person` handler on Line 7 is syntactic sugar for the HydroLogic statements:

```
people.merge(Person(a.pid) for a in add_person)
send add_person<response>(message_id: int, payload: Status):
  {(a.message_id, OK) for a in add_person}
```

The implicit mailbox `add_person<response>` is used to send results to the caller of an `add_purpose` API—e.g., to send the HTTP status response to a REST call.

— *UDFs* are black-box functions, and may keep internal state across invocations. An example UDF, `covid_predict`, can be seen in the `likelihood` handler of line 28. UDFs cannot access HydroLogic variables and should avoid any other external, globally-visible data storage. Because UDFs can be stateful and non-idempotent, each UDF is invoked once per input per tick (memoized by the runtime), in arbitrary order.

— *Send* is an asynchronous merge into a mailbox. As with mutations, sends are not visible during the current tick. Unlike mutations, sends might not appear atomically—each individual object sent from a given tick may be "delayed" an

[1]HydroLogic supports recursion and non-monotonic operations (with stratified negation) for both relations and lattices. These features are based on Bloom^L and the interested reader is referred to [28] for details.

unbounded number of ticks, appearing non-deterministically in the specified mailbox at any later tick. Sends capture the semantics of unbounded network delay. Line 25 provides an internal example, letting the compiler know that we expect alerts to be delivered asynchronously. As another example, we can rewrite the `likelihood` handler of line 28 to use a remote FaaS service. This requires sending a request to the service and handling a response:

```
on async_likelihood(pid:int, isolation=snapshot)
  send FaaS((covid_predict, handler.message_id, find_person(pid)))

on covid_predict<response>(al_message_id: int, result: bool):
  send async_likelihood<response>((handler.message_id,
                                   al_message_id, result))
```

HydroLogic statements can be bundled into *blocks* of multiple statements, as in the bodies of the `add_contact` and `vaccinate` handlers. Blocks can be declared as object-like *modules* with methods to scope naming and allow reuse. Blocks and modules are purely syntactic sugar and we do not describe them further here.

4 LIFTING TO HYDROLOGIC

We aim for HydroLogic to be an evolutionary, general-purpose IR that can be targeted from a range of legacy design patterns and languages, while pointing the way toward coding styles that take advantage of more recent research.

Our goal in the near term is not to convert any arbitrary piece of code into an elegant, easily-optimized HydroLogic program. In particular, we do not focus on lifting existing "hand-crafted" distributed programs to HydroLogic. We have a fair bit of experience (and humility!) about such a general goal. Instead we focus on two scenarios for lifting:

Lifting single-threaded applications to the cloud: Many applications consist largely of single-threaded logic, but would benefit from scaling—and autoscaling—in the cloud. In our earlier work, we have had success using verified lifting to convert sequential imperative code of this sort into declarative frameworks like SQL [26], Spark [4, 5] and Halide [6]. One advantage of sequential programs—as opposed to hand-coded multi-threaded or distributed "assembly code"—is that we do not have to reverse-engineer consistency semantics from ad hoc patterns of messaging or concurrency control in shared memory. Some interesting corpora of applications are already written in opinionated frameworks that assist our goals. For example, applications that are built on top of object-relational mapping (ORM) libraries such as Rails [75] and Django [36] are essentially built on top of data definition languages (e.g., ActiveRecord [1]), which makes it easy to lift the data model, and sometimes explicit transactional semantics as well. ORM-based applications also often serve as backends for multiple clients and need to scale over time—Twitter is a notorious example of a Rails app that had to be rewritten for scalability and availability.

Evolving a breadth of distributed programming frameworks: There are existing distributed programming frameworks that are fairly popular, and our near-term goal is to embrace these programming styles. Simple examples include FaaS interfaces and big-data style functional dataflow like Spark. Other popular examples for asynchronous distributed systems include actor libraries (e.g., Erlang [37], Akka [7], Orleans [22]), libraries for distributed promises/futures (e.g., Ray [69] and Dask [32] for Python), and collective communication libraries like that of MPI [70]. Programs written with these libraries adhere to fairly stylized uses of distributed state and computation, which we believe we can lift relatively cleanly to HydroLogic. In Appendix A we share our initial thoughts and examples in this direction.

Our goals for lifting also offer validation baselines for the rest of our research. If we can lift code from popular frameworks, we can auto-generate a corpus of test cases. Hydro should aim to compete with the native runtimes for these test cases. In addition, lifting to HydroLogic will hopefully illustrate the additional flexibility Hydro offers via faceted re-specification of consistency, availability and performance goals. And finally, success here across different styles of frameworks will demonstrate the viability of our stack as a common cloud runtime for multiple styles of distributed programming, old and new.

5 HYDROLOGIC'S DATA MODELING

HydroLogic data models consist of four components: 1) a class hierarchy that describes how persistent data is structured, 2) relational constraints, such as functional dependencies, 3) persistent collection abstractions like relations, ordered lists, sets, and associative arrays, and 4) declarations for data placement across nodes in distributed deployments.

For instance, Lines 1-5 in Figure 3 show an example of persistent data specification for our Covid application. The data is structured as `Person` objects, each storing an integer `pid` that serves as a unique id (key), along with a set of references to other `Persons` that they have been in contact with. Line 3 illustrates an optional `partition` value to suggest how `Person` objects should be partitioned across multiple nodes. (HydroLogic uses the class's unique id to partition by default). Line 4 then prescribes that the `Persons` are to be collectively stored in a `table` keyed on each person's `pid` that is publicly accessible by all functions in the program.

Partitioning allows developers to hint at ways to scatter data; a similar syntax for locality hints is available. These hints are not required, however: HydroLogic programmers can define their data model without needing to know how their data will be stored in the cloud. The goal of Hydro is to take such user-provided specifications and generate a concrete implementation afterwards.

5.1 Design Space

As part of compilation, we need to choose an implementation of the data model facet. For example in our Covid tracker we might store `Person` objects in memory using an associative array indexed on each person's `pid`, with each person's `contacts` field stored as a list with only the `pids` of the `Person` objects. Obviously this particular implementation choice has tradeoffs with more normalized choices, depending on workload.

In general, a concrete data structure implementation consists of two components: choosing the container(s) to store persistent data (e.g., a B+-tree indexed on a field declared in one of the persistent classes), and determining the access path(s) given the choices for containers (e.g., an index or full container scan) when looking up a specific object.

We envision that there will be multiple algorithms to generate concrete implementations. These can range from a rule-driven approach that directly matches on specific forms of queries and determines the corresponding implementation (e.g., for programs with many lookup queries based on `id`, use an associative array to store `Person` objects), to a synthesis-driven approach that enumerates different implementations based on a grammar of basic data structure implementations [48] and a cost model. Access paths can then be determined based on how the containers are selected.

5.2 Promise and Challenges

We have designed a data structure synthesizer called Chestnut in our earlier work [76, 88], focusing on database-backed web applications that are built using ORM libraries. Similar to HydroLogic, Chestnut takes in a user-provided data model and workload specification, and synthesizes data structures to store persistent data once it is loaded into memory. Synthesis is done using an enumeration-based approach based on a set of provided primitives. For example, if a persistently stored class contains N attributes, Chestnut would consider storing all objects in an ordered list, or in an associative array keyed on any of the N unique attributes, or split the N attributes into a subset that is stored in a list, and the rest stored in a B+-tree index. The corresponding access path is generated for each implementation. Choosing among the different options is guided by a cost model that estimates the cost of each query that can potentially be issued by the application. Evaluations using open-source web apps showed that Chestnut can improve query execution by up to 42×.

Searching for the optimal data representation is reminiscent of the physical design problem in data management research, and there has been a long line of work on that front [3] that we can leverage. There has also been work done on data structure synthesis in the programming systems research community [55, 57] that focuses on the single-node setting, with the goal to organize program state as relations and persistently store them as such.

Synthesizing data structures based on HydroLogic specifications will raise new challenges. First, we will need to design a data structure programming interface that is expressive enough for the program specifications that users will write. Next, we will need a set of data structure "building blocks" that the synthesizer can utilize to implement program specifications. Such building blocks must be composable such that new structures can be designed, yet not too low-level that makes it difficult to verify if the synthesized implementation satisfies the provided specifications.

In addition, synthesizing distributed data structures will require new innovations in devising cost models for data transfer and storage costs, and reasoning about data placement and lookup mechanisms. New synthesis and verification algorithms will need to be devised in order to handle both aspects efficiently. Finally, workload changes (both client request rates and cloud service pricing) motivate incremental synthesis, where initial data structures are generated when the program is deployed, and gradually refined or converted to other implementations based on runtime properties.

6 THE AVAILABILITY FACET

The availability facet starts with a simple programmer contract: ensure that each application endpoint remains available in the face of f independent failures. In this discussion we assume that failures are non-Byzantine. The definition of independence here is tied to a user-selected notion of *failure domains*: two failures are considered independent if they are in different failure domains. Typical choices for failure domains include virtual machines, racks, data centers, or availability zones (AZs). In line 38 of Figure 3 we specify that our handlers should tolerate faults across 2 AZs. In line 39 we override that spec for the case of the `likelihood` handler, an ML routine that requires expensive GPU reservations, for which we trade off availability to save cost.

6.1 Design Space

The natural methodology for availability is to replicate service endpoints—execution and state—across failure domains. This goes back to the idea of process pairs in the Tandem computing systems, followed by the Borg and Lamport notions of state machine replication, and many distributed systems built thereafter. For example, when compiling the handler for the `add_contact` endpoint in line 11 of Figure 3, we can interpose HydroLogic implementing a load-balancing client proxy module that tracks replicas of the endpoint, forwards requests on to $f + 1$ of them, and makes sure that a response gets to the client.

Another standard approach is for backend logic to replicate its internal state—often by generating logs or lineage of state mutation events for subsequent replay. We could do this naively in our example by matching each mutation statement with a log statement.

6.2 Promise and Challenges

Whether using replication or replay, availability is fundamentally achieved by *redundancy* of state and computation. The design of that redundancy is typically complicated by two issues. The first is cost. In the absence of failure, redundancy logic can increase latency. Worse, running an identical replica of a massive service could be massively expensive. As a result, some replication schemes reduce the cost of replicas by having them perform logic that is different from—but semantically equivalent to—state change at the main service. A standard example is to do logical logging at the storage level, without redundantly performing application behavior. In general, it would of course be challenging to synthesize sophisticated database logging and recovery protocols from scratch. But simpler uses of activity logs for state replication are an increasingly common design pattern for distributed architectures [20, 39, 50], and use of these basic log-shipping patterns and services could offer a point in the optimization space of latency, throughput and resource consumption that differs from application-level redundancy.

The second complication that arises immediately from availability is the issue of consistency across redundant state and computation, which we address next with its own facet.

7 THE CONSISTENCY FACET

The majority of distributed systems work has relegated consistency issues to the storage or memory layer. But the past decade has seen a variety of clever applications (shopping carts [33], collaborative editing systems [83], gradiant descent [72], etc.) that have demonstrated massive performance and availability benefits by customizing consistency at the application layer. In Hydro we aim to take full programs—including compositions of multiple independent modules—and automatically generate similarly clever, "just right" code to meet *application-specific* consistency specifications.

The idea of raising transactional consistency from storage to the programming language level is familiar from object databases [16] and distributed object systems. Liskov's Argus language [56] is a canonical example, with each distributed method invoked as an isolated (nested) transaction, strictly enforced via locking and two-phase commit. This provides strong correctness properties—unnecessarily strong, since not every method call in a distributed application requires strong consistency or perfect isolation. From our perspective today, Argus and its peers passed up the biggest question they raised: if all the application code is available, how *little* enforcement can the compiler use to provide those semantics? And what if those semantics are weaker than serializability?

As seen in the example of Figure 3, HydroLogic allows consistency to be specified at the level of the client API handlers. Like all our facets, consistency can be specified inline with the handler definition (as in Figure 3), or in a separate `consistency` block. In practice, applications are built from code written by different parties for potentially different purposes. As a result the original consistency specs provided for different handlers may be heterogeneous within a single application. What matters in the end is to respect the (possibly heterogeneous) consistency that clients of the application can observe from its public interfaces.

In Figure 3, the `add_person` handler uses default eventual consistency. This ensures that if the two `people` in the arguments are not physically co-located, then each person (and each replica) can be updated without waiting for any others.

As a different example, the `vaccinate` handler specifies `serializability` and a non-negative `vaccine_count` constraint. We might be concerned that serializability for this handler will require strong consistency from other handlers. Close analysis shows this is not the case: `vaccinate` is the only handler that references `vaccine_count`, and all references to `people` are monotonic and hence reorderable—including the mutation in `vaccinate`. Hence if `vaccinate` completes for some `pid` in any history, there is an equivalent serial history in which `vaccinate(pid)` runs successfully with the same initial value of `vaccine_count` and the same resulting value of both `vaccine_count` and `people`.

7.1 Design Space

In HydroLogic we enable two different types of consistency specifications: traditional history-based guarantees, and application-centric *invariants*. History-based guarantees are prevalent today, with widely agreed-upon semantics. For example, serializability, linearizability, sequential consistency, causal consistency, and others specifically constrain the ordering of conflicting operations and in turn define "anomalies" that applications can observe. The second type of consistency annotation we allow is application-centric, and makes use of Hydrologic's declarative formulation. Past work has demonstrated that invariants are a powerful way for developers to precisely specify what guarantees are necessary at application level [18, 21, 30, 45, 60, 74, 79, 84]. These include motonicity invariants that guarantee convergent outcomes, or isolation invariants for predicates on visible states—e.g., positive bank accounts or referential integrity.

7.2 Promise and Challenges

Many challenges fall out from an agenda of compiling arbitrary distributed code to efficiently enforce consistency invariants. Based on work to date, we believe the field is ripe for innovation. Here we highlight some key challenges and our reasons for optimism.

Metaconsistency Analysis: Servicing a single public API call may require crossing multiple internal endpoints with different consistency specifications. This entails two challenges: identifying the possible composition paths, and ensuring *metaconsistency*: the consistency of heterogeneous consistency specs along each path. The first problem amounts to dataflow analysis across HydroLogic handlers; this is easy to do conservatively in a static analysis of a HydroLogic program, though we may desire more nuanced conditional solutions enforced at runtime. The question of metaconsistency is related to our prior work on mixed consistency of black-box services [31, 67, 68]. In the Hydro context we may use third-party services, but we also expect to have plenty of white-box HydroLogic code, where we have the flexibility to change the consistency specs across modules to make them consistent with the consistency of endpoint specifications. Our recent work on client-centric consistency offers a unified framework for reasoning about both transactional isolation *and* distributed consistency guarantees [29].

Consistency Mechanisms: Given a consistency requirement, we need to synthesize code to enforce it. There are three broad approaches to choose from. The first is to recognize when no enforcement is required for a particular code block—examples include the monotonicity and invariant confluence analyses mentioned above. Another is for the compiler to wrap or "encapsulate" state with lattice metadata that allows for local (coordination-free) consistency enforcement at each endpoint—this is the approach in our work on the Cloudburst FaaS [81] and Hydrocache [86]. The third approach is the traditional "heavyweight" use of coordination protocols, including barriers, transaction protocols, consensus-based logs for state-machine replication and so on. The space of enforcement mechanisms is wide, but there are well-known building blocks in the literature that we can use to start on our software synthesis agenda here.

Consistency Placement: Understanding consistency specs and mechanisms is not enough—we can also reason about where to invoke the mechanism in the program, and how the spec is kept invariant downstream. This flexibility arises when we consider consistency at an application level rather than as a storage guarantee. As a canonical example, the original work on Dynamo's shopping carts was coordination-free *except* for "sealing" the final cart contents for checkout [11, 33, 43]. Conway [28] shifted the sealing to the end-user's browser code where it is decided unilaterally (for "free") in an unreplicated stage of the application. When shopping ends, the browser ships a compressed *manifest* summarizing the final content of the cart. Maintaining the final cart state at the replicas then becomes coordination-free as well: each replica can eagerly move to checkout once its contents match the manifest. Alvaro systematized this sealing idea in Blazes [12]; more work is needed to address the variety of coordination guarantees we wish to enforce and maintain.

Clearly these issues are correlated, so a compiler will have to explore the space defined by their combinations.

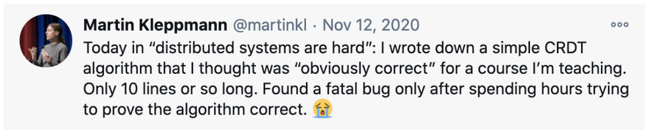
8 THE HYDROFLOW IR

Like many declarative languages, to execute HydroLogic we translate it down to a lower-level algebra of operators that can be executed in a flow style on a single node, or partitioned and pipelined across multiple nodes (Section 9). Most of these operators are familiar from relational algebra and functional libraries like Spark and Pandas. Here we focus on the unique aspects of the HydroLogic algebra.

8.1 Design Space

The Hydroflow algebra has to handle all the constructs of HydroLogic's event loop. One of the key goals of the Hydroflow algebra design is a *unification of dataflow, lattices and reactive programming.* Typical runtimes implement a dataflow model of operators over streaming collections of individual items. This assumes that collection types and their operators are the primary types in any program. We want to accommodate lattices beyond collection types. For example, a `COUNT` query takes a set lattice as input and produces an integer lattice as output; we need the output of that query to "pipeline" in the same fashion as a set. In addition, to capture state mutation we want to adapt reactive programming models (e.g., React.js and Rx). that provide ordered streams propagating changes to individual values over time.

In deployment, a Hydro program involves Hydroflow algebra fragments running at multiple nodes in a network, communicating via messages. Inbound messages appear at Hydroflow ingress operators, and outbound messages are produced by egress operators. These operators are agnostic to networking details like addressing and queueing, which are parameterized by the target facet. However, as a working model we can consider that a network egress point in Hydroflow can be parameterized to do explicit point-to-point networking, or a content-hash-based style of addressing. As a result, local Hydroflow algebra programs can participate as fragments of a wide range of deployment models, including parallel intra-operator partitioning (a la Exchange or MapReduce) as well as static dataflows across algebraic operators, or dynamic invocations of on-demand operators.



Martin Kleppmann @martinkl · Nov 12, 2020

Today in "distributed systems are hard": I wrote down a simple CRDT algorithm that I thought was "obviously correct" for a course I'm teaching. Only 10 lines or so long. Found a fatal bug only after spending hours trying to prove the algorithm correct. 😭

Figure 4: On the trickiness of manual checks for monotonicity. The full thread includes pseudocode and fixes [52].

8.2 Promise and Challenges

A program in HydroLogic can be lowered (compiled) to a set of single-node Hydroflow algebra expressions in a straightforward fashion, much as one can compile SQL to relational algebra. Our concern at this stage of lowering is scoped to single-node, in-memory performance; issues of distributed computing are deferred to Section 9. The design space here is similar to that of traditional query optimization, and classical methods such as Cascades are a plausible approach [40]. We are also considering more recent results in program synthesis here, since they have shown promise in traditional query optimization [24, 26].

The design of the Hydroflow algebra is a work in progress, and achieving a semantics that unifies all of its aspects is non-trivial. In addition, two other challenges arise naturally.

Monotonicity typechecking: Current models for monotonic programming like CRDT libraries expect programmers to guarantee the monotonicity of their code manually. This is notoriously tricky—see Figure 4. Bloom^L attempted to simplify this problem by replacing monolithic CRDTs with monotone compositions of simpler lattices, but correctness was still assumed for the basic lattices and composition functions. We wish to go further, providing an explicit `monotone` type modifier, and a compiler that can typecheck monotonicity. Guarantees of monotonicity can be exploited to ensure guarantees from the consistency facet (Section 7) as part of the Optimization facet (Section 9).

Representation of flows beyond collections: Algebras defined for a collection type `C<T>` (e.g., relational algebra on `set<tuple>`) are often implemented in a dataflow of operators over the underlying element type `T`, or over incremental batches of elements. This *differential* approach is well-suited for operators on `C<T>` that have stateless implementations over `T`—e.g., map, select and project. Other operators require stateful implementations that use ad-hoc internal memory management to rebuild collections of type `C<T>` across invocations over type `T`. This makes it difficult for a compiler to check properties like determinism or monotonicity. Moreover, in Hydroflow we want to expand flow computation beyond collection types to lattices and reactive scalar values. Hence we need to support operators that view inputs differentially *or* all-at-once, providing clear semantics for both cases, and allowing properties like monotonicity to be statically checked by a compiler.

Copy efficiency: In many modern applications and systems, the majority of compute time is spent in copying and formatting data. Developers who have built high-performance query engines know that it is relatively easy to build a simple dataflow prototype, and quite hard to build one that makes efficient use of memory and minimizes the cost of data copying and redundant work. Taking a cue from recent systems like Timely Dataflow [61], we use the ownership properties of the Rust language to help us carefully control how data is managed in memory along our flows.

9 THE TARGET FACET

After specifying various semantic aspects of the application, the final facet describes the targets for optimization that the cloud runtime should achieve, as described in Section 1.1. Such targets can include a cost budget to spend on running the application on the cloud, maximum number of machines to utilize, specific capabilities of the hosted machines (e.g., GPU on board), latency requirements for any of the handlers, etc. We imagine that the user will provide a subset of these configuration parameters and leave the rest to be determined by Hydrolysis.

For example, lines 41-43 in Figure 3 show the targets for our COVID application. Line 42 specifies the default latency/cost goals for handlers; line 43 specializes this for machine-learning-based `likelihood` handler, dictating the use of GPU-class machines with a higher budget per call.

Compared to the current practice of deployment configurations spread across platform-specific scripts [13, 66], program annotations [73], and service level agreements, HydroLogic allows developers to consolidate deployment-related targets in an isolated program facet. This allows developers to easily see and change the cost/performance profiles of their code, and enables HydroLogic applications to be deployed across different cloud platforms with different implementations.

9.1 Design Space

Given a HydroLogic specification, the Hydrolysis compiler will attempt to find an implementation that satisfies the provided constraints subject to the desired overall objectives. As discussed in Section 2, Hydro will first generate an initial implementation of the application based on the previously described facets. The initial implementation would have various aspects of the application determined: algorithms for data-related operations, replication and consistency protocols, etc. What remains are the runtime deployment aspects such as mapping of functions and data to available machines.

For instance, given the code in Figure 3, Hydro can formulate the runtime mapping problem as an integer programming problem, based on our prior work [25, 87]. Such a mapping problem can be formulated as a dynamic program partitioning problem. Suppose at any given time we have M different types of machine configurations to choose from, and n_i represents the number of instances we will use for machine of type i. We then have the following constraints:

- $latency(\texttt{add\_person}, n_i) \leq 100ms$. The latency incurred by hosting add_person on n_i instances of type i machines must be less than the specified value. We have one constraint for each pair of handler and machine type, and Figure 3 shows a shortcut using the default construct while overriding it for the likelihood handler.
- $cost(\texttt{add\_person}, n_i) \leq 0.01$. The cost of running add_person on n_i instances of type i machines must be less than the specified value. The value can either be specified by the end user or provided by the hosting platform.
- $\sum_i n_i > 0$. Allocate some machines to fulfill the workload.

The overall objective depends on the user specification, for instance minimizing the total number of machines used ($\sum_i n_i$), or maximizing overall throughput of each handler f_j while executed on n_i machines ($\sum_{i,j} tput(f_j, n_i)$). More sophisticated objectives are possible, for instance incurring up to a fixed cost over a time period [14].

As formulated above, our integer programming problem relies on having models to estimate latency, throughput, and cost of running each function given machine type and number of instances. Solving the problem gives us the values of each n_i, i.e., the number of instances to allocate for each machine configuration. Given that program objectives or resource costs can vary as the application executes in the cloud, we might need to periodically reformulate the problem based on the data available. Predicting or detecting when a reformulation is needed will be interesting future work.

Note that the above integer program might not have a solution, e.g., if the initial implementations were too costly to meet the given targets. If this arises, Hydro can ask previous components to choose other implementations and reiterate the mapping procedure. This iterative process is simplified by decomposing the application into facets, allowing Hydro to revert to a previous search state during compilation.

9.2 Promise and Challenges

Our problem formulation above is inspired by prior work in multi-query optimization [77], multi-objective optimization [82] and scheduling tasks for serverless platforms [49]. Our faceted setting, however, also raises new challenges.

Cost modeling: Our integer programming problem formulation relies on having accurate cost models for different aspects of program execution on the cloud (e.g., latency prediction). While cost prediction has been a classic research topic in data management, much of the prior work has focused on single and distributed relational databases. The cloud presents new challenges as functions can move across heterogeneous machines, and aspects such as machine prices and network latencies can vary at any time.

Solution enumeration: As mentioned earlier, our faceted approach allows Hydro to easily backtrack during compilation, should an initial strategy turn out to be infeasible given the configuration constraints. Implementing backtracking will rely on an efficient way to enumerate different implementations based on the previously described facets, and being able to do so efficiently in real time. This depends on the algorithms used to generate the initial implementations, for instance by considering types of query plans that were previously bypassed during code generation, or asking solvers to generate another satisfiable solution if formal methods-based algorithms are used. We will also need feedback mechanisms to interact with the user, should the provided specifications prove too stringent.

Adaptive optimization: One of the reasons for deploying applications on the cloud is to leverage the cloud's elasticity. As a consequence, the implementation generated by Hydro will likely need to change over time. While Hydro's architecture is designed to tackle that aspect by not having hard-wired rules for code generation, we will also devise new runtime monitoring and adaptive code generation techniques, in the spirit of prior work [17, 25, 34].

10 CONCLUSION

We are optimistic that the time is ripe for innovation and adoption of new technologies for end-user distributed programming. This is based not only on our assessment of research progress and potential, but also the emerging competition among cloud vendors to bring third-party developers to their platforms. We are currently implementing the Hydroflow runtime, and exploring different algorithms to lift legacy design patterns to HydroLogic. Our next goals are to design the compilation strategies from HydroLogic to Hydroflow programs, and to explore the compilation of application-specific availability and consistency protocols. We are also contemplating related research agendas in security and developer experience including debugging and monitoring. There are many research challenges ahead, but we believe they can be addressed incrementally and in parallel, and quickly come together in practical forms.

11 ACKNOWLEDGMENTS

This work is supported in part by the National Science Foundation through grants CNS-1730628, IIS-1546083, IIS-1955488, IIS-2027575, CCF-1723352, and DOE award DE-SC0016260; the Intel-NSF CAPA center, and gifts from Adobe, Amazon, Ant Group, Ericsson, Facebook, Futurewei, Google, Intel, Microsoft, NVIDIA, Scotiabank, Splunk and VMware.

REFERENCES

[1] *Active Record.* https://github.com/rails/rails/tree/master/activerecord.

[2] G. Agha. Concurrent object-oriented programming. *Communications of the ACM*, 33(9):125–141, 1990.

[3] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in sql databases. In *VLDB*, 2000.

[4] M. Ahmad and A. Cheung. Leveraging parallel data processing frameworks with verified lifting. In *Proceedings Fifth Workshop on Synthesis, SYNT@CAV 2016*, pages 67–83, 2016.

[5] M. Ahmad et al. Automatically leveraging mapreduce frameworks for data-intensive applications. In *SIGMOD*, pages 1205–1220, 2018.

[6] M. Ahmad et al. Automatically translating image processing libraries to halide. *ACM Trans. Graph.*, 38(6):204:1–204:13, 2019.

[7] *Akka homepage*, Aug. 2020.
https//akka.io, retrieved 8/28/2020.

[8] I. E. Akkus, R. Chen, I. Rimac, M. Stein, K. Satzke, A. Beck, P. Aditya, and V. Hilt. SAND: Towards high-performance serverless computing. In *USENIX ATC*, pages 923–935, 2018.

[9] P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J. M. Hellerstein, and R. Sears. Boom analytics: exploring data-centric, declarative programming for the cloud. In *Proceedings of the 5th European conference on Computer systems*, pages 223–236, 2010.

[10] P. Alvaro, T. Condie, N. Conway, J. M. Hellerstein, and R. Sears. I do declare: consensus in a logic language. *ACM SIGOPS Operating Systems Review*, 43(4):25–30, 2010.

[11] P. Alvaro et al. Consistency Analysis in Bloom: a CALM and Collected Approach. In *CIDR*, pages 249–260, 2011.

[12] P. Alvaro et al. Blazes: Coordination analysis for distributed programs. In *ICDE*, pages 52–63. IEEE, 2014.

[13] Amazon. Amazon API gateway. https://aws.amazon.com/api-gateway/.

[14] Amazon. Aws budgets update – track cloud costs and usage. https://aws.amazon.com/blogs/aws/aws-budgets-update-track-cloud-costs-and-usage.

[15] T. J. Ameloot, F. Neven, and J. Van den Bussche. Relational transducers for declarative networking. In *Principles of Database Systems (PODS)*, pages 283—-292, June 2011.

[16] M. Atkinson, D. Dewitt, D. Maier, F. Bancilhon, K. Dittrich, and S. Zdonik. The object-oriented database system manifesto. In *Deductive and object-oriented databases*, pages 223–240. Elsevier, 1990.

[17] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, pages 261–272. ACM, 2000.

[18] P. Bailis, A. Fekete, M. J. Franklin, A. Ghodsi, J. M. Hellerstein, and I. Stoica. Coordination avoidance in database systems. *Proc. VLDB Endow.*, 8(3):185–196, Nov. 2014.

[19] H. C. Baker and C. Hewitt. The incremental garbage collection of processes. *SIGPLAN Not.*, 12(8):55–59, Aug. 1977.

[20] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobber, M. Wei, and J. D. Davis. CORFU: A shared log design for flash clusters. In S. D. Gribble and D. Katabi, editors, *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*, pages 1–14. USENIX Association, 2012.

[21] V. Balegas, N. Preguiça, R. Rodrigues, S. Duarte, C. Ferreira, M. Najafzadeh, and M. Shapiro. Putting consistency back into eventual consistency. In *EuroSys*, pages 6:1–6:16, Bordeaux, France, Apr. 2015. Indigo.

[22] S. Bykov, A. Geller, G. Kliot, J. R. Larus, R. Pandya, and J. Thelin. Orleans: cloud computing for everyone. In *SoCC*, pages 1–14, 2011.

[23] C. Chedeau. React's architecture. In *OSCON*, July 2014.

[24] A. Cheung, O. Arden, S. Madden, A. Solar-Lezama, and A. C. Myers. StatusQuo: Making familiar abstractions perform using program analysis. In *CIDR*, 2013.

[25] A. Cheung, S. Madden, O. Arden, and A. C. Myers. Automatic partitioning of database applications. *PVLDB*, 5(11):1471–1482, 2012.

[26] A. Cheung, A. Solar-Lezama, and S. Madden. Optimizing database-backed applications with query synthesis. In *PLDI*, pages 3–14, 2013.

[27] T. Condie, D. Chu, J. M. Hellerstein, and P. Maniatis. Evita raced: metacompilation for declarative networks. *Proceedings of the VLDB Endowment*, 1(1):1153–1165, 2008.

[28] N. Conway et al. Logic and lattices for distributed programming. In *SoCC*, pages 1–14, 2012.

[29] N. Crooks. *A client-centric approach to transactional datastores*. PhD thesis, University of Texas, Austin, 2020.

[30] N. Crooks, Y. Pu, L. Alvisi, and A. Clement. Seeing is believing: A client-centric specification of database isolation. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, PODC '17, page 73–82, New York, NY, USA, 2017. Association for Computing Machinery.

[31] N. Crooks, Y. Pu, N. Estrada, T. Gupta, L. Alvisi, and A. Clement. Tardis: A branch-and-merge approach to weak consistency. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, page 1615–1628, New York, NY, USA, 2016. Association for Computing Machinery.

[32] Dask parallel computing library. https://dask.org/.

[33] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. *ACM SIGOPS operating systems review*, 41(6):205–220, 2007.

[34] A. Deshpande, Z. G. Ives, and V. Raman. Adaptive query processing. *Foundations and Trends in Databases*, 1(1):1–140, 2007.

[35] D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *Communications of the ACM*, 35(6):85–98, 1992.

[36] *Django.* https://www.djangoproject.com/.

[37] The Erlang programming language. https://www.erlang.org.

[38] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, 1 edition, 1994.

[39] J. Goldstein, A. S. Abdelhamid, M. Barnett, S. Burckhardt, B. Chandramouli, D. Gehring, N. Lebeck, C. Meiklejohn, U. F. Minhas, R. Newton, R. Peshawaria, T. Zaccai, and I. Zhang. A.M.B.R.O.S.I.A: providing performant virtual resiliency for distributed applications. *Proc. VLDB Endow.*, 13(5):588–601, 2020.

[40] G. Graefe. The cascades framework for query optimization. *IEEE Data Eng. Bull.*, 18(3):19–29, 1995.

[41] C. Granger. Against the current: What we learned from eve. In *Future of Coding LIVE Conference*, 2018. https://futureofcoding.org/notes/live/2018.

[42] S. Gulwani. Dimensions in program synthesis. In T. Kutsia, W. Schreiner, and M. Fernández, editors, *PPoPP*, pages 13–24, 2010.

[43] P. Helland and D. Campbell. Building on quicksand. In *CIDR*, 2009.

[44] J. M. Hellerstein. The declarative imperative: experiences and conjectures in distributed logic. *ACM SIGMOD Record*, 39(1):5–19, 2010.

[45] J. M. Hellerstein and P. Alvaro. Keeping calm: when distributed consistency is easy. *Communications of the ACM*, 63(9):72–81, 2020.

[46] J. M. Hellerstein, J. Faleiro, J. E. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu. Serverless computing: One step forward, two steps back. In *CIDR*, 2019.

[47] C. Hewitt. Viewing control structures as patterns of passing messages. *Artificial intelligence*, 8(3):323–364, 1977.

[48] S. Idreos, K. Zoumpatianos, M. Athanassoulis, N. Dayan, B. Hentschel, M. S. Kester, D. Guo, L. M. Maas, W. Qin, A. Wasay, and Y. Sun. The periodic table of data structures. *IEEE Data Eng. Bull.*, 41(3):64–75, 2018.

[49] K. Kaffes, N. J. Yadwadkar, and C. Kozyrakis. Centralized core-granular scheduling for serverless functions. In *Proceedings of the ACM Symposium on Cloud Computing, SoCC 2019, Santa Cruz, CA, USA, November 20-23, 2019*, pages 158–164. ACM, 2019.

[50] Kafka. Apache kafka. https://kafka.apache.org.

[51] V. Kalavri, J. Liagouris, M. Hoffmann, D. Dimitrova, M. Forshaw, and T. Roscoe. Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 783–798, 2018.

[52] M. Kleppmann. Twitter thread, Nov. 2020. https://twitter.com/martinkl/status/1327020435419041792.

[53] L. Kuper and R. R. Newton. Lvars: lattice-based data structures for deterministic parallelism. In *ACM SIGPLAN*, pages 71–84, 2013.

[54] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[55] W. Lee, M. Papadakis, E. Slaughter, and A. Aiken. A constraint-based approach to automatic data partitioning for distributed memory execution. In M. Taufer, P. Balaji, and A. J. Peña, editors, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2019, Denver, Colorado, USA, November 17-19, 2019*, pages 45:1–45:24. ACM, 2019.

[56] B. Liskov. Distributed programming in argus. *Commun. ACM*, 31(3):300–312, Mar. 1988.

[57] C. Loncaric, E. Torlak, and M. D. Ernst. Fast synthesis of fast collections. In C. Krintz and E. Berger, editors, *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 355–368. ACM, 2016.

[58] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative networking. *Communications of the ACM*, 52(11):87–95, 2009.

[59] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing declarative overlays. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 75–90, 2005.

[60] T. Magrino, J. Liu, N. Foster, J. Gehrke, and A. C. Myers. Efficient, consistent distributed computation with predictive treaties. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[61] F. e. a. McSherry. A modular implementation of timely dataflow in rust, 2017. https://github.com/TimelyDataflow/timely-dataflow, retrieved 12/23/2020.

[62] E. Meijer. Reactive extensions (rx) curing your asynchronous programming blues. In *ACM SIGPLAN Commercial Users of Functional Programming*, pages 1–1. ACM, 2010.

[63] C. Meiklejohn and P. Van Roy. Lasp: A language for distributed, coordination-free programming. In *Principles and Practice of Declarative Programming*, pages 184–195, 2015.

[64] Message Passing Interface Forum. Mpi-2: Extensions to the message-passing interface, 1997. https://www.mcs.anl.gov/research/projects/mpi/mpi-standard/mpi-report-2.0/mpi2-report.htm, retrieved 12/16/2020.

[65] Metadata Partners, LLC. Datomic: Technical overview, 2012. https://web.archive.org/web/20120324023546/http://datomic.com/docs/datomic-whitepaper.pdf. Accessed: Dec. 13 2020.

[66] Microsoft. Azure management API. https://azure.microsoft.com/en-us/services/api-management.

[67] M. Milano and A. C. Myers. Mixt: A language for mixing consistency in geodistributed transactions. *SIGPLAN Notices*, 53(4):226–241, 2018.

[68] M. Milano, R. Recto, T. Magrino, and A. C. Myers. A tour of gallifrey, a language for geodistributed programming. In *SNAPL*, 2019.

[69] P. Moritz et al. Ray: A distributed framework for emerging AI applications. In *OSDI*, pages 561–577, 2018.

[70] MPI Collective Functions. https://docs.microsoft.com/en-us/message-passing-interface/mpi-collective-functions.

[71] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices*, 48(6):519–530, 2013.

[72] B. Recht, C. Re, S. Wright, and F. Niu. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. *Advances in neural information processing systems*, 24:693–701, 2011.

[73] D. Rogora, S. Smolka, A. Carzaniga, A. Diwan, and R. Soulé. Performance annotations for cloud computing. In E. de Lara and S. Sundararaman, editors, *9th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud*. USENIX Association, 2017.

[74] S. Roy, L. Kot, G. Bender, B. Ding, H. Hojjat, C. Koch, N. Foster, and J. Gehrke. The homeostasis protocol: Avoiding transaction coordination through program analysis. In *SIGMOD*, SIGMOD '15, page 1311–1326, New York, NY, USA, 2015. Association for Computing Machinery.

[75] *Ruby on Rails*. http://rubyonrails.org/.

[76] M. Samuel, C. Yan, and A. Cheung. Demonstration of chestnut: An in-memory data layout designer for database applications. In D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, and H. Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 2813–2816. ACM, 2020.

[77] T. K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, 1988.

[78] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*, pages 386–400. Springer, 2011.

[79] K. C. Sivaramakrishnan, G. Kaki, and S. Jagannathan. Declarative programming over eventually consistent data stores. *ACM SIGPLAN Notices*, 50(6):413–424, 2015.

[80] A. Sivaraman et al. Packet transactions: High-level programming for line-rate switches. In *SIGCOMM*, pages 15–28, 2016.

[81] V. Sreekanti et al. Cloudburst: Stateful functions-as-a-service. *PVLDB*, 13(11):2438–2452, 2020.

[82] I. Trummer and C. Koch. Multi-objective parametric query optimization. *SIGMOD Rec.*, 45(1):24–31, 2016.

[83] S. Weiss, P. Urso, and P. Molli. Logoot: A scalable optimistic replication algorithm for collaborative editing on p2p networks. In *2009 29th IEEE International Conference on Distributed Computing Systems*, pages 404–412. IEEE, 2009.

[84] M. Whittaker and J. M. Hellerstein. Interactive checks for coordination avoidance. *Proc. VLDB Endow.*, 12(1):14–27, Sept. 2018.

[85] C. Wu, J. Faleiro, Y. Lin, and J. Hellerstein. Anna: A kvs for any scale. *IEEE TKDE*, 2019.

[86] C. Wu, V. Sreekanti, and J. M. Hellerstein. Transactional causal consistency for serverless computing. In *SIGMOD*, pages 83–97, 2020.

[87] C. Yan and A. Cheung. Leveraging lock contention to improve OLTP application performance. In *PVLDB*, pages 444–455, 2016.

[88] C. Yan and A. Cheung. Generating application-specific data layouts for in-memory databases. *Proc. VLDB Endow.*, 12(11):1513–1525, 2019.

A LIFTING LEGACY DESIGN PATTERNS

We aim for HydroLogic to be a general-purpose IR that can be targeted from a range of input languages. In this section, we provide initial evidence that HydroLogic can be a natural target for code written in a variety of accepted distributed design patterns: actors, futures, and MPI. We provide working implementations of all code from the paper at https://github.com/hydro-project/cidr2021.

A.1 Actors

The Actor model has a long history [47]. In a nutshell, an actor is an object with three basic primitives [2]: (a) exchange messages with other actors, (b) update local state, and (c) spawn additional actors. Like other object abstractions, actors have private, encapsulated state. Actors are often implemented in a very lightweight fashion running in a single OS process; actor libraries like Erlang can run hundreds of thousands of actors per machine. At the same time, the asynchronous semantics of actors makes it simple to distribute them across machines.

Actors are like objects: they encapsulate state and handlers. HydroLogic does not bind handlers to objects, but we can enforce that when lifting by generating a HydroLogic program in which we have an `Actor` class keyed by `actor_id`, and each handler's first argument identifies an `actor_id` that associates the inbound message with a particular `Actor` instance. The HydroLogic to emulate spawning an actor simply creates a new `Actor` instance with a unique ID and runs any initialization code to associate initial state with the actor. In keeping with other actor implementations, each actor is very lightweight. HydroLogic allows us to optionally specify availability, consistency and deployment for our actors' handlers. Hydrolysis can choose to how to partition/replicate actors across machines.

Actor frameworks provide event loops, and at first blush it is straightforward to map an actor method into HydroLogic. Consider an actor method `do_foo(msg)` with an RPC-like behavior in Erlang style:

```
do_foo(msg) ->
  foo(msg);
```

This translates literally into a HydroLogic handler:

A Simple Actor Method in HydroLogic

```
on do_foo(actor_id, msg):
  return foo(msg)
```

RPC is a good match to the transducer model where code fragments are bracketed by send and receive. But actors are not transducers. In particular, they can issue blocking requests for messages at any point in their control flow. In the next listing, note that the actor first runs the function `m_pre(msg)`, then waits to receive a message in the `mybox` mailbox, after which it runs `m_post` on the message it finds:

```
m(msg) ->
  m_pre(msg)
  receive
    {mybox, newmsg} ->
      m_post(newmsg)
  end.
```

We can translate this into two separate handlers in HydroLogic, but we need to make sure that (a) the state of the computation (heap and stack) after `m_pre` runs is preserved, (b) `m_post` can run from that same state of computation, and (c) that the handler doesn't do anyting else while waiting for `newmsg`. *Coroutines* are a common language construct that provides convenient versions of (a) and (b), and are found in many host languages for actors (including C# for Orleans and Scala for Akka). The third property (c) can be enforced across HydroLogic ticks by a status variable in the actor[2]:

Mid-Method Message Handling in HydroLogic

```
on m(actor_id: int, msg):
  actors[actor_id].state := m_pre(msg)
  actors[actor_id].waiting := true
on m_receive_mybox(actor_id: int, newmsg):
  result = m_post(actors[actor_id].state, newmsg)
  actors.delete([actor_id])
  return result
```

Note that this HydroLogic has to use non-monotonic mutation to capture the (arguably undesirable!) notion of blocking that is implicit in a synchronous receive call.

A.2 Promises and Futures

Another common language pattern for distributed messaging is Promises and Futures; this has roots in the actor literature as well [19], but often appears independently. The basic idea is to spawn an asynchronous function call with a handle for the computation called a *Promise*, and a handle for the result called a *Future*. In the basic form, sequential code generates pairs of Promises and Futures, effectively launching the computation of each Promise in a separate thread of execution (perhaps on a remote machine), and continuing to process locally until the Future needs to be resolved. We take an example from the Ray framework in Python:

```
futures = [f.remote(i) for i in range(4)]
x = g()
print(ray.get(futures))
```

The function `f` is invoked as a promise for the numbers 0 through 3 via Ray's `f.remote` syntax; four futures are immediately stored in the array `futures`. The function `g()` then runs locally while the promises execute concurrently and remotely. After `g()` completes, the futures are resolved (in

[2]The attentive reader will note that we have elided a bit of bookkeeping here that buffers new inbound messages to `m` while the actor is waiting.

batch, in this case) by the `ray.get` syntax. In this simple example, futures are little more than a syntactic sugar for keeping track of asynchronous promise invocations. The translation to HydroLogic is straightforward. It could be sugared similarly to Ray if desired, but we show it in full below. Much like our mid-method receipt for actors, we implement waiting across HydroLogic ticks with a condition variable.

Promises/Futures in HydroLogic

```
import promises from PromisesEngine
var waiting
on start:
  send promises(handle: int, f, i: int):
    {(unique_id(), f, i) for i in range(4)})
  x := g()
  waiting := true
on futures(handle: int, result).len() >= 4:
  print([f.result for f in futures])
  futures.delete()
  waiting := false
```

Promise/Future libraries vary in their semantics, and it's relatively easy to generate HydroLogic code for each of these semantics. For example, note that promises and futures are data, so we can implement semantics where they can be sent or copied to different logical agents (like our actors above). Similarly, we can support a variety of "kickoff" semantics for promises. Our example above eagerly executes promises, but we could easily implement a lazy model, where pending promises are placed in a table until future requests come along.

A.3 MPI Collective Communications

MPI is a standard Message Passing Interface for scientific computing and supercomputers [64]. While this domain per se is not a primary target for HydroLogic, the "collective communication" patterns defined for MPI are a good suite of functionality that any distributed programming environment should support.

The MPI standard classifies these patterns into the following categories:

One-to-All, in which one agent contributes to the result, but all agents receive the result. The two basic patterns are `Bcast` (which takes a value and sends a copy of it to a set of agents) and `Scatter` (which takes an array and partitions it, sending each chunk to a separate agent).

All-to-One, in which all agents contribute to the result, but one agent receives the result. The basic patterns are `Gather` (which assembles data from all the agents into a dense array) and `Reduce` (which computes an aggregate function across data from all agents).

All-to-All, in which all agents contributes to *and* receive the result. This includes `Allgather` (similar to gather but all agents receive all the data and assemble the result array), `Allreduce` (similar to reduce except the result arrives at all agents) and `Alltoall` (all processes send and receive the same amount of data to each other).

These operations map very naturally into HydroLogic. Assume we start with a static table `agents` containing the set of relevant agentIDs.

MPI collective communication in HydroLogic

```
table agents(agent_id: int, key=agent_id)
query acount int:
  agents.count()
table gathered(request_id: int, ix: int, val,
               tombstone: bool,key=(request_id, ix))
query gcount(req_id: int, cnt: int):
  (req_id, gathered[req_id].count())

on mpi_bcast(msg_id: int, msg):
  send mpi_bcast_channel(agent_id: int, msg_id, int, msg):
    {(a.aid, msg_id, msg} for a in agents))

on mpi_scatter(req_id: int, arr):
  send mpi_scatter_channel(agent_id: int, req_id: int,
                           subarray):
    chunksz = arr.len()/acount
    if chunksz > 1:
      {(i, req_id, arr[range(i*chunksz, (i+1)*chunksz - 1)])
       for i in acount }
    else:
      {(i, req_id, arr[i]) for i in arr.len() }

on mpi_gather(req_id: int, ix: int, val):
  gathered.merge(req_id, ix, val)
  if (gcount[req_id] >= acount):
    result = gathered[req_id].array_agg(val, order=ix)
    gathered[req_id].tombstone.merge(true)
    return result

on mpi_reduce(req_id: int, ix: int, val, lambda):
  gathered.merge(req_id, ix, val)
  if (gcount[req_id] >= acount):
    result = reduce(lambda, gathered[req_id])
    gathered[req_id].tombstone.merge(true)
    return result

on mpi_allgather(req_id: int, ix: int, val):
  result = mpi_gather(req_id, ix, val)
  if (result):
    send mpi_bcast(req_id, result)

on mpi_allreduce(req_id: int, ix: int, val, lambda):
  result = mpi_reduce(req_id, ix, val)
  if (result):
    send mpi_bcast(req_id, result)
```

Note that these are naive specifications, and there are various well-known optimizations that can be employed by Hydrolysis, including tree-based or ring-based mechanisms.