Distributed Dataflow Across the Edge-Cloud Continuum

Tyler Ekaireb
Department of Computer Science
University of California, Santa Barbara
Santa Barbara, USA

Lukas Brand
HAW Landshut
Landshut, Germany

Nagarjun Avaraddy
Department of Computer Science
University of California, Santa Barbara
Santa Barbara, USA

Markus Mock
HAW Landshut
Landshut, Germany

Chandra Krintz
Department of Computer Science
University of California, Santa Barbara
Santa Barbara, USA

Rich Wolski
Department of Computer Science
University of California, Santa Barbara
Santa Barbara, USA

***Abstract*—Internet of Things (IoT) applications span the edge-cloud continuum to form multiscale distributed systems. The heterogeneity that defines this architecture, coupled with the asynchronous, event-triggered and failure-prone nature of these deployments create significant programming and maintenance challenges for developers of IoT applications.**

To address this impediment to innovation, we present LAMINAR, a dataflow programming model for IoT applications implemented using a novel log-based and concurrent runtime system that spans all resource scales. We describe the properties that underpin LAMINAR's design and compare it to a lower-level event-based approach. We show that LAMINAR's dataflow model hides many of the complexities of "lock-free" event-driven programming. Through an empirical evaluation of LAMINAR, we find its design and implementation are both more straightforward for developers and more performant.

I. INTRODUCTION

The "Internet of Things" (IoT) is a distributed computing infrastructure in which collections of devices (all network-enabled) communicate with each other and with services at the network edge or in the cloud, to perform data acquisition, analysis, and intelligent actuation and control. To use this paradigm, developers must design their applications to be both robust to failures (which are common) and *multiscale*, i.e., able to leverage battery-powered or resource-constrained sensors, single-board computers, mobile devices, and public/private clouds that comprise the edge-cloud continuum. Current IoT application development technologies [1], [2], [3], [4], [5], [6] are commonly derived from cloud technologies and combine independently-developed software stacks that interoperate via a diversity of network protocols. Such amalgamations are non-portable, error-prone, require significant developer expertise, and are difficult to deploy and maintain at scale. New programming systems are needed to ease this burden and expedite development of robust, multiscale IoT applications.

Toward this end, we present LAMINAR, a log-based, unified, distributed dataflow programming system for IoT. LAMINAR uses a two-tier approach to simplify distributed IoT application development by using a high-level distributed dataflow programming model to hide the low-level details of a failure-resilient (but very difficult to program) multiscale, log-based runtime (the MLR). The MLR is a distributed serverless system in which stateless functions are invoked in response to events. Program state is persisted via network-transparent logs that are append-only and lock-free (i.e., logs cannot be locked while an append is in progress). Logs are less complex and more resilient than their file system and database counterparts, and they facilitate causal tracking and failure recovery [7], [8], [9]. They can also be implemented efficiently at all device scales, even as low level as microcontrollers [10]. As a result, the MLR supports execution on heterogeneous devices – from very resource-limited devices to resource-rich cloud servers. This combination of distributed programming features makes the MLR both portable and extremely robust to failures. However, it also makes it very difficult to program correctly: its programming model is event-driven, highly concurrent, lock-free, log-based, and distributed over devices with a wide range of resource constraints.

To tame such a platform, we design and implement a universal execution model for multiscale, distributed dataflow that sits atop the MLR. Dataflow is a programming paradigm that formulates programs as directed graphs in which nodes are data-parallel functions and edges represent the flow of data between them. Nodes fire when all their inputs are present; thus function execution order is determined solely by the program's data dependencies. Focusing on data instead of control flow makes this paradigm well-suited to stream processing, event-driven computing, and highly parallel and concurrent workloads (all characteristics of modern IoT applications). Dataflow has been used successfully for asynchronous systems [11], [12], [13], parallel computing [14], [15], [16], and more recently for big data analytics [17], [18], [19] and visual programming [20], [21], [22].

We posit that despite these advantages, dataflow has heretofore yet to be explored as a distributed programming model in *multiscale* IoT settings primarily because it is difficult to implement its runtime semantics efficiently in heterogeneous

and error-prone networked environments. Dataflow semantics do not include ways to represent typical distributed systems modalities, such as partial failures, duplicated messages, and network partitions.

To address this challenge, LAMINAR separates the concerns of the runtime from those of the programming model. In particular, we show that it is possible to express dataflow programming semantics using append-only logs. By specifying stateless computations and append-only persistent storage as abstractions supported by the MLR, LAMINAR programs exhibit referential transparency [23], [24], which is useful for optimization and for building fault-tolerant applications [25], [26], [27]. Finally, LAMINAR can leverage the dataflow representation to provide visualization tools that aid the analysis and debugging of distributed programs.

With this paper, we make the following contributions:

- We describe the properties of the MLR and provide a brief rationale for its design.
- We describe the efficient implementation of typed dataflow programs using the MLR as a target runtime.
- We develop program optimizations that exploit the compositional properties of dataflow to control concurrency.
- We measure the efficiency of the approach using IoT applications and benchmarks.

We find that LAMINAR simplifies the development of distributed event-based programs. Combined with its type system, this yields substantial performance benefits, further enhanced by the parallelism management that dataflow enables. Despite the pessimistic assumptions about resource availability, we find LAMINAR's performance to be comparable with other IoT technologies that assume failures are rare. Thus, we conclude that LAMINAR is an effective, unifying programming system for distributed IoT applications.

II. Related Work

LAMINAR has several antecedents. Early functional languages, such as Id [28], FP [29], Haskell [30], ML [31], SISAL [32], and Lucid [33] all demonstrated the feasibility of using dataflow as a runtime system, either with hardware support such as i-structures [34] or purely in software. LAMINAR shares the goal of automated and/or programmer-guided management of parallelism and robust distributed execution with these early systems. Modern functional languages (including Haskell), such as Miranda [35] and Purescript [36] can also use dataflow as a runtime system, although they often compile to a more imperative language variant, such as JavaScript [37]. LAMINAR's visual programming component is inspired by programming systems such as GNURadio [20], LabView [21], Keysight VEE [38], KNIME [39], and Node-RED [22]. LAMINAR includes a similar graphical representation capability but is distinct from these systems in that it implements a dataflow runtime using a log-based, append-only storage model and triggered execution.

More recent distributed dataflow systems target big data workloads [17], [40], [18], [19]. These systems provide a simple programming model for large-scale, parallel processing of structured and semi-structured data on commodity clusters or cloud servers. Their execution engines automatically schedule, place, synchronize, and manage faults for these workloads. MapReduce [17], [40] represents programs as a bipartite graph and Dryad [18] uses a more general directed acyclic graph (DAG) (like LAMINAR). These early systems have been extended in multiple ways to reduce their restrictions and support a wider range of algorithms with greater efficiency [41], [42], [43], [44]. Ciel [19] extends these programming models with better support for iterative computations. Specifically, it adds dynamic control flow creation while maintaining fault resiliency. Unfortunately, none of these past works support multiscale or wide-area settings because they were designed for resource-rich systems. Although Ciel is more dynamic, we find that static specification of a deployment that is fault resilient (nodes can come and go) works well for IoT applications. Moreover, as Section III-I shows, LAMINAR provides support for iteration within the dataflow language without the added complexity of dynamic DAG generation.

III. LAMINAR

LAMINAR uses a two-tier approach to simplifying distributed IoT application development. The lower layer is a distributed runtime system (called the MLR) that executes across a vast diversity of IoT devices that span the edge-cloud continuum, i.e., it is multiscale. This runtime layer provides a set of properties and abstractions that facilitate fault-tolerant, event-driven computing. We describe how to extend these features to also support the programming model of the upper layer. The upper layer is a dataflow execution engine that hides many of the details of the lower layer while benefiting from its robustness. We first overview the runtime system and then describe how we integrate these two layers to simplify the development of robust IoT applications.

A. Multiscale Log-based Runtime (MLR)

To support a dataflow programming model, LAMINAR requires the following properties from the MLR:

- program state variables, i.e., the data values that flow along the edges in a DAG, are single-assignment,
- stateless computations synchronize only as a result of the communication of data between them,
- names of state variables are network-transparent,
- and computations cannot use locks to implement message-based synchronization.

The first two are generic requirements for any system implementing dataflow language semantics. The latter two are motivated by our experiences implementing IoT applications in low-infrastructure or weak-infrastructure settings. In these environments, power and network infrastructure may be intermittently available, devices routinely fail, or malfunction, and undetected device "upgrades" or replacements can cause latent data integrity errors. It must be possible to redeploy state in response to these dynamics, e.g., using network transparency and threads or other concurrency abstractions should not hold locks waiting for data communication (that may never

arrive). Our experience with POSIX "timed" locks and thread cancellation [45] further supports this requirement.

It is possible to meet many of these design requirements using cloud-based and edge-based Functions-as-a-Service (FaaS) or serverless technologies. By restricting their use to trigger stateless computations in response to single-assignment storage events, e.g., using a database to implement a log, it is possible to implement the MLR using CloudPath [46], tinyFaaS [47], AWS Greengrass [48], and Azure IoT Edge [49]. Heavier-weight FaaS systems, e.g., AWS Lambda [50], Azure Functions [51], Google Functions [52], OpenWhisk [53], and OpenFaaS [54] can also be used. However, they require special purpose libraries, systems, and protocols (e.g., AWS SDK, FreeRTOS, MQTT, etc.) to support very resource-restricted edge devices. In this paper, we use CSPOT [10] to implement the MLR. CSPOT is a simple, lightweight, distributed serverless runtime (available as open source) that executes FaaS applications across heterogeneous IoT deployments [10]. It uses logs to hold program state and track events across the system. Logs are circular buffers with programmable history and element size optionally persisted to disk via memory-mapped files (i.e., for devices with file system support).

B. MLR Properties and API

LAMINAR leverages the following MLR properties.

- All program state variables communicated between computations are implemented as append-only logs. Computations are otherwise stateless.
- A successful log-append returns a unique sequence number for the resulting log entry, and sequence numbers are strictly increasing.
- Computations (referred to herein as functions or handlers) can only be triggered in conjunction with a single log-append event.
- Logs are named using a network-resolvable name (e.g., a Universal Resource Name) for network transparency.
- Computations can only synchronize using log sequence numbers explicitly. That is, the MLR is lock-free and includes no provision for locking one or more logs while an append is in progress.

LAMINAR requires support (via an API) for log create and delete, log read (with or without a sequence number), log append (with and without triggering a handler function), and the ability to get the latest sequence number of a log. LAMINAR assumes that an append event has a monotonically increasing sequence number associated with it, which it uses for log scans. Synchronization between computations is in terms of log sequence numbers (i.e. computations decide whether to proceed based on a comparison of log sequence numbers associated with log elements). Further, computations can only be triggered as a result of some log advancing. That is, only new state (appended to some log) results in a new computation being initiated. Finally, log wraps, if any, must be detected and reported as errors if/when they occur.

From the perspective of an applicative functional language, and equivalently, dataflow, logs and append-only semantics correspond to single-assignment variables. In effect, each variable in an MLR program is versioned, and each version is immutable. Thus, an implementation of dataflow in which MLR functions are stateless and all program state is stored in logs, layers applicative programming semantics atop the MLR.

C. Log-based Synchronization

The MLR design is "lock-free" in that there are no provisions for a computation to block and wait (perhaps in a loop) for one or more events to transpire. For example, it includes no explicit primitives for implementing mutual exclusion, "test-under-lock" (e.g., `pthread_cond_wait/signal`) control blocks, or concurrent computation "joins." While convenient and undoubtedly useful in a single-machine setting, such a facility can present difficulties in a distributed setting. Specifically, it is possible for computations to "block on the tail" and for the system to crash. When it is restarted, these computations would need to be recovered in the state they were in and re-blocked before any log appends originating from remote machines are accepted.

In short, computations would need a checkpoint facility that is synchronized with the log to implement a persistent "wait for advance" capability when a node failure does not imply a fail-stop. The MLR exposes this facility through the log APIs. As a result, the MLR itself need not include lock-timeouts, explicit critical section detection and recovery, computation cancellation, etc., all of which can be an impediment to implementation in resource-restricted environments.

Further, an MLR program can never "deadlock" due to node failure with computations holding locks. It certainly can stop because a computation that is responsible for advancing the state of the overall program has been permanently lost, but when it does, there are no pending threads (address spaces, stacks, etc.) that are suspended, indefinitely holding resources pending a full reset of an entire deployment. When an MLR program stops, all of the program, state is "at rest" in the MLR logs, and none is stored in the address spaces or stacks of pending computations.

This design decision enables two essential features. The first is that a (distributed) MLR program can be paused and resumed based on the contents of the logs. The second is that (with the optional inclusion of dependency information in each log entry) it tracks causal dependencies as a debugging aid. That is, the logs can be configured to record the identity (log name and append sequence number) of each state advance that triggers a computation. A complete log, then, captures the causal order for all program state changes.

These features come at the expense of program clarity (relative to modern concurrency abstractions) and, as a result, programmer productivity. While serverless runtime systems enable higher-level programming approaches to be applied in distributed settings, our early experiences with using such systems to develop IoT deployments "by hand" have confirmed their productivity costs when used as a primitive distributed event-based programming platform. To overcome this limitation, we layer a dataflow framework atop of the MLR

to hide its complex programming model, abstractions, and interface from developers, so that more familiar, high-level programming languages can be used.

D. Programming Model

LAMINAR implements *strict* dataflow semantics [55] using the MLR. A LAMINAR program is represented as a directed acyclic graph (DAG). Nodes in the graph represent computations, and edges represent data values transmitted between nodes. A node becomes executable by the runtime system when the values corresponding to all of its input edges are available. Its outputs are available to other nodes only when the node has completed executing.

Developers specify a program as a hierarchical set of DAGs. They also implement the computations associated with each node as an MLR function that "fires" when the node is executed in a LAMINAR program. More specifically, a LAMINAR program consists of:

- **Sources**, which are external computations that introduce data into a LAMINAR program. These include sensor readings (in an IoT context), database reads, remote API calls, or arbitrary program functions from a program or script capable of exercising LAMINAR's API.
- **Nodes**, which perform operations on data using stateless functions written by the programmer.
- **Edges**, which express data flow between nodes.
- **Sinks**, which transmit data outside a LAMINAR program, e.g., database writes, remote API calls, or arbitrary program functions that consume data via LAMINAR's API.

A node can have an arbitrary number of inputs and outputs, each represented by a unique "port". A directed edge links nodes (output port to input port) and represents a "subscription" on the output of a node by the input node. Output ports can have fan-out, but input ports receive data from a single output. Sources and sinks are special nodes. Sources have no input ports but ingress data from outside the program, e.g., program inputs, to the program.

Each node is implemented using two logs: a subscription log and a subscriber log. The subscription log records node inputs that are available to the node (i.e., have been produced as outputs by predecessor nodes in the DAG). A subscription event is triggered for every input that arrives at a node (i.e., every time a data item is appended to the subscription log). The handler checks if all inputs have arrived; if not, it exits. On arrival of the last input, the handler extracts all input values from the subscription log, executes the computation associated with the node and populates its output ports. When a datum is appended to an output port, a subscription event is posted to the subscription log of each node subscribing to that port.

Figure 1 shows a LAMINAR C++ code snippet for implementing $a + b$ using the LAMINAR API and corresponding DAG. The API provides support for DAG specification, initialization, and execution. Developers add hosts (add_host), nodes (add_node), sources (add_operand), and edges (subscribe) to construct an application. They also use the API to initiate compilation (setup), computation (by assigning values to sources, e.g., fire_operand), and to read results (get_result). In this example, there is one host (hd). Node n1 (on hd) is specified to perform the computation (ADD; add_node). There is a source node for both a and b created using add_operand. An edge between the source node outputs is connected to the input edges of n1 (in0 and in1) via subscriptions to n2 and n3, respectively. Each host in the deployment is assigned a unique ID, and all hosts receive a copy of the program; hosts execute the program selectively based on their local host ID. fire_operand initiates execution by assigning the boundary values 7 and 10. We use graphviz [56] to automatically generate hierarchical drawings of LAMINAR program graphs to aid program analysis and debugging.

```
// a + b
...
// Setup hosts
12: add_host(hd, IP, "/logs");
// Create 3 nodes n1, n2, n3
13: add_node(hd, n1, ADD); // a + b
14: add_operand(hd, n2); // a@boundary
15: add_operand(hd, n3); // b@boundary
// Edges
16: subscribe(n1, in0, n2);
17: subscribe(n1, in1, n3);
18: L_setup();
//  Example: (7 + 10)
19: fire_operand(n2, 7);
20: fire_operand(n3, 10);
21: err = get_result(n1, &result);
```

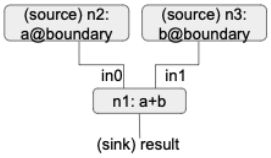

Fig. 1. Using the LAMINAR API: C++ code snippet (left) for the DAG (right) implementing $(a + b)$ for host hd.

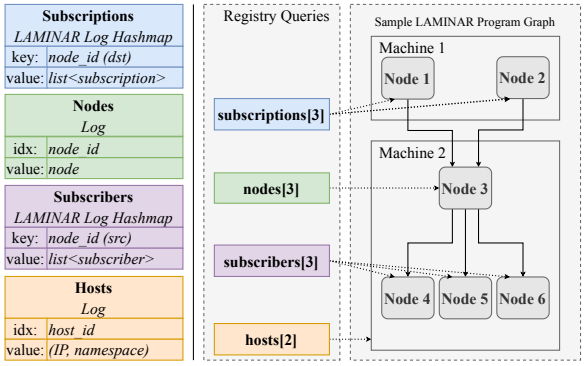

Fig. 2. Data structures associated with a LAMINAR registry. The example indicates which program elements are referenced when each data structure is queried for information regarding `Node 3`.

E. LAMINAR *Program Registry*

The LAMINAR registry consists of four data structures that track *Nodes*, *Subscriptions*, *Subscribers*, and *Hosts*, as shown in Figure 2. The registry implements these data structures using MLR logs. The *Nodes* log stores both the node ID and the host ID to uniquely identify the machine on which the node is running, and an operation ID that is used to dispatch the node's computation when all inputs are available.

Note that the program registry is immutable with respect to the program. That is, even though LAMINAR is using the MLR to implement the registry, the registry contents are fixed when the LAMINAR program is defined. A LAMINAR program executes a preamble to populate the registry (logically part

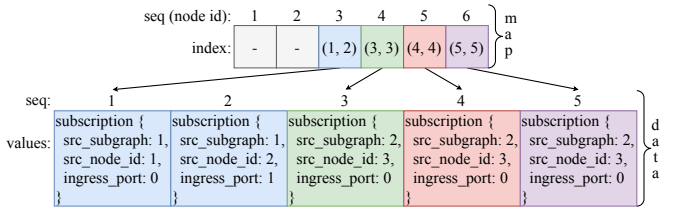

Fig. 3. Detailed view of the Subscriptions structure of the sample LAMINAR program shown in Figure 2. Each consumer node ID is mapped to a list of subscriptions. Each subscription associates a consumer node input port with the producer node's output.

of its compilation and deployment process) before executing DAG computations. The registry is also global, i.e., it must be replicated on all potential execution sites in a LAMINAR deployment. In the current LAMINAR prototype, all program components are compiled, the preamble is executed on one site (usually the compilation site), and the resulting MLR logs and handlers are copied to other hosts in the deployment.

The *Subscriptions* data structure maps node IDs to those of its input edges, enabling fast access when determining whether all inputs have arrived. Subscription information is stored as a hashmap implemented using two MLR logs, as shown in Figure 3. The *map* log maps a range of sequence numbers in the data log to the node to which those subscriptions pertain. Each node's ID is used as an index into the *map* log (the `ReadLog` function takes a sequence number indicating the specific log entry to return). The *data* log stores all subscriptions for each node contiguously in the sequence number space of the log. Each element of the *data* log contains a C++ `subscription` structure that associates an input port of the consumer node with the output of a producer node.

For the example in Figure 3, sequence number 3 in the *map* log (which corresponds to `Node 3` in Figure 2) contains $(1, 2)$, indicating that `Node 3`'s input port edges can be found by scanning the `data` log sequentially from sequence numbers 1 through 2. The input ports for `Node 4` begin at sequence number 3 in the *data* log, and so on.

The hashmap of the *Subscribers* uses the same two-level log encoding to access *subscriber* structures to represent the relationship of the output ports of each node to their consumer nodes. The *Hosts* log stores the information necessary to locate an MLR log remotely, e.g., the host network address and path to the MLR log storage. The LAMINAR runtime uses this log for host discovery and dataflow across hosts.

F. Subgraphs

Nodes can be grouped to implement scoping and modular composition. A subgraph represents a functional "subprogram" that acts as a node in any LAMINAR program in which it is embedded. That is, no node within a subgraph fires until all of the inputs to the subgraph are available, and no outputs from the subgraph can be consumed as inputs by other nodes or subgraphs until all subgraph outputs have been produced.

In LAMINAR, a subgraph implements scope for identifiers and state. Multiple LAMINAR programs can be developed in isolation and deployed together by grouping nodes into subgraphs. Moreover, subgraphs can encapsulate implementation details and provide communication interfaces between programs without exposing graph internals, i.e., to support modular design. LAMINAR uses subgraphs to implement iteration, conditionals, and placement partitioning.

G. Type System

The LAMINAR type system maps program data structures to MLR logs. We refer to these data structures as LAMINAR *Typed Values* (LTVs). The type system supports both primitive and complex LTVs (e.g. arrays, strings, records, maps, etc.), and for the latter, transparently provides (de-)serialization of typed values to/from logs. The current LAMINAR prototype supports C primitive types, arrays, vectors, and strings.

The in-memory representation of an LTV is a tagged union that contains the type identifier and a union of the possible LAMINAR value types. For primitive types, the union holds the primitive value. For complex types, the union type holds a structure that contains type-specific information. This includes data structure size, a pointer to its memory representation (managed by LAMINAR), and a unique ID that identifies an additional MLR log that is used to implement the complex type. This log (called *head*) can contain the literal data value or further substructure descriptors (referring to additional logs) depending on the type.

Figure 4 overviews how the type system works. Data flows from `Node 1` to `Node 2`, and `Node 2` has a single input and output port. When `Node 1` completes and appends to its *Subscribers log*, it triggers `Node 2`'s subscription event handler. The handler calls `Load Value` to read and deserialize the data. It also creates the *Loaded* LTV in memory for use in the operation which uses getters/setters to access the value. The operation uses a separate Loaded LTV for its output port which the handler serializes and appends to `Node 2`'s Subscribers log via `Save Value`, when generated. Primitive types require no (de-)serialization.

For complex types, the Subscribers log LTV contains the unique ID of the head log. Load Value uses it to read 1+ logs and deserialize values as it constructs the Loaded LTV. For instance, for an array of integers, all array values will be loaded in this step, fully constructing the array in memory.

The unique ID is a 16-byte UUID generated by the type system for each log used in a data structure. To construct the log name, the system concatenates the LTV type and this UUID. It stores the UUID (along with the data structure size) in the LTV for easy access and loading.

Figure 5 exemplifies this process using a 2x3 matrix (2 rows of 3 elements each). The matrix is produced by a producer node and appended to its Subscribers log. Its LTV structure is shown on the left and contains the UUID of the head log (UUID1), its LTV type, the element type, and the size (number of rows). The head log contains the UUIDs and structure of two rows: both contain 3 elements (specified in size), and their respective UUIDs. The sub-logs hold the row data which has integer element type.

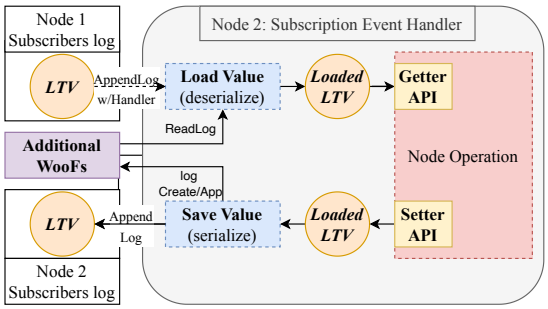

Fig. 4. Overview of the interaction of LAMINAR typed values and the underlying MLR logs.

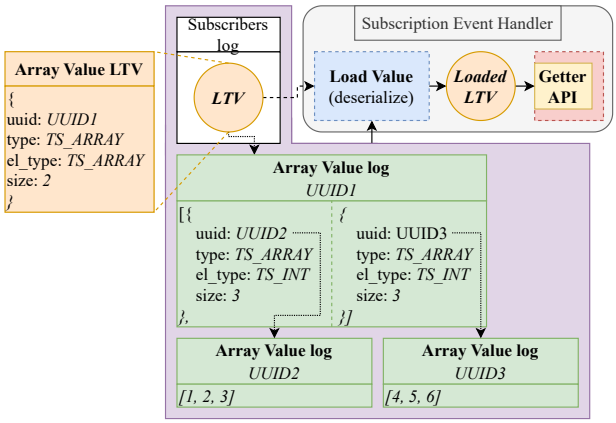

Fig. 5. Storage-persisted representation of a LAMINAR 2x3 matrix.

When the matrix is appended to the Subscribers log, it triggers the consumer node's subscription event handler as in the previous example. This handler allocates space for the matrix and constructs its value using the `Array Value log` (UUID1) and its sub-logs (UUID2 and UUID3). The handler then implements the node's operation using getters/setters to access the matrix directly in memory.

In general, for an NxM matrix, (N+1) memory allocations and `ReadLog` MLR API calls are necessary. Similarly, when a matrix is written, the same number of `CreateLog` and `AppendLog` API calls are performed, in addition to writing the Subscribers log representing the matrix LTV.

The LAMINAR string data type is similar (omitted due to space constraints). We represent strings (with a configurable maximum size) using a single log entry in the common case. When a string exceeds this size, LAMINAR splits it across multiple log entries (and uses an additional field in the proceeding substring log to link them).

H. Integrating Data Sources

A typical IoT application will consume data from multiple sources. As described above, input (i.e. *boundary*) data is represented via *Source* nodes. Developers link input data to source nodes using the LAMINAR API (as shown in the earlier example in Figure 1). Our current prototype supports arbitrary C and C++ functions so that the code can link different data sources (e.g., sensors, GPIO pins, streaming services, databases) to a LAMINAR program. We plan to extend the API with Python bindings as part of future work.

I. LAMINAR *Structured Programming Constructs*

In addition to traditional dataflow, LAMINAR supports constructs that facilitate the development of more complex programs with control flow and iteration.

Conditionals. LAMINAR supports conditional statements `SELECT` and `FILTER`. A `SELECT` node uses its first port as a selector, whose value is used to index the remaining ports to be forwarded as output. The second construct, `FILTER`, accepts a boolean value on its first port, determining whether or not the data on the second port is forwarded.

Iteration. The asynchronous and event-driven nature of LAMINAR naturally supports powerful node-level, i.e., function-level, parallelism since each node operates independently. LAMINAR also supports loop iteration using an approach inspired by IF1 [57], [58] via nested subgraphs. A loop consists of four subgraphs. **Initialization** sets up the loop. **Body** performs the computation that resides within a traditional loop body. **Test** calculates a boolean value, determining whether to exit the loop. **Result** produces the result once the loop finishes. This looping construct and its variants can be used to achieve the same functionality of the popular `for` or `while` constructs in imperative programming languages.

IV. EVALUATION

We evaluate LAMINAR in two ways: comparative performance using IoT benchmarks and clarity of program representation (versus programming events and logs directly). We use a campus cloud, an edge cloud, and resource-constrained single-board computers (SBCs) for this evaluation. The campus cloud VMs run on HP 2.5Ghz x86-blades via KVM. Each VM has eight virtual CPUs and 32 GB of memory and runs CentOS 7.2. The edge cloud (located at a remote research reserve) is equipped with Intel NUCs [59]) each with 3.2 GHz processors is configured to host 2-core VMs with 512 megabytes of memory and large secondary storage for data acquisition. Co-located at the remote site is a Raspberry Pi 3 B+ with 1 GB of memory and a 4-core ARM V7 CPU (running in 32-bit mode) clocked at 1.2 GHz. It is connected to the edge cloud by 1 Gb switched Ethernet. All MLR functions corresponding to LAMINAR nodes are written in C++ and compiled with g++9.

We first demonstrate generality by evaluating IoT benchmarks and comparing LAMINAR against a popular stream-processing-based approach. We then demonstrate how LAMINAR reduces the complexity associated with distributed IoT programming, validate the performance of the type system, and show the potential for performance optimization, using matrix multiply. We choose to include matrix multiply for this programmability study since it is well-understood (has well-understood scaling properties), and provides a familiar example for demonstrating the use, complexity, and optimization potential of LAMINAR.

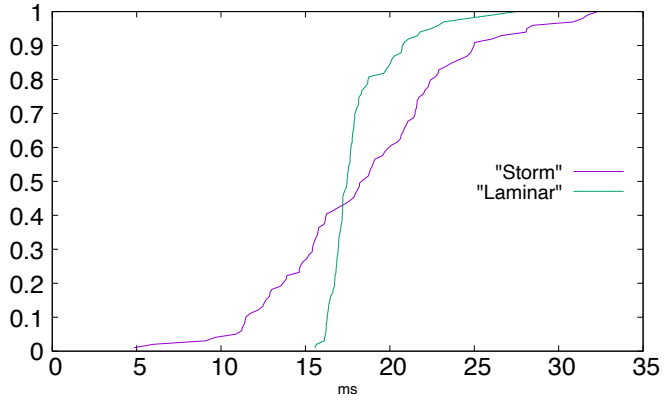

Fig. 6. CDF of end-to-end ETL Benchmark performance for RIoT and LAMINAR implementations. The sample is 100 separate runs of each, and the units on the x-axis are milliseconds.

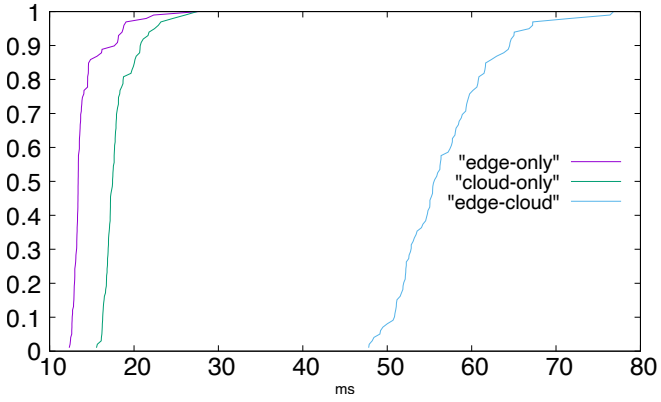

Fig. 7. CDFs of end-to-end ETL Benchmark performance for the LAMINAR implementation of the RIoT benchmark in three different deployments. The sample is 100 separate runs of each, and the units on the x-axis are milliseconds.

A. RIoT Benchmarks

To empirically evaluate LAMINAR's performance and versatility, we used it to implement a subset of the RIoTBench [60] suite. RIoTBench provides a set of realistic applications to benchmark distributed stream processing systems for IoT applications, implemented using Apache Storm and Microsoft Azure [61], [62]. RIoTBench consists of various pipelines for data processing and analysis. We compare the ETL pipeline, which consumes and re-formats sensor data for analysis. The pipeline comprises a source and six microservice stages (parse, range filter, bloom filter, interpolation, annotate, and persist). The source is a workload from the MHEALTH (Mobile Health) dataset [63], a real-world IoT dataset with body motion and vital sign measurements of ten volunteers.

Figure 6 shows a comparison of the empirical cumulative distribution functions (CDFs) for 100 executions of the RIoT ETL benchmark, and the LAMINAR version of the ETL benchmark. Note that the original results reported in [60] are 2 to 3 orders of magnitude slower than the LAMINAR results shown in the figure. To make a fair comparison in this paper, we replaced the Azure storage access at the end of the pipeline in the original benchmark with an additional program node that persists the benchmark results to a local file in the same file system used by LAMINAR to generate results.

The mean execution time for the original RIoT version is 18.6 milliseconds compared to 18.0 milliseconds for the LAMINAR version and a student-t test indicates no statistical difference. However, the data indicates that the standard deviation for the LAMINAR version is lower than for the original.

Note that LAMINAR is persisting the state of *every* operation in every stage of the benchmark. If the Storm version were to replicate this persistence behavior (for the purposes of crash recovery), then the persist stage would be absent, and every other stage would incur its own persistence overhead. We did not feel it reasonable to modify the Storm stages to include such persistence (since there are many ways to do so). Instead, we measured the average duration of the persist stage as 4.6 milliseconds, which is the approximate time necessary to persist the final result of the ETL benchmark to a file. If the Storm-based original implementation were to persist every stage to a file, we estimate the end-to-end latency would be 37 milliseconds ($5 \times 4.6 + 18.6 - 4.6$ ms).

More abstractly, LAMINAR's persistence behavior assumes that in an IoT context, node failures are likely and expected, and that recovery speed should be optimized. At the same time, it can achieve the same performance as the Storm in memory version, which assumes that node failures are rare and only the final results need to be persisted. When this optimistic assumption is relaxed for the Storm version the performance of the LAMINAR version is nearly a factor of two better.

To better understand LAMINAR's ability to operate at different resource scales we deployed it to an edge cloud located in a utility closet at a remote ecological study site approximately 50 miles from our university campus. Network connectivity between the site and the university is via a dedicated, 150-megabit long-range microwave link that (because of the intervening topology and the need for line-of-sight) traverses approximately 120 miles of linear distance.

Figure 7 shows three empirical CDFs of the end-to-end latency for different deployments of the LAMINAR benchmark version. The "edge-only" deployment shows the end-to-end latency for the LAMINAR version of the benchmark when executed entirely in a small VM at the remote site. The CDF marked "cloud-only" is the same CDF shown in Figure 6 for the LAMINAR implementation, and the CDF marked "edge-cloud" shows the end-to-end latency when the first stage of the pipeline in the benchmark is executed at the remote site and the other stages are located in the campus cloud.

Note that we cannot show comparative results for the original RIoT implementation that uses Storm as a runtime platform. Apache Storm does not include a documented way to assign a Storm "bolt" (representing a computation) to a specific host in a Storm cluster. Thus, a distributed deployment (similar to "cloud-edge in the figure) is not possible. Further, the memory available in the VM hosted in the edge cloud is insufficient to execute the Storm benchmark in its entirety.

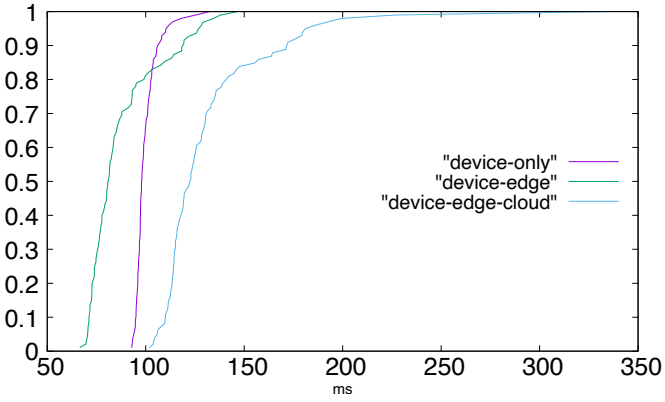

Fig. 8. CDFs of end-to-end ETL Benchmark performance for the LAMINAR implementation of the RIoT benchmark in three different deployments that link a Raspberry Pi to an edge-cloud VM to a campus cloud. The sample is 100 separate runs of each, and the units on the x-axis are milliseconds.

In contrast, the LAMINAR implementation supports both a specific deployment of LAMINAR nodes to hosts and also is memory parsimonious enough to execute the full benchmark in the small edge VM. Further, the only code change needed is to update the host ID in the VM hosted in the edge cloud That is, the LAMINAR code expressing the computation is identical.

It is also noteworthy that the end-to-end latency at the edge is lower than in the campus cloud when the benchmark is hosted entirely in either venue. The reason is that the LAMINAR code "fits" inside the 512-megabyte limit at the edge, making it entirely CPU limited. Thus, its multiscale capability can exploit the faster edge processor in this deployment setting. Also, predictably, when the network link is introduced between stages, its performance dominates the end-to-end latency. Indeed, we have omitted a fourth CDF that places the first four stages of the benchmark in the edge cloud and the fifth in the campus cloud. It is almost indistinguishable from the CDF marked "edge-cloud" in the figure and thus obscures those results at the graph scale shown.

Similarly, to further demonstrate the deployment versatility of LAMINAR, we deployed the LAMINAR implementation of the RIoT benchmark to a Raspberry Pi 3 that is also located at the remote site. This Raspberry Pi is used to gather sensor data that it forwards to the edge cloud VM discussed previously and to reconfigure the sensor duty cycle. We temporarily pause this sensing function during benchmarking.

Figure 8 shows 3 empirical CDFs of three different deployment configurations. The CDF marked "device-only" shows the end-to-end latency distribution when the LAMINAR implementation of the RIoT benchmark is deployed entirely on the Raspberry Pi. The CDF denoted "device-edge" shows the end-to-end latency when the first stage of the 5-stage pipeline comprising the benchmark is hosted on the Raspberry Pi, and the other 4 stages are hosted on the edge-cloud VM. Finally, the CDF marked "device-edge-cloud" shows the deployment of the first stage on the Raspberry PI, stages 2, 3, and 4 on the edge-cloud VM, and stage 5 in the campus cloud. Note that the network connection between the edge cloud and the campus cloud traverses the microwave network discussed previously.

From the figure, comparing "device-only" to "device-edge" shows the effect of using the faster edge-cloud VM for 4 stages of the benchmark. However, because of the network variability between the Pi and the edge-cloud VM, approximately 20% of the execution runs were slower in the distributed configuration. The "device-edge-cloud" curve shows the effect of traversing the slower and lossier microwave link between stages 4 and 5. Note that the "device-edge" and "device-edge-cloud" CDFs have similar shapes, but the latter is shifted right by approximately 45 milliseconds. Note that we cannot include comparative results for the original RIoT implementation on the device alone because it will not execute on the ARM processors in 32-bit mode.

B. *Programming* LAMINAR *versus MLR Events and Logs*

To highlight the programmer productivity features of LAMINAR, we compare square two-dimensional matrix multiply implementations at the finest granularity possible, using native MLR directly (i.e. coded "by hand") and LAMINAR. The comparison is for illustrative purposes only, depicting simple, i.e., "naïve" implementations that have not been optimized. They are both intended to show the "baseline" programming model for each approach, i.e., the implementation that is maximally concurrent and includes no optimizations.

At the finest level of granularity (i.e., with maximal possible concurrency), one MLR handler must be "fired" for every multiplication of a pair of matrix elements and another handler is required to sum the multiplications to create the dot product stored in the matrix resulting from the matrix multiplication. All functions receive the unique sequence number of the append that fired it as input, but the sequence number order does not necessarily correspond to the order in which handler functions are executed. Further, because the MLR is lock-free and handlers are stateless, there is no way for handlers to block their execution pending log advance. Each function executes to completion, without blocking, once triggered.

For example, consider the depiction of a 2×2 implementation of matrix multiply shown in Figure 9. To compute $A \times B$ for 2×2 matrices A and B, the algorithm must compute the dot product for each element el in the result corresponding to el's row in A and el's column in B. The figure shows the computation of the $[0, 0]$ element of the result using row 0 of A and column 0 of B.

In the example, the values of the elements in A and B are stored in separate logs along with their row and column coordinates. The matrix multiply computation is started when a control program appends a record to a log indicating that the $C[0, 0]$ element is to be produced (shown as the box marked *control* in the figure). An MLR handler fires as a result of this append operation and performs two appends to a different log (marked *multiply* in the figure) to trigger the multiplications of $A[0, 0] \times B[0, 0]$ and $A[0, 1] \times B[1, 0]$ respectively in separate handler invocations. Each of the two elements that are appended indicates that the multiplication

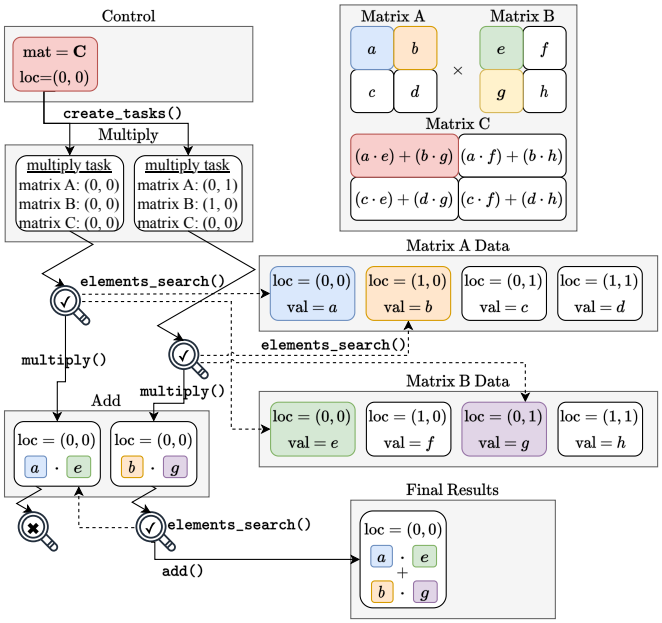

Fig. 9. Direct MLR implementation of 2×2 matrix multiplication, with logs and operations used to generate the first element of the result matrix.

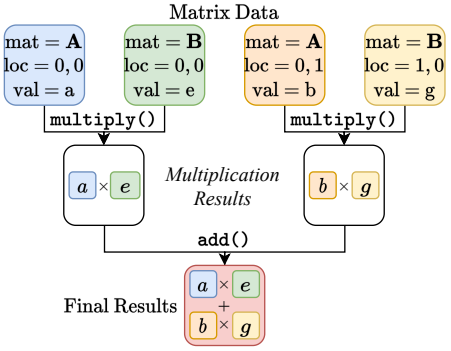

Fig. 10. LAMINAR implementation of 2×2 matrix multiplication, with logs and operations used to generate the first element of the result matrix.

is for the result $C[0,0]$ which will be used in a subsequent handler to gather the dot product terms.

Note that the dot-product handler must logically "wait" for the terms to be appended before performing an individual dot product and storing the result. Without the ability to block handlers, each dot product handler must scan the log (as described in Section III-A) to "join" the computations that are producing the operands necessary to complete the dot product ($C[0,0]$ in the figure).

The LAMINAR implementation of the same matrix multiply step is shown in Figure 10. Instead of log-appends and events, LAMINAR uses subscriptions and subscribers logs as buffers between computations. Recall that these data structures are created statically by the LAMINAR preamble when the application is deployed and accessed via a hashmap (as described in Sec. III) so that the runtime system can limit the number of elements that must be scanned to determine when a LAMINAR dataflow node is ready to fire.

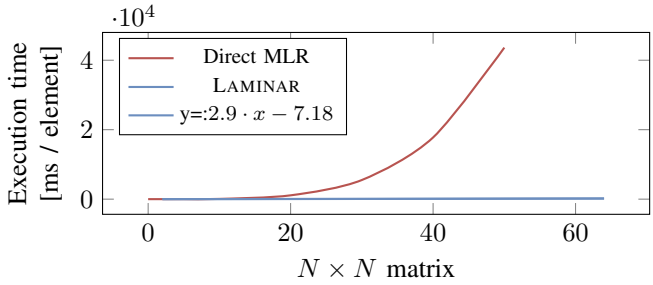

Fig. 11. LAMINAR matrix multiply performance. Execution time per element increases quadratically with matrix size using the direct MLR implementation due to log scans. LAMINAR execution time per element scales linearly.

This comparison illustrates how LAMINAR can improve programmer productivity over a highly concurrent event system such as the MLR. Specifically, the LAMINAR program abstracts away logs and handlers in favor of a high-level dataflow representation. LAMINAR does not require the user to understand MLR logs, tails, sequence numbers, or scans in the context of an application. An application is specified using dataflow, and the runtime system handles all log manipulation, joins, and synchronization.

1) Performance Comparison: Not only is matrix multiplication easier to write, but we also find that the code generated by LAMINAR is more efficient than the naïve MLR equivalent, illustrating the benefits of simplified programming and attesting to the efficiency of LAMINAR's type system implementation. Figure 11 compares the average time (in milliseconds) to compute a single element of the result matrix (on the y-axis) as a function of the dimensions of the matrix multiplicands (shown on the x-axis).

Simple matrix-multiply is an $O(N^3)$ algorithm where each of the dimensions of the multiplicands is N. The naïve MLR application must do N scans for each dot-product so the time to compute a single element of the result scales as $O(N^2)$. dimension N. In contrast, LAMINAR's use of a hashmap limits the scans to a small constant value.

As such, the average time to compute a single element (sample size 30) scales linearly with the dimension N. To clarify the linear relationship, a regression line is shown for LAMINAR because the units on the x-axis, which are $\times 10^4$, obfuscate the linearity, making the execution time for LAMINAR appear constant in the figure. The regression R^2 value is 0.96.

We emphasize that this scalability difference is purely due to the naïve but "straight-forward" direct MLR implementation. Since LAMINAR is using MLR as a target, it is certainly possible for a developer to implement the same (or better) scan-elimination optimizations than LAMINAR implements. However, hand-optimized MLR code is more complex to develop and more onerous to maintain. In the same way modern compilers generate machine code that is efficient enough to obviate hand-assembly programming for most applications, LAMINAR can generate simple yet efficient MLR code.

The examples shown in Figures 9 and 10 are intention-

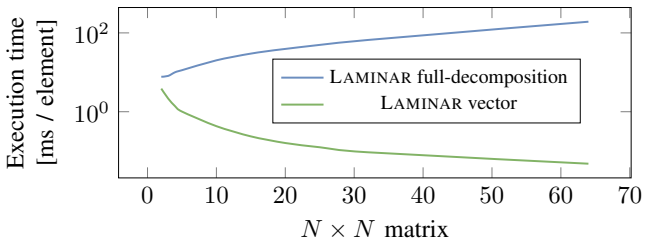

Fig. 12. LAMINAR per-element latency for fully-decomposed matrix multiplication versus a vector implementation.

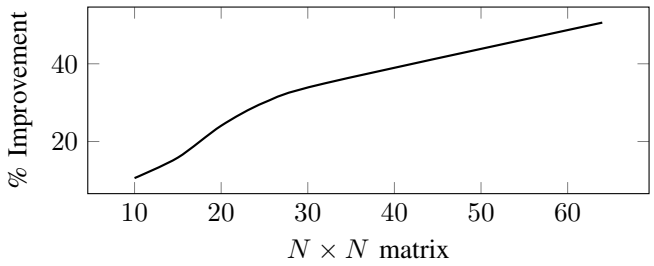

Fig. 13. Percentage improvement in per-element latency of LAMINAR partitioned vector implementation (with 8 partitions) over the fully-concurrent vector version (using an 8-CPU VM).

ally over-decomposed to compare the maximally concurrent versions of the matrix-multiply algorithm. From a practical perspective, they generate far too many fine-grained tasks for commodity multiprocessors to exploit effectively, accentuating potential LAMINAR overheads. LAMINAR's type system (cf. Sec. III-G) supports a vector data type that allows each handler to compute a full dot product as its output.

Figure 12 compares the performance of the maximally concurrent LAMINAR implementation (also shown in blue in Figure 11) to the performance of a LAMINAR implementation using the vector data type (shown in green in Figure 12) on a log scale. The per-element scaling exhibited by the maximally concurrent version is a linear slowdown, while the vector version shows a slight speedup as a function of matrix size. For example, the average time to compute a single element of a 64×64 result matrix for the fully-decomposed version scales up to 192.16 milliseconds compared to an average of 0.047 milliseconds for the vector version.

2) LAMINAR *Performance Optimization:* Finally, the vector version over-parallelizes the computation by assigning each of the N dot products to a concurrently executable node in the program. To show the effects of limiting parallelism to exploit only the number of CPUs available, we implemented a partitioned version of the LAMINAR vector code. This partitioned version assigns the computation of a contiguous "strip" of rows of the result matrix to each node of the LAMINAR DAG.

Figure 13 shows the percentage improvement in the time to compute a single element over the fully-concurrent vector version with 8 partitions (1 assigned to each virtual CPU). For example, the 64×64 per-element latency generated by the fully-concurrent version is 0.047 milliseconds and the latency for the same problem size generated by the partitioned version is 0.023 milliseconds. This factor-of-2 improvement is depicted as a 50% improvement in the figure. Note that for values of N less than 8, both implementations assign a single vector to each node in the LAMINAR DAG and thus are equivalent. As a result, we show the percentage improvement for $10 <= N <= 64$.

The improvement trajectory shown in Figure 13 shows the combined effect of eliminating unusable concurrency with increasing computation-to-overhead ratio for each LAMINAR node. While we have coded this example explicitly to illustrate the ability to control concurrency as a performance optimization, LAMINAR inherits and shares this capability with its high-performance functional-language antecedents. As such, it is possible to automate the aggregation of unusable concurrency as system-implemented optimizations (the subject of our ongoing work).

For reference, we implemented a version of the partitioned vector algorithm using POSIX threads and C. A direct performance comparison is difficult since the C version is neither (easily) distributed nor crash-consistent and based on locks. In the same VM (with eight virtual CPUs) with eight partitions, for a 64×64 matrix product, the in-memory C is an order of magnitude faster than LAMINAR. However, when the C version logs each element to a Linux file to achieve the same persistence characteristics as LAMINAR, it is an order of magnitude slower. Analyzing this disparity is beyond the scope of our current work since working with POSIX threads and C as universal development and implementation technologies has well-known programmer productivity and software maintenance drawbacks that LAMINAR is designed to address. It is noteworthy, however, that LAMINAR is "somewhere between" in-memory C and C with simple Linux-file-based persistence.

V. CONCLUSIONS

We have presented LAMINAR, a new programming system for creating robust IoT applications. LAMINAR layers a distributed, multiscale dataflow programming model over an event-driven, highly concurrent, lock-free, log-based, multiscale runtime. LAMINAR combines triggered computation and append-only, log-based persistent storage to implement its dataflow semantics efficiently. By doing so, we show that it is possible to use dataflow to hide many of the complexities of "lock-free" event-driven programming while leveraging the the portability and fault resiliency that such a system provides.

Our evaluation shows that LAMINAR simplifies the synchronization of events leading to less complex and more performant implementations. Further, the LAMINAR type system facilitates effective scaling, which is enhanced by partitioning optimizations enabled by its dataflow semantics. Designed for distributed IoT applications that experience frequent device and network failures, including its aggressive failure resiliency and recovery features, LAMINAR achieves performance comparable to alternative IoT technologies that treat failures as rare events and thus must incur greater recovery latencies. These features add to minimizing downtime, enhancing application reliability without developer intervention, thus making LAMINAR a robust dataflow application runtime.

REFERENCES

[1] M. Alam, J. Rufino, J. Ferreira, S. H. Ahmed, N. Shah, and Y. Chen, "Orchestration of microservices for iot using docker and edge computing," *IEEE Communications Magazine*, vol. 56, no. 9, pp. 118–123, 2018.

[2] K. Vandikas and V. Tsiatsis, "Microservices in iot clouds," in *2016 Cloudification of the Internet of Things (CIoT)*. IEEE, 2016, pp. 1–6.

[3] B. Butzin, F. Golatowski, and D. Timmermann, "Microservices approach for the internet of things," in *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2016, pp. 1–6.

[4] A. Kurniawan, *Learning AWS IoT: Effectively manage connected devices on the AWS cloud using services such as AWS Greengrass, AWS button, predictive analytics and machine learning*. Packt Publishing Ltd, 2018.

[5] B. Sharma and M. S. Obaidat, "Comparative analysis of iot based products, technology and integration of iot with cloud computing," *IET Networks*, vol. 9, no. 2, pp. 43–47, 2020.

[6] S. Klein, *IoT Solutions in Microsoft's Azure IoT Suite*. Springer, 2017.

[7] W.-T. Lin, C. Krintz, and R. Wolski, "Tracing Function Dependencies Across Clouds," in *IEEE Cloud*, 2018.

[8] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobbler, M. Wei, and J. D. Davis, "Corfu: A shared log design for flash clusters," in *USENIX NSDI*, 2012.

[9] F. Nawab, V. Arora, D. Agrawal, and A. El Abbadi, "Chariots: A scalable shared log for data management in multi-datacenter cloud environments." in *EDBT*, 2015, pp. 13–24.

[10] R. Wolski, C. Krintz, F. Bakir, G. George, and W.-T. Lin, "CSPOT: Portable, Multi-scale Functions-as-a-Service for IoT," in *ACM Symposium on Edge Computing*, 2019.

[11] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Transactions on computers*, vol. 100, no. 1, pp. 24–35, 1987.

[12] T. Meng, G. Jacobs, R. W. Brodersen, and D. G. Messerschmitt, "Asynchronous processor design for digital signal processing," in *ICASSP-88., International Conference on Acoustics, Speech, and Signal Processing*. IEEE, 1988, pp. 2013–2016.

[13] B. Marr, J. Karl, L. Lewins, K. Prager, and D. Thompson, "An asynchronous dataflow signal processing architecture to minimize energy per op," in *2013 IEEE 19th International Symposium on Asynchronous Circuits and Systems*. IEEE, 2013, pp. 50–57.

[14] S. S. Bhattacharyya, G. Brebner, J. W. Janneck, J. Eker, C. Von Platen, M. Mattavelli, and M. Raulet, "Opendf: a dataflow toolset for reconfigurable hardware and multicore systems," *ACM SIGARCH Computer Architecture News*, vol. 36, no. 5, pp. 29–35, 2009.

[15] E. C. Klikpo, J. Khatib, and A. Munier-Kordon, "Modeling multiperiodic simulink systems by synchronous dataflow graphs," in *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2016, pp. 1–10.

[16] E. Spertus and W. J. Dally, *Experiments with Dataflow on a General-Purpose Parallel Computer*. Massachusetts Institute of Technology, Artificial Intelligence Laboratory, 1991.

[17] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Symposium on Operating System Design and Implementation (OSDI)*, 2004.

[18] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *EuroSys*, 2007.

[19] D. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand, "CIEL: a universal execution engine for distributed data-flow computing," in *USENIX Symposium on Networked Systems Design and Implementation*, 2011.

[20] "GNURadio," 2023, https://wiki.gnuradio.org/index.php/Tutorials [Online; accessed 10-Jul-2023].

[21] J. Kodosky, "Labview," *Proceedings of the ACM on Programming Languages*, vol. 4, no. 78, 2020.

[22] M. Blackstock and R. Lea, "Toward a distributed data flow platform for the web of things (distributed node-red)," in *Proceedings of the 5th International Workshop on Web of Things*, 2014, pp. 34–39.

[23] H. Søndergaard and P. Sestoft, "Referential transparency, definiteness and unfoldability," *Acta Informatica*, vol. 27, pp. 505–517, 1990.

[24] D. Matei, "Referential transparency tutorial," 2023, https://www.baeldung.com/cs/referential-transparency.

[25] Z. Xie, H. Sun, and K. Saluja, "Survey of software fault tolerance techniques," *University of Wisconsin-Madison, Department of Electrical and Computer Engineering*, 2006.

[26] M. Kitakami, S. Kubota, and H. Ito, "Fault-tolerance of functional programs based on the parallel graph reduction," in *Proceedings 2001 Pacific Rim International Symposium on Dependable Computing*. IEEE, 2001, pp. 319–322.

[27] J. P. Morrison, J. J. Kennedy, and D. A. Power, "Fault tolerance in the webcom metacomputer," in *Proceedings International Conference on Parallel Processing Workshops*. IEEE, 2001, pp. 245–250.

[28] R. S. Nikhil, "The parallel programming language id and its compilation for parallel machines," *International Journal of High Speed Computing*, vol. 5, no. 02, pp. 171–223, 1993.

[29] J. Guttag, J. Horning, and J. Williams, "Fp with data abstraction and strong typing," in *Proceedings of the 1981 conference on Functional programming languages and computer architecture*, 1981, pp. 11–24.

[30] P. Hudak and J. H. Fasel, "A gentle introduction to haskell," *ACM Sigplan Notices*, vol. 27, no. 5, pp. 1–52, 1992.

[31] J. D. Ullman, *Elements of ML programming (ML97 ed.)*. Prentice-Hall, Inc., 1998.

[32] J. T. Feo, D. C. Cann, and R. R. Oldehoeft, "A report on the sisal language project," *Journal of Parallel and Distributed Computing*, vol. 10, no. 4, pp. 349–366, 1990.

[33] W. W. Wadge, E. A. Ashcroft *et al.*, *Lucid, the dataflow programming language*. Academic Press London, 1985, vol. 303.

[34] R. S. Nikhil and K. K. Pingali, "I-structures: Data structures for parallel computing," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 11, no. 4, pp. 598–632, 1989.

[35] D. A. Turner, "Miranda: A non-strict functional language with polymorphic types," in *Functional Programming Languages and Computer Architecture: Nancy, France, September 16–19, 1985*. Springer, 1985, pp. 1–16.

[36] "The purescript language," 2023, https://www.purescript.org.

[37] "JavaScript," "http://www.ecma.ch/ecma1/STAND/ECMA-262.HTM".

[38] "Keysight vee," 2023, https://www.keysight.com/us/en/home.html.

[39] M. R. Berthold, N. Cebron, F. Dill, T. R. Gabriel, T. Kötter, T. Meinl, P. Ohl, K. Thiel, and B. Wiswedel, "Knime-the konstanz information miner: version 2.0 and beyond," *AcM SIGKDD explorations Newsletter*, vol. 11, no. 1, pp. 26–31, 2009.

[40] "Hadoop MapReduce," "http://hadoop.apache.org/" Accessed Mar 2024.

[41] Y. Bu, B. Howe, M. Balazinska, and M. Ernst, "HaLoop: Efficient iterative data processing on large clusters," in *Proc. VLDB Endow.*, 2010.

[42] J. Ekanayake, S. Pallickara, and G. Fox, "Mapreduce for data intensive scientfic analysis," in *eScience*, 2008.

[43] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy, and C. Curino, "Apache Tez: A Unifying Framework for Modeling and Building Data Processing Applications," in *SIGMOD*, 2015.

[44] "Apache TEZ," 2024, https://tez.apache.org, Accessed Mar 1 2024.

[45] "Lwn thread cancel," 2023, https://lwn.net/Articles/683118/.

[46] S. H. Mortazavi, M. Salehe, C. S. Gomes, C. Phillips, and E. De Lara, "Cloudpath: A multi-tier cloud computing framework," in *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, 2017, pp. 1–13.

[47] T. Pfandzelter and D. Bermbach, "tinyfaas: A lightweight faas platform for edge environments," in *2020 IEEE International Conference on Fog Computing (ICFC)*. IEEE, 2020, pp. 17–24.

[48] "AWS Greengrass," 2019, https://aws.amazon.com/greengrass/ [Online; accessed 12-Sep-2019].

[49] "Iot edge — microsoft azure," 2023, https://azure.microsoft.com/en-us/services/iot-edge/.

[50] "Aws lambda – serverless compute - amazon web services," 2023, https://aws.amazon.com/lambda/.

[51] craigshoemaker, "Azure functions overview," 2023, https://docs.microsoft.com/en-us/azure/azure-functions/functions-overview.

[52] "Cloud functions," 2021, https://cloud.google.com/functions.

[53] M. Sadowski, L. Frantzell, and Sadowski, "Apache openwhisk–open source project," *Serverless Swift: Apache OpenWhisk for iOS developers*, pp. 37–57, 2020.

[54] "OpenFaaS," 2020, https://www.openfaas.com [Online; accessed 1-Sep-2020].

[55] J. Herath, T. Yuba, and N. Saito, "Dataflow computing," in *Parallel Algorithms and Architectures: International Workshop Suhl, GDR, May 25–30, 1987 Proceedings*. Springer, 1987, pp. 25–36.

[56] "GraphViz DOT Language," 2023, https://graphviz.org/docs/layouts/dot/, Accessed Dec 1 2023.
[57] S. Skedzielewski and J. Glauert, "If1 an intermediate form for applicative languages," *Lawrence Livermore National Laboratory Manual M-170, Livermore, CA*, 1985.
[58] S. Skedzielewski and M. Welcome, "Data flow graph optimization in if1," in *Functional Programming Languages and Computer Architecture*, 1985.
[59] "Intel NUC," http://www.intel.com/content/www/us/en/nuc/overview.html [Online; accessed 14-Feb-2015].
[60] A. Shukla, S. Chaturvedi, and Y. Simmhan, "Riotbench: An iot benchmark for distributed stream processing systems," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 21, p. e4257, 2017.
[61] "Apache Storm," 2023, https://storm.apache.org/.
[62] "Microsoft Azure," 2023, https://azure.microsoft.com/.
[63] O. Banos, R. Garcia, and A. Saez, "MHEALTH Dataset," UCI Machine Learning Repository, 2014, DOI: https://doi.org/10.24432/C5TW22.