Automatic Compartmentalization of Distributed Protocols

Graduate

1 Introduction

Promises of better cost and scalability have driven the migration of systems to distributed implementations in the cloud. To work correctly in a distributed setting, systems must address asynchronous messaging, inconsistent state, and machine failures, leading to the design of complex protocols like Paxos [8]. As systems scale and distributed protocols become the bottlenecks, scalable variants [5, 6, 10] have emerged. However, these protocols are highly specialized to specific workloads, complicating the challenge of choosing the right protocol for every different environment. In addition, creating new distributed protocols is a tricky process; variant protocols are often even more intricate and frequently error-ridden [10].

Recent results suggest a simpler, systematic approach to optimizing distributed protocols without fundamentally changing their logic. Compartmentalized Paxos [11] identifies "compartments" of a distributed protocol that can be **refactored** to allow for **partitioning** and **replication**, common scaling techniques in distributed systems. Refactoring splits logical components in traditional protocols into independent subservices that are more amenable to optimization. Replication can lower latency and improve availability, while partitioning can split the load of requests across many machines. Whittaker et al. [11] show how to manually refactor consensus protocols and scale them up with replication and partitioning *without* a fundamental redesign, in a technique they coin "compartmentalization". Compartmentalization, however, is not trivial: to preserve original semantics while scaling, the authors had to reason through each protocol's inner workings. We aim to automate this process by creating an optimizer with the ability to perform protocol analysis and optimization.

Distributed protocols are complex because they must tolerate non-deterministic ordering: asynchronous messages arrive at non-deterministic times, triggering concurrent state changes and responses. **Declarative logic programming** can help us find the protocol properties we need in order to automatically apply optimizations. In declarative logic languages, all core data structures holding program state are unordered collection types, such as sets or relational tables. Moreover, program logic is expressed not as sequences of instructions, but as unordered sets of inference rules consisting of pre- and post-conditions. The default assumption of sequentiality in imperative programming adds unintentional data entanglement; the naturally *unordered* state and code in declarative languages allows for analysis of intent instead of implementation. Declarative logic rules—with their pre- and post-conditions—are also a natural fit to the request-response paradigm used throughout distributed protocols.

We combine the simple protocol transformations introduced by Whittaker et al. [11] with the clarity of intent offered by declarative logic programming to make the following contributions:

- We present AUTO-JOHN, an optimizer that automatically identifies opportunities for optimizations in a distributed protocol and applies them in a provably correct way. We are able to refactor (Section 4) a program into compartments, such that we can automatically and correctly apply partitioning (Section 6) and replication (Section 5).
- We use a simple syntactic check for *monotonicity* [1] to identify subsets of programs whose outputs "increase" with more input and are insensitive to order of message processing (Section 4.1). We apply monotonicity to simplify refactoring and replication.
- We exploit axioms about *dependencies* [1] that provide invariants concerning the relationship between data attributes (Section 6.1). We re-purpose dependency analysis — traditionally used to reason about database update anomalies — to analyze the safety of state- and program partitioning.
- We show initial results in applying transformations to a canonical design pattern found in a wide variety of distributed protocols such as 2PC, consensus, and blockchains, demonstrating the viability of automatic compartmentalization (Section 7).

2 Background and Related Work

AUTO-JOHN combines existing work on methodical optimizations for distributed systems and the formal clarity and ease of analysis offered by declarative logic programming.

2.1 Optimizing Distributed Systems

Compartmentalization [11] identifies scaling opportunities *without* changing program semantics. This process, however, is based on manual program rewriting, requiring expert knowledge in order to preserve correctness. Our goal therefore is to design an optimizer that automates compartmentalization for scaling any distributed program, removing the need for expert analysis of each individual protocol.

Our work is inspired both by prior work that demonstrates the impact of simple protocol optimizations [5, 11] as well as

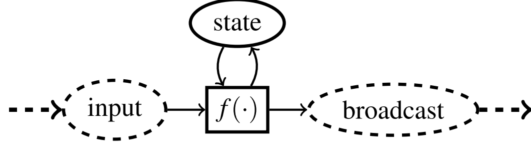

Figure 1: A single-machine program that receives incoming messages through `input`, processes messages in batches locally with some function f, then broadcasts the result while applying updates local state.

work that employs declarative logic programming to detect where coordination is necessary in a distributed protocol [2, 3]. To the best of our knowledge, AUTO-JOHN is the first to automatically apply optimizations to distributed protocols in a provably correct way.

2.2 Programming Model

We analyze and apply optimizations to distributed protocols implemented in Dedalus [4]. Dedalus is a subset of Datalog$^\neg$ designed for implementing distributed systems; Datalog$^\neg$ is a declarative logic language that extends the relational calculus of SQL with recursion, aggregation (akin to `GROUP BY` in SQL), and negation (akin to `NOT IN`). Dedalus differentiates from Datalog$^\neg$ by appending two extra attributes to all values: a location attribute containing a machine identifier, and a time attribute containing a logical timestamp. By adding formalisms for time and space, Dedalus can model distributed systems while inheriting formal results over Datalog$^\neg$.

For readability, we represent Dedalus programs as dataflow graphs: Figure 1 shows a simple example with a single-machine. Asynchronous message passing between machines is represented by dotted edges, functions by square nodes, and data structures by ellipses. Unlike imperative programming, concurrent processing is the default in Dedalus: multiple values can be processed in parallel for any given step, and steps in the pipeline can be executed in parallel as well.

3 Approach

Our main insight comes from viewing compartmentalization through the lens of declarative logic languages. We observe that the correctness of compartmentalization rests on two well-known properties from database theory: functional dependencies and monotonicity (defined more formally in Sections 4.1 and 6.1). To the best of our knowledge, this is the first work where these concepts are applied to program transformations.

We leverage this observation to develop automatic compilation techniques that analyze declarative logic programs for these properties, and automatically transform these programs into scalable variants that provably preserve semantics. We validate our optimizations with formal proofs, excluded for space; a practical implementation is ongoing.

Our compiler has two phases. The first phase focuses on *safe scalability*: we use the techniques of compartmentalization to rewrite the given program into a new yet semantically identical program that exposes opportunities to correctly scale-up performance. The second phase focuses on *performance tuning*: it takes the result of the first phase, and determines optimal values for parameters such as the number of replicas of a program component, the degree to which a task is partitioned across machines, and the assignment of workers to machines.

In this initial paper, we focus only on the first phase, which consists of three parts: refactoring (Section 4), replication (Section 5), and partitioning (Section 6).

4 Refactoring

Refactoring splits a monolithic program into disjoint sub-services that communicate through message passing. This transformation has two benefits. First, the resulting sub-services can be *flexibly deployed*—either colocated on the same machine, or separated across multiple machines. Deploying sub-services on separate machines enables pipeline parallelism, alleviating the load on a single machine, at the expense of additional latency due to message passing. Second, and more significantly, the refactored sub-services can then be *individually* analyzed for optimizations. Optimizations like partitioning and replication often cannot be applied to programs as a whole, but can be applied to some subset of the program. By first refactoring, our optimizer produces a range of semantic-preserving sub-services to apply optimizations to. This is the heart of the success behind Compartmentalized Paxos [11].

Our primary challenge is ensuring safety: the resulting composition of sub-services must preserve the logic of the original single-site code, even as it introduces non-determinism through added asynchronous communication.

To provide intuition for refactoring, we consider the example in Figure 1. Incoming messages are processed through a function f, whose output is appended to `state` and broadcasted. We can refactor this code into two sub-services, following a pattern seen in Compartmentalized Paxos: the use of a *remote proxy* to offload core logic. Figure 2 summarizes the refactored code: the initial *sender* sub-service (in yellow) continues to compute function f and update the local state. However, a second component (in blue) serves as the proxy that offloads the work of broadcasting from the sender. Intuitively, the additional asynchronous messaging (dashed line in the Figure) seems safe: messages in the original program would have arrived at their final destinations with arbitrary delays and reordering, so initiating the broadcast task via an asynchronous message cannot introduce any reorderings or resends that were not already expected at the receivers.

Of course, we must formalize this intuition to apply it cor-

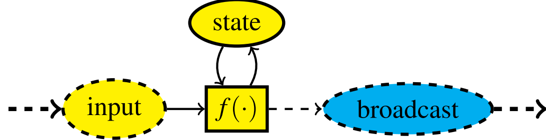

Figure 2: Refactored Figure 1, moving broadcast to another sub-service. Each color represents a separate sub-service.

rectly to arbitrary programs. Given a component that runs synchronously on a single machine, when can we safely divide this logic into disjoint sub-services executing on different machines? To answer this question, we make use of monotonicity analysis and the CALM Theorem [7]. To the best of our knowledge, this is the first time that monotonicity analysis has been used for program rewriting.

4.1 Monotonicity

The familiar notion of monotonicity is that some property "grows" and never "shrinks". In logic programming, monotonicity specifically refers to growth in the contents of sets: given a monotonic logic program P, we are assured that if $S \subseteq T$ then $P(S) \subseteq P(T)$.

The CALM Theorem [7] showed that distributed monotonic logic programs are *confluent*: their state will eventually converge to a deterministic result once all messages are received, regardless of reordering or resending. Monotonic code can thus be distributed across machines without coordination. Conveniently, the CALM Theorem also provides a simple (but not comprehensive) test for monotonicity in Dedalus that depends only on syntax: a Dedalus program is monotonic if it does not make use of negation nor aggregation. Armed with this simple analysis, we can easily identify monotonic code.

Returning to Figure 1, the component does not contain any use of negation or aggregation, aside from the function f, which may perform arbitrary operations over state. In particular, we can introduce asynchronous messaging between f and the broadcasting of its results without affecting the program's eventual outcome. The rewrite into Figure 2 does precisely this: it introduces asynchronous messaging but otherwise leaves the program semantics untouched. Other services will not be able to distinguish the rewritten service from the original, except perhaps via latency if the sub-service is placed on a separate machine.

Generalizing this intuition, our optimizer includes transformation rules that take a component as input, and refactors monotonic subprograms into sub-services linked through asynchronous communication.

4.2 Non-monotonicity and Coordination

Up to this point we have considered the simple case of refactoring monotonic code. Unfortunately, the majority of distributed protocols are not fully monotonic. However, they are often *mostly* monotonic: the use of non-monotonicity is often limited to a small fraction of the the program logic. Our optimizer must correctly handle the interaction between monotonic and non-monotonic code when refactoring. We defer careful discussion to the full paper, but briefly describe our approach here. Informally, any code becomes trivially "monotonic" when all of its inputs have been received (sealed), because no further inputs exist to affect the output. We leverage existing sealing techniques [2] using punctuations from stream processing to correctly handle non-monotonic logic in otherwise monotonic code when refactoring.

5 Replication

Replication duplicates program logic onto multiple machines. It can be used to reduce a straggler's impact on latency, improve fault tolerance, and distribute load. Replicating functionality when the logic is monotonic is, once again, straightforward. Arbitrary replica states diverge due to both message reordering and message duplication; monotonicity provides resilience against both "anomalies". Refactoring isolates monotonic sub-services, which can be trivially replicated while preserving semantics; senders into the monotonic component simply broadcast messages to all replicas.

The main challenge of replication stems from how to safely *read* values from replicated sub-services and perform non-monotonic operations while preserving program semantics. Reading requires collecting outputs across replicas, deciding which messages belong in that output set (sealing), and calculating a value to represent the state of all replicas (merging). Coordination with punctuation from Section 4.2 can be used to implement sealing. Safely selecting a value to represent state across replicas remains ongoing work.

6 Partitioning

Partitioning shards a refactored sub-service across machines; each machine processes a subset of the data, allowing the system to scale with the number of machines.To correctly partition the system, we only partition sub-services with provably independent data processing pipelines. Data pipelines are independent if inputs across pipelines do not interact. Our challenge in partitioning is finding independent data pipelines and modifying the program while preserving program semantics.

Returning to the refactored example in Figure 2, the refactored proxy is trivially partitionable. Each proxy can handle a subset of messages to broadcast. Outputs of f can be arbitrarily routed to individual proxies, as shown in Figure 3.

Now consider a more complex example involving the use of joins ($\bowtie$) in Figure 4, which models a snippet of a 2-Phase

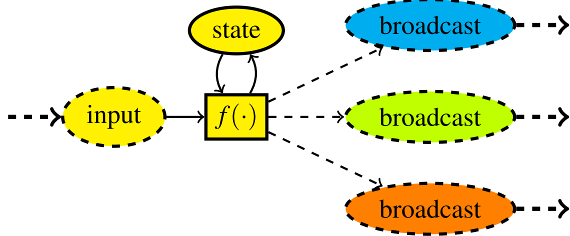

Figure 3: Partitioned broadcast from Figure 2. Each color represents a separate sub-service.

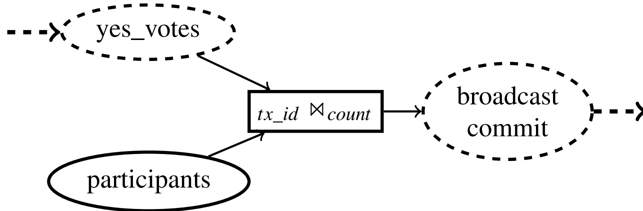

Figure 4: A snippet of code from a 2PC coordinator, which broadcasts commit messages for a transaction when the number of `yes_votes` *received is equal to the number of* `participants`*. The notation $_{tx_id} \bowtie_{count}$ translates to* `SELECT COUNT(*) FROM tbl1 JOIN tbl2 GROUP BY tx_id` *in SQL.*

Commit (2PC) coordinator. The coordinator receives incoming `yes_votes` from participants, groups them by transaction ID (`tx_id`), then checks if the number of votes received is equal to the number of `participants`, a list of machine IDs provided at compile time. If the counts match, the join produces a value, which is broadcasted.

A quick inspection reveals that this program can be partitioned by `tx_id`. Concretely, partitioning entails duplicating logic seen in Figure 4 and modifying any sender sub-service (with messages arriving at `yes_vote`) such that messages are forwarded to the correct partition. Given n available machines, $m_0, \ldots, m_{n-1}$, one possible partitioning scheme sends all `yes_votes` with `tx_id`$\equiv 0 \bmod n$ to m_0, all `yes_votes` with `tx_id`$\equiv 1 \bmod n$ to m_1, etc.

Figure 4 can be safely partitioned on `tx_id` because unique transaction IDs define independent data pipelines. Intuitively, `yes_votes` with different values for `tx_id` do not interact; the number of `yes_votes` is calculated *independently* per `tx_id`, similar to `GROUP BY` in SQL. A partition that collects votes for some `tx_id` does not need to consider how many votes were received for another `tx_id`. In fact, this is true across all rules in 2PC coordinators, which can be partitioned entirely without refactoring! To determine whether independent data pipelines exist for general Dedalus programs, we must formalize our intuitions for analyzing such pipelines.

We repurpose an existing database partitioning technique called *co-hashing* [9] to statically identify when partitioning is safe. In a relational database, if two tables T_1 and T_2 are

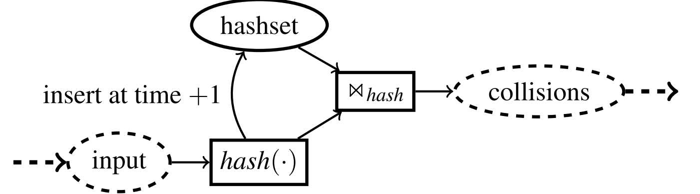

Figure 5: A hashset implementation that finds hash collisions between `input` *and* `hashset`*, then inserts the input and its hash into* `hashset` *in the next timestep.* `hash` *is a function that outputs a tuple,* `{input, hash(input)}`*.*

joined on attributes $T_1.A$ and $T_2.B$, then we can partition by those attributes: distribute rows of T_1 by hashing on $T_1.A$, then forward rows of T_2 to the partition where $T_1.A = T_2.B$. Partitioning via co-hashing is only coordination-free if T_1 always joins on A; otherwise, databases must add an additional shuffle step, introducing coordination as all partitions wait for the complete set of shuffled data to arrive. Furthermore, for all tables that T_1 joins with, those tables must also use the same column consistently across its joins, propagating a set of restrictions on how tables must join for partitioning to be coordination-free.

These requirements can be analyzed at compile-time by inspecting joins across data structures in a Dedalus program, formalized as sets and relational tables. If all data structures are consistently joined on the same attributes, then the program can be partitioned on those attributes.

This analysis may be too strict for some programs, however: there exist partitionable programs that do not consistently join over the same attributes, but are partitionable due to known *dependencies* between the joined attributes. We describe such an example in Figure 5, which implements a hashset that finds the hash `collisions` of an `input`, then inserts the input with its hash into the `hashset`.

A careful manual analysis will show that Figure 5 can be partitioned by `hash(input)`. All inputs with the same hash must be sent to the same partitions so hash collisions can be detected; inputs with different hashes are independent. Arriving at this conclusion is non-trivial for an automatic optimizer: `input` does not contain any information about its hashed value until it *enters* the sub-service. A sender with some `input` must "pre-compute" its hash before determining which partition to forward it to. Our manual inspection revealed a partitioning scheme which required applying a function (`hash`) to the input, a conceptual leap more powerful than co-hashing. To automatically discover such possibilities for partitioning, we perform dependency analysis.

6.1 Dependencies

In classic database theory, a functional dependency $T.A \rightarrow T.B$ states that for any two rows in table T, if they have the

same value for A, they must have the same value for B; column A determines column B. In Figure 5, `input` determines `hash(input)`. `input` is not an eligible attribute for partitioning because it is not used to join; intuitively, if the program were partitioned on `input`, another value that has the same hash may not be forwarded to the same partition, leading the join that calculates `collisions` to miss. `hash(input)`, however, is eligible: it is used to join for the calculation of hash collisions, and it is wholly dependent on `input`. Therefore, partitioning over `hash(input)` guarantees successful joins. In general, understanding functional dependencies between attributes of data provides us more flexibility in partitioning.

We can find dependencies for each data structure in a Dedalus program by tracing their sources of data. We demonstrate the dependency tracing process by analyzing `hashset` in Figure 5. Values in `hashset` are only introduced via `hash(input)`, so we must trace through the dependencies of `hash(input)`. Assuming the existence of built-in operations (or user-defined lookup tables) with annotated functional dependencies — such as `hash`, `encrypt`, string and arithmetic operations — we know that `hash` is a function with a dependency from a value to its hash. This functional dependency (FD) is passed from `hash` to `hashset`.

To generalize, FD analysis boils down to two rules:

1. If there is only one "constructor" (as in Java constructors) for values in a data structure, then FDs for that data structure are inherited from its sources, based on how values are joined and which FDs exist between source attributes.

2. If there are multiple constructors for values in a data structure, then the constructors may produce values with different FDs. Only the *intersection* of FDs generated across the constructors are retained, because functional dependencies must *guarantee* a relationship between all values of a data structure.

Dependency cycles complicate analysis. We provide an algorithm for analyzing FDs with provable termination by first taking the union of all possible FDs generated across rules, propagating FDs to generate transitive FDs until fixpoint, then taking the intersection of FDs generated across rules.

In order to automatically partition with dependencies, we must also determine how attributes *across* data structures are joined, not just what properties are guaranteed between attributes *within* a data structure. We coin these join conditions **co-partition dependencies**, which can be arbitrary functions and analyzed using a similar technique. For co-hashing, all co-hashed attributes must be joined via equality; co-partition dependencies relaxes joins to cover arbitrary functions. Full details of the use and analysis of co-partition dependencies are omitted for space.

7 Results and Contributions

Our initial results are promising. We modify the Dedalus program described in Figure 1, replacing f with layered hashes on the input $hash_1(hash_2(\ldots(hash_{1000}(input))))$ and broadcast to 11 machines. We then apply refactoring and partitioning to generate the program captured by Figure 3, which is translated into Hydroflow, a fast dataflow runtime implemented in Rust (we will eventually automate this step). On an AWS c1.xlarge EC2 machine (7GB memory, 8 cores equipped with Intel Xeon processors), we can scale the throughput from 4,300 op/s to 7,900 op/s with 6 broadcasting proxies, with similar performance with a direct implementation in Rust. We are working towards evaluating AUTO-JOHN on Paxos and comparing against manual optimizations applied in [11].

References

[1] ABITEBOUL, S., HULL, R., AND VIANU, V. *Foundations of Databases*. Addison-Wesley, 1995.

[2] ALVARO, P., CONWAY, N., HELLERSTEIN, J. M., AND MAIER, D. Blazes: Coordination analysis and placement for distributed programs. *ACM Trans. Database Syst. 42*, 4 (Oct. 2017).

[3] ALVARO, P., CONWAY, N., HELLERSTEIN, J. M., AND MARCZAK, W. R. Consistency analysis in bloom: a CALM and collected approach. In *Fifth Biennial Conference on Innovative Data Systems Research, CIDR 2011, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings* (2011), pp. 249–260.

[4] ALVARO, P., MARCZAK, W. R., CONWAY, N., HELLERSTEIN, J. M., MAIER, D., AND SEARS, R. Dedalus: Datalog in time and space. In *Datalog Reloaded* (Berlin, Heidelberg, 2011), O. de Moor, G. Gottlob, T. Furche, and A. Sellers, Eds., Springer Berlin Heidelberg, pp. 262–281.

[5] CHARAPKO, A., AILIJIANG, A., AND DEMIRBAS, M. PigPaxos: Devouring the communication bottlenecks in distributed consensus. In *Proceedings of the 2021 International Conference on Management of Data* (June 2021), ACM.

[6] DING, C., CHU, D., ZHAO, E., LI, X., ALVISI, L., AND RENESSE, R. V. Scalog: Seamless reconfiguration and total order in a scalable shared log. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)* (Santa Clara, CA, Feb. 2020), USENIX Association, pp. 325–338.

[7] HELLERSTEIN, J. M., AND ALVARO, P. Keeping CALM. *Communications of the ACM 63*, 9 (Aug. 2020), 72–81.

[8] LAMPORT, L. The part-time parliament. *ACM Trans. Comput. Syst. 16*, 2 (May 1998), 133–169.

[9] SUNDARMURTHY, B., KOUTRIS, P., AND NAUGHTON, J. Locality-aware distribution schemes. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

[10] SUTRA, P. On the correctness of egalitarian paxos. *Information Processing Letters 156* (2020), 105901.

[11] WHITTAKER, M., AILIJIANG, A., CHARAPKO, A., DEMIRBAS, M., GIRIDHARAN, N., HELLERSTEIN, J. M., HOWARD, H., STOICA, I., AND SZEKERES, A. Scaling replicated state machines with compartmentalization. *Proc. VLDB Endow. 14*, 11 (July 2021), 2203–2215.