Data Repair for Distributed, Event-based IoT Applications

Wei-Tsung Lin, Fatih Bakir, Chandra Krintz, Rich Wolski
{weitsung,bakir,ckrintz,rich}@cs.ucsb.edu
Computer Science Dept.
Univ. of California, Santa Barbara

Markus Mock
mock@haw-landshut.de
Dept. of Computer Science
Univ. of Applied Sciences
Landshut, Germany

ABSTRACT

Motivated by the growth of Internet of Things (IoT) technologies and the volumes and velocity of data that they can and will produce, we investigate automated data repair for event-driven, IoT applications. IoT devices are heterogeneous in their hardware architectures, software, size, cost, capacity, network capabilities, power requirements, etc. They must execute in a wide range of operating environments where failures and degradations of service due to hardware malfunction, software bugs, network partitions, etc. cannot be immediately remediated. Further, many of these failure modes cause corruption in the data that these devices produce and in the computations "downstream" that depend on this data.

To "repair" corrupted data from its origin through its computational dependencies in a distributed IoT setting, we explore *SANS-SOUCI* – a system for automatically tracking causal data dependencies and re-initiating dependent computations in event-driven IoT deployment frameworks. *SANS-SOUCI* presupposes an event-driven programming model based on cloud functions, which we extend for portable execution across IoT tiers (device, edge, cloud). We add fast, persistent, append-only storage and versioning for efficient data robustness and durability. *SANS-SOUCI* records events and their causal dependencies using a distributed event log and repairs applications dynamically, across tiers via replay. We evaluate *SANS-SOUCI* using a portable, open source, distributed IoT platform, example applications, and microbenchmarks. We find that *SANS-SOUCI* is able to perform repair for both software (function) and sensor produced data corruption with very low overhead.

CCS CONCEPTS

• **Applied computing → Event-driven architectures**; • **Computing methodologies →** *Distributed computing methodologies*; • **Computer systems organization →** *Distributed architectures.*

KEYWORDS

replay, IoT, serverless, cloud functions

ACM Reference Format:
Wei-Tsung Lin, Fatih Bakir, Chandra Krintz, Rich Wolski and Markus Mock. 2019. Data Repair for Distributed, Event-based IoT Applications. In *DEBS '19: The 13th ACM International Conference on Distributed and Event-based Systems (DEBS '19), June 24–28, 2019, Darmstadt, Germany.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3328905.3329511

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
DEBS '19, June 24–28, 2019, Darmstadt, Germany
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6794-3/19/06…$15.00
https://doi.org/10.1145/3328905.3329511

1 INTRODUCTION

The Internet of Things (IoT) is a rapidly emerging set of technologies that is fueling remarkable innovation in which ordinary physical objects in our environment are equipped with Internet connectivity, sensing, control, and computing capabilities. Because IoT devices are vastly heterogeneous and execute in a wide range of remote locations and operating conditions, they are subject to frequent hardware and software errors and failures, performance degradations, network partitions, etc., which in many cases cannot be easily or immediately remediated. Further, many of these failure modes corrupt the data that these devices produce as well as the "downstream" computations that depend on this data. Data corruption can mislead human and automated decision making, result in inaccurate predictions and inferences, and compromise the security, efficiency, and quality of service of data-driven IoT applications.

In this paper, we explore a new approach for repairing corrupted data in distributed, IoT settings. Replay is a technique used in single-host runtimes and distributed systems to fix errors in software and data structures, for trace-based simulation and prediction, to explore alternative application execution paths, and to perform post-mortem program analysis. We investigate a new approach to repair and replay, called *SANS-SOUCI*, which automatically tracks causal data dependencies and replays dependent computations across multi-tiered (device, edge, and cloud) IoT deployments. Unique to *SANS-SOUCI* is the integration of a function-as-a-service (FaaS)[1] programming model, append-only (versioned) data structures built into the runtime that persist application state, and distributed and causally-ordered event logging. To enable this, we extend an open source, distributed, FaaS runtime system called *CSPOT*[20], which executes over a wide range of devices – including microcontrollers, single board computers, edge computing systems, and public clouds – making portability and repair possible in a distributed IoT setting and across heterogeneous devices.

FaaS (also known as cloud functions) is the event-driven programming model and execution environment that underpins serverless and lambda computing architectures [6, 9, 13, 15]. Serverless systems based on FaaS were designed to simplify and automate the development, deployment, and elastic execution of cloud computing applications[2]. Developers construct applications from arbitrary functions, which they upload to cloud-hosted, serverless platforms.

[1]https://en.wikipedia.org/wiki/Function_as_a_service
[2]https://aws.amazon.com/lambda/, https://azure.microsoft.com/en-us/services/functions/, https://cloud.google.com/functions/docs/, https://openwhisk.apache.org, https://iron.io

These platforms link functions to cloud services and trigger their execution in response to service events (e.g. datastore updates, notifications, queue posts, API requests, etc.). These frameworks do not define specific storage abstractions for persisting state – instead they enable event-driven execution, automatic scaling, and access to cloud services (via programmatic interfaces exported via cloud software development kits (SDKs)).

CSPOT integrates an append-only storage abstraction into the FaaS platform for data durability and robustness, which functions use to persist application state. *SANS-SOUCI* builds upon and extends this versioned storage abstraction to simplify and facilitate data repair and replay. In particular, *SANS-SOUCI* couples this persistent storage with distributed event logging to track events, function invocations, and causal dependencies among events in multi-function serverless applications. Causal order is a partial order on the events in a distributed application that can be induced from observing internal events and messages between functions [8]. A number of event logging and profiling services are provided by serverless platforms [10, 11, 19][3].

SANS-SOUCI combines these features to maintain a history of transactions (up to a programmer-specified or device capacity maximum), which it uses to repair persistent data structures and re-initiate dependent computations. It uses this history to update corrupted data structures with corrected values at the historical point in time at which corruption occurred, and replays all causally dependent computation from that point forward. For example, if a network partition causes disruption in the production of sensor data, an application might instead forward interpolated, repeated, or error values (e.g. -1) to subscribers to mask the interruption. Subscribers downstream might use the values for analysis (e.g. prediction and classification). *SANS-SOUCI* can repair the historical data for the sensor when it comes back online, and automatically replay any dependent analysis functions.

For robustness in the presence of partial failures (common in distributed systems), *SANS-SOUCI* is also unique in that it maintains append-only semantics as it implements repair. That is, it does not "update-in-place" corrupted data, but rather it generates a new set of uncorrupted appends that occur (logically) after the corrupted dependencies. After a repair is complete, however, an application considering the most recent appends to a set of persistent data structures will "see" only the repaired data.

We integrate *SANS-SOUCI* into *CSPOT* and evaluate it using multi-tier IoT deployments, applications, and benchmarks. We find that *SANS-SOUCI* is able to perform repair for corruption produced by either software (function) or sensor data errors with very low overhead. The contributions that we make with this work include

- a new robust and distributed programming capability that allows data to be replaced at an arbitrary point in an application's state update history causing an automatic *repair* of all dependent state,
- a description of the implementation of this capability that is portable across all devices – from microcontrollers to public clouds – in an IoT setting, and
- an evaluation of this capability that is based on a combination of distributed IoT applications and microbenchmarks. Our results show that the *SANS-SOUCI* can achieve effective data repair while introducing overheads typically less than 10%.

[3] https://aws.amazon.com/cloudwatch/, https://aws.amazon.com/xray/

We next discuss the *SANS-SOUCI* design and use cases, and describe its abstract functionality with an example. We then overview the *CSPOT* system, and discuss how we extend the system to facilitate data repair and replay (Section 3). We present our results in Section 4, discuss related work (Section 5), and conclude (Section 6).

2 *SANS-SOUCI*

SANS-SOUCI is a new approach and system for repairing corrupted data in distributed, IoT applications. It automatically tracks causal data dependencies and replays dependent computations across multi-tiered (device, edge, and cloud) IoT deployments. To enable this, *SANS-SOUCI* combines functions-as-a-service (FaaS) for the event-driven system, append only data structures for persisting application state durably, and distributed, causal dependency tracking for efficient replay. Although each of these features are well understood distributed systems concepts, their combination reveals a rich set of design trade-offs that motivate this exploration.

In its typical form, FaaS is a programming, deployment, and event-driven execution model in which developers construct applications from arbitrary functions and upload them to cloud-hosted runtime systems. The model restricts function implementations to facilitate simplicity, scale, and fine-grained, isolated multitenency. In particular, functions are stateless, non-addressable, and must execute within resource constraints (e.g. time and memory limits). FaaS runtimes, referred to as *serverless* platforms, link functions to other cloud services and trigger their execution in response to service events (e.g. datastore updates, notifications, queue posts, API requests, etc.). Functions communicate and share data with other functions only through shared cloud services or via function call arguments. Examples of FaaS frameworks and serverless platforms include Amazon Web Services (AWS) Lambda Azure Functions, Google Cloud Functions, IBM OpenWhisk, and Iron.io.

SANS-SOUCI combines FaaS with versioned persistent storage and causal event tracking to simplify and expedite data repair and function replay. Because most serverless platforms and FaaS frameworks do not define specific storage abstractions for functions, functions persist state via remote cloud services (e.g. AWS Simple Storage Service (S3), AWS DynamoDB streams, OrpheusDB [7]) that are programmatically accessible via SDKs. Similarly, event tracking (with and without causal dependencies) is also possible via cloud services such as AWS Cloudwatch and XRay, GammaRay [11], and Dapper [19], among others.

Although it is possible for *SANS-SOUCI* to use these services for its implementation (e.g. by combining AWS Lambda, DynamoDB Streams, Cloudwatch, and XRay), in this paper, we overlay *SANS-SOUCI* on *CSPOT*, an open source FaaS framework, which we sketch in section 3. *CSPOT* is a *distributed* FaaS system that executes over heterogeneous devices and clouds, facilitating portable FaaS deployment and support for IoT applications and settings. *CSPOT* provides append-only persistent storage as *built-in* data structures that functions access directly to manipulate application state. It does so to ensure durability through eventually consistent replication. *CSPOT* also implements causal dependency tracking as part of its distributed logging system.

SANS-SOUCI uses the histories of persistent storage updates and their causal relationships to update corrupted or approximated data structures with corrected values at the historical point in an applications state update sequence at which they occurred. It then replays all dependent computation from that point forward. To ensure robustness to partial failures, *SANS-SOUCI* performs this update and replay (of causally dependent functions) using append-only semantics (versus update-in-place).

2.1 Use Cases

We envision three primary use cases for such data repair capabilities in IoT deployments. The first is to correct downstream historical results when a faulty sensor or data source (that has been issuing "bad" data) is repaired, and some data correction for the data that has been produced is available. One such real-world example is a misconfigured microclimate monitoring system, in which a subset of temperature sensors are configured to report Celsius rather than Fahrenheit as specified and required by the deployment. Such errors are commonly detected post deployment, after the data has been consumed by downstream analytics applications. *SANS-SOUCI* is able to correct the misconfigured readings and the downstream computations *in situ* – without stopping the deployment, gathering the data to a centralized location, cleaning it, and then redeploying it and the applications that use it.

The second use case is a debugging, exploration, and experimentation utility for deployed IoT applications. In a tiered IoT setting in which small sensor/actuator devices communicate with more powerful edge computing and storage devices, private clouds, or public clouds, computation and data are often distributed throughout the deployment. To experiment with or debug new computational methods (e.g. improved analytics) it is often inconvenient (or impractical) to create a parallel deployment. *SANS-SOUCI* uses replay to propagate data repair throughout the deployment making it possible to change specific computational components and then to observe the results and also to roll back such changes.

The third use case is to manage the arrival of late, but correct, data. As another real-world example, an IoT deployment might incorporate meteorological data from the CIMIS[4] network of weather stations. CIMIS publishes data on 5-minute intervals, but it does so retroactively, once every hour. The deployment itself expects data every 5 minutes. Existing applications generate an interpolation of the previous hour's CIMIS data every 5 minutes for the downstream components of the application to use immediately. *SANS-SOUCI* can "repair" the interpolations once the CIMIS data arrives at the top of the next hour.

Thus, the "corruption" that *SANS-SOUCI* is designed to repair covers several IoT use cases in which data gathered in the past can be replaced with better or more useful data in the future. Moreover, *SANS-SOUCI* can propagate the effects of those replacements throughout a distributed deployment.

2.2 *SANS-SOUCI* Data Repair

To effect a repair, *SANS-SOUCI* depends on the following properties.

- It must have access to all the program state that is used as input by any function that is casually dependent on the target of the repair at the time the target was initially produced.

[4]https://data.cnra.ca.gov/dataset/cimis-weather-station-data

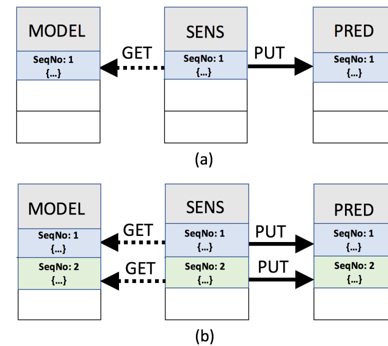

Figure 1: Data repair example. Application state is stored in persistent, append-only data structures (SENS, MODEL, and PRED) on 1+ hosts; each version has a sequence number (SeqNo). (a) shows the state after the first sensor element arrives; (b) shows the state after a second sensor element.

- It must be able to reproduce the order of execution (i.e. the causal execution order) of the functions that take this state as input.
- The functions must be side effect free so that their replay depends only on the program state visible to *SANS-SOUCI* .
- The repair itself cannot overwrite any of the previous program state that will be used as function input during replay before it is used.

An Illustrative Example

We overview the repair process via an example of the first use case above using a common prediction (or classification) streaming workflow. The example applies a trained model to each new datum that arrives from a sensor and produces a prediction. In the example, the programmer (or automatic service on her behalf) identifies an error in the model and produces a new version of it with the error corrected. She/it then initiates a repair to update the previous version and replay all previous predictions that depended upon the original "bad" value.

The example application has three data structures called SENS, MODEL, and PRED as depicted in Figure 1 (a). All data structure is persistent and append-only (i.e. each has multiple versions); the tail of each (i.e. the most recent version) holds the current state of the structure. Each version is identified via a sequence number (SeqNo). These data structures are persisted to disk and thus reside on a particular host. Data structures that make up an application can be on the same or different hosts. Functions access local data structures directly and remote data structures via messaging.

Periodically, the application receives sensor data, which it appends to SENS. The append triggers a function, fSENS, which reads from the tail of MODEL to retrieve the most recent prediction model. The function applies the model to the newly arrived data and produces a result (a prediction), which it writes to PRED. We refer to data structure reads as Gets and writes as Puts.

Dependent events are written to a local log as part of execution of the application. *SANS-SOUCI* records when data is appended, when the tail of the data structure is accessed, and when functions are executed (fired). It also records the causal dependencies with sequence numbers. In this case, there are two such dependencies

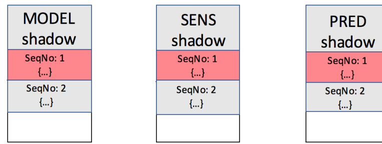

Figure 2: During repair *SANS-SOUCI* constructs shadows for dependent data structures to perform the repair. MODEL SeqNo 1 is repaired directly; its dependencies (marked in red) are repaired via replay.

(sequence numbers are specified in parentheses): (i) *SENS(1)* **Get** *MODEL(1)*, and (ii) *SENS(1)* **Put** *PRED(1)*.

When the programmer realizes that the MODEL has a bug, she appends the new model to the MODEL data structure. The next time a SENS value arrives, it will use the new MODEL as shown in (b) in the figure. The updates in the system that result from this event are shown in green. The new dependencies appended to the log are (i) *SENS(2)* **Get** *MODEL(2)*, and (ii) *SENS(2)* **Put** *PRED(2)*.

The programmer then also initiates a repair to fix PRED(1) via the *SANS-SOUCI* API, passing in the new value and sequence number of the version in need of repair (SeqNo 1 in this case). To effect the repair, *SANS-SOUCI* first requests the logs from all hosts and merges them into a total order. *SANS-SOUCI* uses this merged log to identify

- the chains of data dependencies (i.e. Puts and Gets) rooted at the target that must be updated, and
- a correct execution order of functions that will be "replayed" to generate these updates.

We refer to the dependency tree rooted at the repair as the "repair graph." For current commercial cloud functions implementations such as AWS Lambda, Azure Functions, and Google Cloud Functions (and also for *CSPOT*), each function has exactly one antecedent in a causal ordering. As a result, the current implementation of *SANS-SOUCI* generates repair graphs that are trees. We discuss relaxing this restriction in Section 3.2. *SANS-SOUCI* generates the repair graph via a scan of the merged log.

SANS-SOUCI creates a shadow data structure for each data structure impacted by the repair as depicted in Figure 2. Versions in red are those marked as dependent in the repair graph. It then copies elements from the original structure to the shadow from the first element up through the element prior to the start of the repair. It then appends the repaired values that the user has passed in. As part of this append, *SANS-SOUCI* retriggers any functions (fSENS in this case) that implement dependent **Get**s. Note that *SANS-SOUCI* only re-fires functions that *also* perform writes (i.e. Puts) on persistent data structures (i.e. perform state updates) because those without writes have no impact on global, shared state. This process continues as *SANS-SOUCI* traverses the repair graph.

The Puts and Gets in replayed functions use the shadow versions of the data structures and the sequence numbers passed in by *SANS-SOUCI*. To enable this, *SANS-SOUCI* replaces API calls that read data structures with those that read specific sequence numbers, and those that read and write data structures with those that that target a shadow data structure during replay. The application functions execute concurrently with the repair without interruption using the original data structures.

SANS-SOUCI copies any remaining values (those independent of the repair), after (or interleaved with) the repair, from the original to the shadow. It then synchronizes the shadow and original (pausing the application briefly) and performs a rename so that the application uses the shadow (i.e. the shadow becomes the original, for use by applications and the next repair, if any) and the original is garbage collected. We next describe the details of this process and provide the intuition behind our design decisions.

3 IMPLEMENTATION

To evaluate *SANS-SOUCI*, we have developed an implementation for *CSPOT* – a portable serverless application platform. In this section, we describe this first implementation of *SANS-SOUCI* in terms of *CSPOT*'s features and modifications to *CSPOT* that *SANS-SOUCI* requires. We believe that this exposition helps to illustrate both the advantages of FaaS and serverless systems in an IoT setting and some of the challenges that future systems (and future implementations of *SANS-SOUCI*) may encounter.

3.1 Background: *CSPOT* Implementation

CSPOT implements a set of lightweight abstractions specifically to support "Functions-as-a-Service" across a spectrum of device scales. Thus, it is possible to run the same *CSPOT* application code, without modification, on microcontrollers, edge devices and edge clouds, private clouds, and public clouds. It is also designed to support distributed FaaS applications in which application components may trigger functions on remote hosts that are running *CSPOT* as well. Finally, *CSPOT* is similar to other serverless application platforms in that the functions are triggered by events and run (when triggered) in isolated Linux "containers"[5] (on hosts where containers are available). On microcontrollers without support for memory isolation, all *CSPOT* functions must belong to the same trust domain.

To support distributed FaaS applications, *CSPOT* defines an intrinsic, low-level append-only data structure termed a WooF – **W**ide-area **O**bject **Of** **F**unctions. To aid with application robustness and data integrity, computations (which must be executed in *CSPOT* functions) can only be initiated as "handlers" that are triggered by an append operation to some WooF. WooFs are persistent with respect to system power down and handlers must be stateless. Thus, at any given moment in a *CSPOT* application execution, it is possible to restart an *CSPOT* application using the current application state stored in the application's WooFs.

There are three abstractions in *CSPOT*:

- **Namespaces** which separates the storage and functions in an application. A namespace contains the hostname and the path to the directory where data is stored and functions are executed.
- **Wide Area Objects of Functions (WooFs)** which are persistent, append-only memory objects for data persistence and
- **Handlers** which are stateless functions that are triggered when data elements are appended to WooF.

[5]https://en.wikipedia.org/wiki/LXC, https://linuxcontainers.org, https://www.docker.com

An *CSPOT* application can be hosted on multiple physical or virtual hosts. Each host contains one or more namespaces. All *CSPOT* objects are addressed by URI (Universal Resource Identifier). In the case where accesses are within a single namespace, the *CSPOT* runtime system implements them directly. When accesses cross namespaces, the access is implemented via network Remote Procedure Call. Thus, an *CSPOT* programmer can implement both locality (by co-locating WooFs and handlers in the same namespace) and distribution (by siting namespaces on separate machines). However the *CSPOT* APIs are consistent throughout the application.

Currently, *CSPOT* exposes four such APIs;

- **WooFCreate(woof_name, element_size, history_size)** which creates a WooF append-only storage within a namespace,
- **WooFPut(woof_name, handler_name, element)** which puts an element to a WooF. If an optional handler_name is specified, the host of WooF will trigger the function handler after the element is put.
- **WooFGet(woof_name, element, seq_no)** which gets the element corresponding to the sequence number.
- **WooFGetLatestSeqno(woof_name)** which returns the latest sequence number of the WooF.

The parameter *woof_name* in these APIs is a URI that encodes the network addressable location of a namespace and an object within that namespace. A WooFCreate() call creates a WooF with finite number of elements each having a fixed element_size. Either an external application client or *CSPOT* handlers use WooFPut() to append new elements to one or more WooFs and WooFGet() to get an element (that was previously appended) from a WooF. Each element in a WooF is assigned a sequence number when successfully appended. The sequence number is unique in a WooF. WooFGetLatestSeqno() returns the latest successfully appended element's sequence number from a WooF. While sequence numbers increase indefinitely, the space occupied by a WooF is managed as a circular buffer of finite size (specified when the WooF is created). Thus only a fixed history of data values (indexed by sequence numbers that do not reset) are available at any given moment.

As indicated previously, only a call to WooFPut() (either by an external client or a handler) which appends a data item to a WooF history, can trigger a subsequent handler computation. The current release of *CSPOT* supports both C-language and Python bindings. In this work, we use the C-language bindings to implement *SANS-SOUCI* and to evaluate its performance (cf Section 4).

The *CSPOT* runtime system also maintains an internal append-only event log in each namespace[6]. The namespace log is used directly to record state updates (WooF appends) and to trigger handlers. That is, when a call to WooFPut() creates a state update that specifies a handler to trigger, the caller appends the event to the end of the namespace log. Threads running within containers associated with the namespace synchronize on the tail of the log and race to "claim", and then execute, a newly added handler.

In the current release of *CSPOT*, only events describing handler triggers, and their eventual claims by container threads are logged.

[6]Note that the namespace log is logically a WooF with elements that describe events but because *CSPOT* uses the namespace log to implement handlers for WooFs the log is implemented separately to avoid a circular dependence.

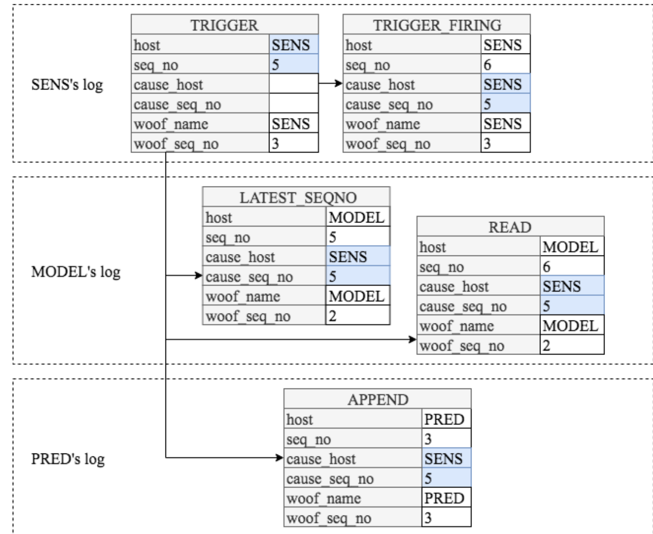

Figure 3: *CSPOT* namespace log events from the Sensor repair example.

SANS-SOUCI modifies this *CSPOT* implementation to include additional log event types for its dependencies (e.g. **Put** and **Get**).

All *CSPOT* namespace log events carry identifiers for the namespace, the object within the namespace, the "cause" namespace that originates the event, and the object within the cause namespace (implemented as hashes). Thus, within a namespace, the namespace log directly records causal order. This log-based runtime system organization is in contrast with commercial FaaS and serverless platforms where event logging relies on a statistical sampling of CPU program counter values (e.g. AWS X-ray[7]). In *CSPOT*, the causal ordering is directly recorded and not reconstructed after the fact via sampling. Because it is not generated from samples, *SANS-SOUCI* can use the *CSPOT* runtime log to implement correct application replay.

When a WooFPut() is called with handler_name specified, the namespace logs a TRIGGER event. If the WooFPut() call does not have handler_name specified, a *SANS-SOUCI* APPEND event is logged instead. When WooFGet() is called, a *SANS-SOUCI* READ event is logged and when a WooFGetLatestSeqno() is called, a *SANS-SOUCI* LATEST_SEQNO event is logged. The *CSPOT* API requires the programmer to implement a read of the current tail of a WooF as a call to WooFGetLatestSeqno() that returns a sequence number followed by a call to WooFGet() specifying the element from the WooF to retrieve. In this way, it is possible to implement applications that do not require strong consistency. Finally, when a thread claims a TRIGGER event, it appends a TRIGGER_FIRING event (atomically). Thus, each thread within a namespace container can determine which handlers have yet to be claimed.

On systems supporting virtual memory, the *CSPOT* namespace log and all WooFs are implemented using memory-mapped files. Thus, the system is capable of very low latency function dispatch compared to commercial counterparts.

[7]https://aws.amazon.com/xray/

Listing 1: Log merging algorithm

```
merge_log(logs):
  pending = []
  events = []
  global_log = []
  for log in logs:
    for event in log:
      append(events, event)

  while !empty(events) and !empty(pending):
    for event in events:
      if cause_event(event) in global_log:
        append(global_log, event)
      else:
        append(pending, event)
      remove(events, event)
    for event in pending:
      if cause_event(event) in global_log:
        append(global_log, event)
        remove(pending, event)

cause_event(event):
    return event->cause_host,
      event->cause_seqno
```

3.2 *SANS-SOUCI* Implementation for *CSPOT*

An *CSPOT* log is only local to its namespace. *SANS-SOUCI* implements a system for gathering and merging the runtime logs from all namespaces used by an application, preserving the causal dependencies globally. Listing 1 shows the log merging algorithm. The algorithm is $O(n \times log n)$ in the total number of events; it keeps event lists in search trees to facilitate causal dependency lookup.

SANS-SOUCI's global log, once generated, contains a correct total order of all application events that have occurred and the storage locations (in WooFs) that are associated with the triggering of those events. The size of the *CSPOT* logs, which is a tunable parameter, determines the length of this global history. Consider the application shown in Figure 1, for example, and assume that *MODEL*, *SENS*, and *PRED* are implemented as three separate *CSPOT* WooFs hosted in three different namespaces. When a new element *SENS(2)* is put into WooF *SENS*, *CSPOT* logs a TRIGGER event and triggers the handler function (fSENS) to calculate the prediction. The handler first calls WooFGetLatestSeqno() to get the latest sequence number of WooF *MODEL*, and then calls WooFGet() to get the latest model parameters with that sequence number. Finally, the handler uses these model parameters to calculate the prediction and puts *PRED(2)* to the *PRED* WooF without triggering a handler. This process generates five events, as shown in Figure 3.

Note that *SANS-SOUCI* only uses a global log to build a repair graph when applications are distributed (e.g. when the application comprises WooFs from more than one namespace). Otherwise, it uses the local namespace log for the namespace containing the application state. In either case it uses the *CSPOT* TRIGGER and APPEND events to identify *Put* dependencies, it adds READ events to identify *Get* dependencies, and LATEST_SEQNO events to capture accesses to WooF tails. We refer to this latter event as a *Sync* dependency.

Using either the global log (or the local log in case of a single namespace), *SANS-SOUCI* creates *shadow* WooFs for all WooFs that contain data that is causally dependent on the data being repaired. *SANS-SOUCI* copies all the previous values from these original WooFs (up to the length of the preserved history) that occur before the target of a repair to the shadow. The repaired value is then inserted with the correct sequence number by replaying the event handler used in the original put with the shadow as the target.

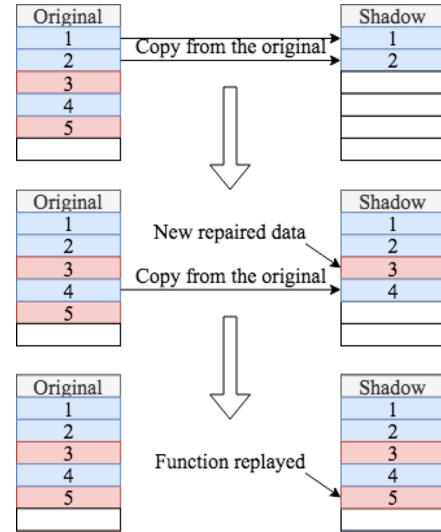

Figure 4: An example of a shadow WooF

Figure 4 shows a simple example of shadow construction. In the example, *Original(3)* is the target of repair and *Original(5)* is a downstream put caused by *Original(3)*. To repair the history, *SANS-SOUCI* first creates a shadow with the same capacity of the original WooF. All the elements from the earliest sequence number in the original WooF history to the last element before *Original(3)* are copied to the shadow, and then *SANS-SOUCI* waits for the arrival of the new value of *Original(3)*. After the new element to replace *Original(3)* is appended, *SANS-SOUCI* copies the intervening element *Original(4)* up to the next downstream element *Original(5)* from the original WooF. Once the intervening element is copied, the value of *Original(5)* which is produced by the function consuming *Original(3)* as input is appended at the correct place in the shadow history. Finally, after all elements are repaired and the remaining elements are copied from the original WooF, the shadow replaces the original WooF (via a rename) and the repair is complete.

Space Optimization

SANS-SOUCI replays *Put* and *Get* dependencies directly when constructing a shadow. However, when a *Sync* dependency is identified, *SANS-SOUCI* creates a separate mapping of the event's seq_no and the correct sequence number in the history that was returned when the application used the "current" latest sequence number in its original execution.

This contextualization is necessary to effect a space-saving optimization. *SANS-SOUCI* constructs the shadow in one pass without making a complete copy of *all* program state. Instead, it shadows only the state that is causally dependent on the data being repaired. Thus, it builds the shadow only by appending data that is dependent on either data occurring previously in the shadow *or* state that is unshadowed (i.e. state that contains no appends that are causally dependent on the data being repaired).

In either case, a *Get* dependency is correctly satisfied because it is identified by WooF and sequence number (either in the shadow or in the uncopied state). The "latest" sequence number is likewise correct if it refers to the shadow. However when a put to the shadow depends on the latest sequence number in state that does not require repair (is unshadowed) this latest sequence number is current to the application and not with respect to the application's history. Thus, the *Sync* dependency represents a space-saving optimization opportunity because it allows *SANS-SOUCI* to avoid a complete copy of all previous program state into a shadow. However it requires extra "bookkeeping" to ensure the correct contextualization.

Total versus Causal Ordering During Replay

Note that the current *SANS-SOUCI* implementation generates a correct *causal* ordering in the repaired WooFs even though each namespace log records a specific *total* order of events in its namespace. It is possible, within a single namespace, to reproduce the total order that occurred, but we elected to forego this additional level of replay accuracy for two reasons.

First, it is only possible to make a total order guarantee within the context of a single namespace (i.e. as recorded by a single log). For applications spanning namespaces, no such guarantee is possible because *CSPOT* does not use a centralized log in a distributed deployment. However, it may be possible to make such a guarantee in a future implementation that uses a system such as Chariots [17] to implement external consistency.

Secondly, even within the context of a single log (e.g. a single namespace) preserving the total order would not permit handler replay directly. That is, the current implementation of *SANS-SOUCI* literally refires *CSPOT* handlers during replay without further synchronization. It is possible that during the original program's execution, events generated by a single handler may have interleaved with events generated by other unrelated (but concurrently executing) handlers in the namespace. The namespace log correctly captures this interleaving but to reproduce it, each WooFPut() call in a handler would need to be synchronized with unrelated events in the log. The replay algorithm would need to pause (logically) after every put in every replayed handler, and determine whether non-dependent state (to reproduce the total order exactly) should be copied into the shadow prior to the next put. Because this additional synchronization would only be warranted for applications without cross-namespace dependencies, we felt it to be unnecessary overhead in an implementation of *SANS-SOUCI* for *CSPOT* .

Nonetheless, it is clear from our experience with the initial implementation of *SANS-SOUCI* that it may be possible to make stronger ordering guarantees when we consider implementing it for other FaaS platforms and runtime systems. The utility of such guarantees is the subject of our ongoing and future work.

Additional Challenges for *SANS-SOUCI* and IoT

This initial implementation of *SANS-SOUCI* for *CSPOT* exposes some important challenges for replay in an IoT setting. First, as described previously, *CSPOT* is like commercial FaaS systems in that it implements a model in which each state update has exactly one cause (each update is attributable to a single handler). This design feature makes causal ordering unambiguous and facilitates dependency tracking, but it is restrictive and can be cumbersome for the programmer. Often, one event is logically triggered either by any one or more antecedent events (the logical "or" of input dependencies) or by a complete set of input dependencies (the logical "and"). In an *CSPOT* application, the hapless programmer must encode these relationships explicitly in the handler logic rather than expressing them as a series of "guards" on handler triggers. However, more generally, this prototype of *SANS-SOUCI* does not need to address "joins" in the repair graph because *CSPOT* does not encode them (by design) in the application execution flow.

The careful reader will also notice that the current implementation logically may require a pause in application execution. First, if the application continues to execute while repair is taking place, it may trigger functions that create new dependencies on repaired data that is not captured in the current repair graph. If the implementation simply copies the state updates generated by these functions from the original to the shadow, the shadow will contain unrepaired dependencies. Thus, the implementation must lock the namespace and make a final "check" to determine if new state, dependent on repaired state, has been added by the application since the repair graph was generated. If it has, the repair can either complete while the namespace is locked, or the repair can be abandoned and retried thereby including the newly arrived dependencies in the repair graph. Also, if *any* of the original WooFs "wrap around" the circular storage abstraction, thereby overwriting needed history, the *SANS-SOUCI* repair will fail. There is also a moment at which the repaired shadow WooF must replace the original atomically. Lastly, while *SANS-SOUCI* uses append-only data structures itself, if the system fails during a repair these structures must be parsed so that the repair can continue from the point where it left off.

The prototype implementation does not address some of these synchronization issues. It does synchronize the replacement of the original WooFs with their shadows by locking the entire namespace during the rename. However it abandons the repair, unlocks, and retries if it detects dependencies that were not part of the repair graph at the start of the repair. Also it does not pause the application when it detects a wrap-around in the WooF or log storage space nor does it check to determine whether the state necessary to effect a repair is available before repair begins. Instead, it recognizes whether the needed sequence numbers are available "on the fly" and if it encounters needed-but-missing history, it correctly fails the repair without causing the application to fail. Thus, a replay could be indefinitely postponed in the prototype implementation.

Moreover, while it could continue a repair after mid-repair failure, we have not yet implemented the logic necessary to effect such a recovery. Thus, it removes all shadow state and restarts the entire repair process if it fails mid-repair. Finally, replay is eventually consistent across namespaces. It is possible to implement strong consistency but *SANS-SOUCI* will require a distributed commit protocol to replace WooFs with their shadows as a distributed transaction.

Note, also, that the *SANS-SOUCI* prototype generates each shadow using append-only updates in the same way that the original WooFs were constructed. As an alternative, it is possible to first copy all the unaffected elements from the original WooFs to the appropriate place in each shadow and then to "fill in the holes" using replay. We chose to preserve append-only semantics in the first prototype

of *SANS-SOUCI* for durability reasons, but we plan to explore the alternative as a possible optimization.

We do not, as yet, have enough experience with *SANS-SOUCI* "in the wild" to be able to judge the effects of these implementation properties or profitable optimizations. For example, as a debugging aid (e.g. to experiment with handler replacement using historical data) preserving the total order of event replay (at the expense of resiliency) may be warranted whereas in a data integrity setting the specific order of unrepaired data in WooF may not matter. The current *SANS-SOUCI* implementation for *CSPOT* will serve as a vehicle for investigating these questions in our future work.

4 EVALUATION

We evaluate *SANS-SOUCI* using two serverless applications originally developed for *CSPOT*. We use these applications to show *SANS-SOUCI*'s capability to repair the application history and to compare the performance with and without *SANS-SOUCI* integrated with *CSPOT*. These applications are designed to run on edge and cloud resources that relatively well-provisioned memory, disk, and operating system capabilities. In particular, they require numerical libraries that are not available or are too large to be hosted across a spectrum of device scales.

Thus, we also implemented a set of micro-benchmarks that are more portable than the IoT applications to further investigate the performance overhead introduced to each *CSPOT* API call by *SANS-SOUCI* across device scales. We first overview our experimental methodology along with the applications and micro-benchmarks, and present our empirical results.

Our results focus on the overheads introduced by *SANS-SOUCI* with respect to application functions rather than end-to-end application performance to avoid an overly optimistic assessment. For example, the Temperature application (described below) executes on a 5-minute duty cycle when deployed for production use. In one deployment, it has been in continuous operation for almost 18 months. During that period, there have been several outages where *SANS-SOUCI* could have effected a repair. Had it been available, the total time required to repair these outages would have been approximately 50 seconds over the entire 18 month time period. The long-lived nature of the IoT applications implemented using *CSPOT*, characterized by possibly intensive computations executed on relatively long duty cycles could obscure the true "costs" associated with *SANS-SOUCI*. Thus, we study the overheads on an application component basis rather than as a fraction of end-to-end execution performance.

4.1 Experimental Methodology

We use a cloud environment, a Raspberry Pi device (representing an edge computing device), and an ESP8266 microcontroller to evaluate the *SANS-SOUCI* implementation. To evaluate the applications, *CSPOT* is installed on a campus-level private cloud (approximately 1500 cores) managed using Eucalyptus 4.2.2[8]. We use a m3.2xlarge instance type having 4 CPUs (each 2.8 GHz) and 4GB of memory and Eucalyptus is configured to use KVM and Virtio for VM hosting. The instances are located in the same availability zone which interconnects physical hosts using switched 10Gb Ethernet. Each

[8]http://www.eucalyptus.cloud

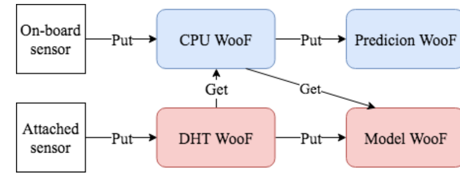

Figure 5: The Temperature prediction application structure

VM instance runs CentOS 7.6 and Docker 18.09 as the container engine. The portable micro-benchmarks run on the same cloud environment and a Raspberry Pi 3 Model B+ device. The device has Raspbian 9 installed and has an ARM Cortex-A53 1.4GHz CPU and 1GB SRAM. To host serverless platform, we modified *CSPOT* version 1.0 to include necessary logging to implement *SANS-SOUCI*. We also run the benchmarks on the ESP8266 microcontroller which has an 80 MHz RISC CPU with 80 KB of memory and 4 MB of flash storage. The microcontroller runs *CSPOT* as a native operating system. The source code for *CSPOT*, *SANS-SOUCI*, the sample applications, and the micro-benchmarks are open sourced [9].

The first sample application implements a "virtual" meteorological temperature prediction sensor using the on-board CPU thermometer. To avoid the need for an additional external thermometer (thereby freeing an I/O port on simple IoT devices) the application monitors internal CPU temperature and regresses it against temperature readings (employing a number of data conditioning techniques to improve the regression) taken from a weather station or remote thermometer (shared among all IoT devices in a deployment). In this paper, we use a Raspberry Pi as the IoT controller to aid in instrumentation and debugging and an externally connected DHT [10] humidity and temperature sensor as "ground truth." [11] The data conditioning and regression are numerically and memory intensive computations. Thus, the typical application deployment sends CPU temperature measurements either to an edge cloud (e.g. a small x86 cluster running private cloud software) sited in an out building, a private cloud, or a public cloud.

Figure 5 shows the structure of the application. To evaluate *SANS-SOUCI*, we use two VM instances to host the application in the private cloud. One instance stores the CPU temperature readings and generates and stores the "virtual" sensor values (i.e. uses the regression coefficients to produce a "predicted" outdoor temperature using CPU temperature as the explanatory variable). The other instance stores the DHT sensor readings and computes and stores the regression coefficients when ever a new "ground truth" reading is available. The application uses the *CSPOT* WooF storage abstractions and event handler mechanisms described previously in Section 3.2.

When a new CPU temperature reading is put to the CPU WooF, it triggers a prediction handler. The handler gets the latest regression coefficients (erroring out if there are none yet), generates a predicted outdoor temperature, puts the result to the prediction WooF. Asynchronously, when the DHT sensor reports a temperature reading, a thread running on the sensor puts the reading to the DHT

[9]https://github.ucsb.edu/rich/cspot
[10]https://www.adafruit.com/product/393
[11]This sensor synthesis application is used in the field, but when deployed in a non-experimental setting, the sensor controllers are microcontrollers and not small Linux platforms.

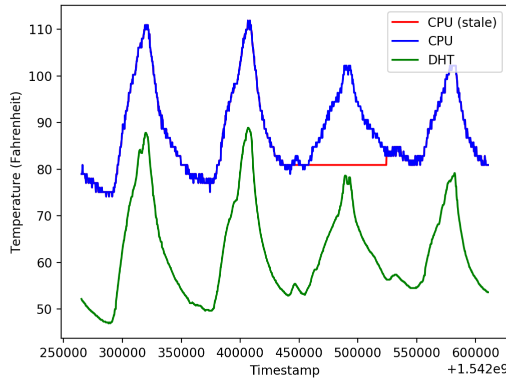

Figure 6: The CPU temperature and DHT sensor readings

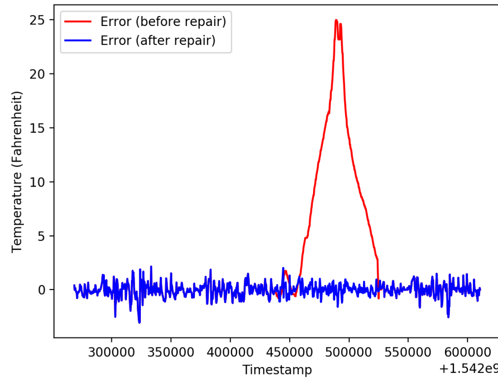

Figure 7: The prediction error before and after repair

WooF, thereby triggering the regression handler. The regression handler uses the latest CPU temperature and DHT sensor readings to generate new regression coefficients (based on the latest data) and puts it to the model WooF.

We placed a Raspberry Pi in an outdoor environment and collected four days worth of data to run the application. It sends its CPU temperature to a private cloud triggering a *CSPOT* handler there. The collected data consists of 1152 CPU temperature readings and 1152 DHT sensor readings (one reading every 5 minutes over four consecutive days). Each run of the application generates 79,316 log events in total. To demonstrate *SANS-SOUCI*'s ability to repair application history, we simulate data blackout (a frequent occurrence in the real deployments where the microcontroller uses Xbee [12] radios to communicate) by manually replacing one day of CPU temperature readings with stale data, as shown in Figure 6. After feeding the application with the data with the blackout period, we then use the correct CPU readings that were removed earlier to repair the application history. Figure 7 shows the prediction errors before and after the repair. Because of the stale CPU reading, the error before repair spikes to a maximum of 25 degree Fahrenheit during the data blackout. However, after repaired, the application manages to generate predictions with an error within 2 degrees Fahrenheit. While this test is contrived so that we could run it in a controlled environment, it reflects the types of outages that the application experiences in its various non-experimental deployments. Indeed, drop out caused by "late" data delivery in this application served as one of the motivations for this work.

[12]https://en.wikipedia.org/wiki/XBee

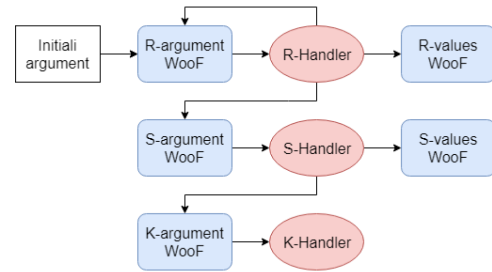

Figure 8: The Runs Test application structure

The second sample application is a *CSPOT* programming example that implements the Wald-Wolfowitz Runs Test[13] for pseudo-random number generators. Its true function is as a *CSPOT* exemplar, illustrating a strategy for translating multi-threaded programs to the event-driven abstractions implemented by *CSPOT*, but it also correctly implements the Runs Test.

Figure 8 shows the structure of the application. The application consists of three handlers: a handler to generate a stream of random numbers (denoted R-handler in the figure), a handler to generate the Wald-Wolfowitz Runs Test statistic (denoted S-handler), and a handler to generate Kolmogorov-Smirnov[14] test statistic comparing a sample of Runs Test statistics to a sample from a Normal distribution having the same sample mean and variance (denoted KS test). This application uses only a single instance in our experiments although the WooFs and handlers can be distributed. Further, the pseudo-random number generator we test in this application is the Mersenne Twister [15] which is known to have good randomness properties. Thus, "ground truth" is a KS statistic that is less than a KS critical value comparing the distribution of Runs Test statistics (over a sample of runs) to a sample from a Normal distribution (having the same mean and standard deviation) for significance level $\alpha = 0.05$. That is, with a Mersenne Twister pseudo-random number generator, we'd expect a KS test comparing a sample of Runs test statistics to a Normal to fail to show a difference at $\alpha = 0.05$ if the test is working correctly.

The application is initialized with the sample size and number of samples to use. When initiated, it triggers the R-handler to generate a new random number which it puts to a sample WooF. It also triggers another R-handler, passing an iteration count, by putting the modified argument structure to the generator WooF.

The R-handler triggers a put to the S-handler when it has accumulated enough data in the sample WooF. The S-handler gets the values from the sample WooF and computes a Runs Test statistic which it puts to the Runs-test WooF. After all iterations (each one producing a Runs Test statistic from a full sample) are finished, the S-handler triggers the K-handler to get the values from the Runs-test WooF, generate an empirical sample from a Normal distribution having the same sample mean and variance as that computed from

[13]https://en.wikipedia.org/wiki/Wald-Wolfowitz_runs_test
[14]https://en.wikipedia.org/wiki/Kolmogorov-Smirnov_test
[15]https://en.wikipedia.org/wiki/Mersenne_Twister

Table 1: Kolmogorov-Smirnov test result

	KS stat	Critical value
Before repair	0.31	0.192065
After repair	0.15	0.192065

Table 2: Applications average elapsed time in seconds with and without *SANS-SOUCI* over 10 separate experiments. Standard deviations are shown in parentheses.

	Temp. App.	Runs Test App.
Cloud w/o extra events	115.98s (1.63s)	2.84s (0.12s)
Cloud w/ extra events	118.56s (0.97s)	2.94s (0.09s)
Cloud w/ *SANS-SOUCI*	118.97s (1.46s)	3.12s (0.15s)
Total replay overhead (time)	2.99s	0.28s
Total replay overhead (%)	2%	10%

the Runs-test WooF values, and to generate a KS test statistic comparing the sample from the Normal to the Runs-test WooF values.

For the purposes of evaluation, we use this application exemplar in two ways. As with the "virtual" temperature sensor application, we show that *SANS-SOUCI* is able to repair application history by replaying dependent events with new data. We also show how *SANS-SOUCI* can also be used as a development or debugging aid by replaying application execution after replacing a handler with a different version. To demonstrate this ability, we intentionally "broke" the random number generating function in the R-handler by having it replace every fourth value with zero in the stream of values it produces. Then, we fixed the R-handler, and use the same arguments to replay the application again. By comparing the output before and after the repair, we can see the different KS-test results of the numbers generated by two versions of random number generator.

Table 1 shows the KS-test before and after the repair. Before repair (i.e. with the broken generator), the KS statistics correctly shows that the sample of Runs test statistics differs from a Normal at significance level $\alpha = 0.05$. However, after the repair, the KS stat becomes 0.15, which is less than the critical value, i.e., the "fix" repaired the application.

4.2 Replay Overhead

To evaluate replay overhead, we inserted timers at the entry and exit of all handlers that are triggered to time the application. Each application was executed 10 times on the campus private cloud with correct data input and working handler. After each run, the sum of all handler execution time was recorded.

Table 2 shows the average execution time with *SANS-SOUCI* and without *SANS-SOUCI*. The original *CSPOT* does not log PUT events that do not require a handler, GET, and LATEST_SEQNO events (these are needed by *SANS-SOUCI* for dependency tracking but not by *CSPOT* to implement handler activation). We separate the overhead introduced by *SANS-SOUCI* into the overhead associated with the necessary additional logging and the overheads associated with *SANS-SOUCI* processing during replay.

For the temperature prediction application, each run takes 115.98 seconds on average with the unmodified version of *CSPOT*. After adding the logging of handlerless PUT, GET, and LATEST_SEQNO events, the average time increases to 118.56 seconds. With *SANS-SOUCI* fully implemented, each run takes 118.97 seconds on average, that is 2.57% of execution overhead. This overhead mainly comes from the additional logging required by *SANS-SOUCI*. If compared to the *CSPOT* version which logs these events (but does not implement other *SANS-SOUCI* functionality), the overhead is merely 0.35%. For the random number generator application, each run takes 2.84 seconds in average without extra logging and 2.94 seconds with extra logging. With *SANS-SOUCI* fully implemented, each run takes 3.12 seconds on average, which translates to 3.52% of logging overhead and 6.12% of *SANS-SOUCI* execution overhead.

For the Runs Test application, the overheads are lower in absolute terms but higher as a percentage (approximately 10% on average). Its computational intensity is significantly less than for the Temperature application meaning the overheads associated with the runtime environment are a larger fraction of overall execution time.

Table 3: Average execution time in milliseconds over 10 experiments for each task in repair request. Standard deviations are shown in parentheses.

Task	Execution time
Dependency discovery	2632ms (76ms)
Merging global log	1426ms (61ms)

4.3 Log Processing Overhead

In this section, we describe the overhead associated with generating the repair graph necessary to enable replay. To generate the graph, *SANS-SOUCI* must "discover" the causal dependencies associated with the "root" of each repair. If the application uses only a single namespace, these dependencies are recorded in the log associated with the namespace. However, when the application spans namespaces, *SANS-SOUCI* first gathers the logs for all the namespaces and merges them into a global log to create a total order of application events that preserves causal order. It then uses the global log to identify the causal dependencies associated with each "root."

To understand the performance of dependency discovery and log merging, we run the Temperature application on the campus cloud described in the previous section and time the repair request 10 times. Table 3 shows the recorded time for each task.

Recall that with *SANS-SOUCI* enabled the Temperature application generates 79,316 logged events (which *CSPOT* stores in approximately in 55 megabytes) each run covering four days of measurement history. Parsing this event log to discover and build the repair graph requires approximately 2.6 seconds on average. If the application uses two separate name spaces (one containing CPU measurements and the predicted CPU values and the other containing all other WooFs) then average time to merge the logs from these namespaces is 1.4 seconds. These results are both application and deployment specific. That is, the complexity of the repair graph and also the distribution of WooFs among namespaces will affect both the discovery and merge times. However as an example, they indicate that the overhead associated with a repair is low.

Specifically, from Table 2, a *SANS-SOUCI* replay adds approximately 3 seconds to a 115 second execution time to the Temperature application when using unmodified *CSPOT* . Each repair will then impose an additional 2.6 seconds to parse the log and, if spanning

Table 4: Micro-benchmarks

Name	Description
Put	Put 1,000 elements to the WooF
Get	Randomly get 1,000 elements from the WooF
GetLatestSeqno	Get the latest sequence number 1,000 times
Replication	Replicate a 4k data object from edge device to cloud

namespaces, a further 1.4 seconds to gather and sequence the global log. Even when a repair is effected, the overall additional overhead introduced by *SANS-SOUCI* is under 10% for this application.

4.4 Micro-benchmarks

To better understand the overhead imposed by *SANS-SOUCI* on each *CSPOT* API call along with the *SANS-SOUCI* performance during the repair, we implemented a set of micro-benchmarks as listed in Table 4. We ran these micro-benchmarks 100 times (each consisting of a batch of 1000 invocations) and recorded the average with three versions of *CSPOT*: the original version without extra dependency events logged, with the additional events needed by *SANS-SOUCI* logged, and with the *SANS-SOUCI* fully integrated. We also ran the micro-benchmarks, repaired the entire benchmark data history using the same input, and timed the operations during the full repair. In each micro-benchmark, we insert timers at entry and exit of the each application function. We present the average execution times and standard deviations in microseconds (us) in Table 5 for each micro-benchmark. Note that no additional PUT events are required by *SANS-SOUCI* for the Put benchmark making the timings with and without these events the same.

On the cloud, adding *SANS-SOUCI* events introduces roughly 6us, on average. On the Raspberry Pi, adding extra logging adds roughly 15us, on average, while on the microcontroller the addition overhead due to logging is 16us, on average. *SANS-SOUCI* doesn't seem to introduce any overhead to WooFPut and WooFGet, mainly because *SANS-SOUCI* only needs to check whether the WooF is in repair mode (i.e. a boolean test). If so, the put and get request will be redirected to the shadow WooF. If the WooF is not being repaired, there's no additional performance overhead introduced. However, if the WooF is being repaired, *CSPOT* needs to open the shadow WooF and redirect the request to it, hence the overhead. For each WooFPut request, the overhead during repair is 42us on the cloud, 176us on Raspberry Pi, but we did not measure any noticeable overhead on the microcontroller. For each WooFGet request, the overhead is 16us on the cloud, and 6us on the microcontroller, but we did not observe overhead on Raspberry Pi.

For WooFGetLatestSeqno request, since it needs to record the mapping between the caller WooF's sequence number and the callee WooF's latest sequence number, even if the callee WooF is not in repair mode, there is a slight overhead introduced. During repair, WooFGetLatestSeqno also needs to find the latest sequence number in the mapping corresponding to the caller WooF's sequence number, introducing more overhead. In the cloud environment, to implement *SANS-SOUCI*, each WooFGetLatestSeqno request requires an additional 10us. During repair, the overhead doubles to 20us. On the microcontroller, we have not yet implemented the sequence number mapping for the *Sync* dependency, so the overhead for WooFGetLatestSeqno is left out. That is, the current *SANS-SOUCI* implementation for the microcontroller stops the application during repair, making the *Sync* dependency superfluous. Again, we did not observe any overhead on Raspberry Pi and are still investigating the reason why *SANS-SOUCI* does not seem to introduce overhead to WooFGet and WooFGetLatestSeqno on this platform.

Finally, we also evaluate the end-to-end overhead from edge device to a private cloud. In the "Replication" benchmark, we installed *CSPOT* on a Raspberry Pi located in a research laboratory located on the same university campus hosting the private cloud. The network interface attached to the Raspberry Pi is a 1 Gb/sec Ethernet and all traffic between this edge device and an instance in the cloud running *CSPOT* traversed the shared campus network. Both edge and cloud systems use NTP [16] to synchronize their internal clocks using a campus NTP server.

The benchmark first puts an object with 4 kilobyte payload to the Raspberry Pi, triggering a handler which reads the local clock to generate a timestamp that it embeds in the 4K payload. It then forwards the object to the private cloud instance. Upon its arrival, a handler is triggered on the cloud instance which takes another timestamp. We then record the difference of the timestamps as the end-to-end latency to replicate a 4K data object from the edge device to the private cloud.

We ran the benchmark, requested a repair, and then ran it again to evaluate the overhead *SANS-SOUCI* introduces to a simple cross-network replication. We repeated the process 100 times and show the average in Table 6. It takes 39.32 milliseconds to replicate a data object from edge to cloud, on average. To replay the replication and repair the WooF requires 40.27 milliseconds. That is, for a simple replication task which is not computationally intensive, *SANS-SOUCI* introduces additional 0.95 milliseconds, which equates to 2.4% of overhead.

5 RELATED WORK

Our work builds upon and extends a large body of related work on causal dependency tracking and record/replay. Causal dependency tracking is useful for a wide variety of applications including debugging, provenance tracking, auditing, speculation, and accounting, among others [10, 11, 14, 18, 19]. [11] provides such support for serverless applications through cloud services and across cloud regions and public cloud deployments. [14] combines causal tracing with dynamic instrumentation for user-guided, low-overhead application monitoring. We combine it with append-only, persistent data structures, and the FaaS programming model, to enable fast data repair and replay in distributed, heterogeneous settings.

Some record/replay systems leverage causal relationships to facilitate distributed debugging and exploration [1–4], and deterministic replay and simulation [5, 12, 16]. The authors in [12] checkpoint applications and intercept system/API calls to facilitate simulated and deterministically reproduced runs of a program. Determinism is captured using a logical clock inserted into messages. The authors of [16] investigate retroactive programming – support for reprogramming application histories. They combine the use of FaaS and causal event capture but change the FaaS programming model

[16] http://www.ntp.org

Table 5: Average Micro-benchmarks performance per 1,000 requests. The number shown represents the time for each *CSPOT* call. The units are microseconds and the standard deviations are shown in parentheses.

	Put	Get	GetLatestSeqno
Cloud w/o extra events	143.42us (2.57us)	136.50us (2.56us)	17.53us (1.72us)
Cloud w/ extra exents	-	142.94us (3.51us)	23.32us (1.66us)
Cloud w/ *SANS-SOUCI*	143.72us (3.55us)	142.76us (3.45us)	29.73us (2.34us)
Cloud during repair	185.40us (2.51us)	157.89us (1.10us)	38.42us (7.72us)
Rpi w/o extra events	504.62us (3.92us)	507.28us (4.49us)	103.18us (2.14us)
Rpi w/ extra events	-	521.20us (8.84us)	118.49us (2.71us)
Rpi w/ *SANS-SOUCI*	506.24us (10.44us)	523.14us (5.61us)	118.51us (2.61us)
Rpi during repair	681.91us (11.81us)	519.75us (4.61us)	116.19us (2.03us)
ucontroller w/o extra events	26.07us (0.17us)	7.64us (0.01us)	-
ucontroller w/ extra events	-	23.45us (0.15us)	-
ucontroller w/ *SANS-SOUCI*	26.45us (0.17us)	23.62us (0.13us)	-
ucontroller during repair	26.75us (0.19us)	29.9us (0.15us)	-

Table 6: Average elapsed time in milliseconds (over 100 runs) to replicate a 4k data object from an edge device to a private cloud. Standard deviations are shown in parentheses.

	Replication time
Normal run	39.32ms (3.16ms)
Replay	40.27ms (3.46ms)

by integrating Command Query Responsibility Segregation at the function level. Function types are partitioned into those that update state, view state, perform retrospection, and perform retroaction. *SANS-SOUCI* in contrast, focuses only on distributed data structure repair (and dependency replay) and so is significantly simpler, does not change the FaaS programming model, is fully distributed, and can be overlayed on any serverless system that supports causal event logging and state updates via versioned data service APIs.

6 CONCLUSION

We explore a new methodology for implementing data repair in IoT applications that use the "Functions as a Service" (FaaS) programming model and/or computing infrastructure. We describe the process of repair *in situ*, taking advantage of stateless function execution to "replay" an application from the point in its state update history where a faulty data item is to be replaced. To do so, the methodology relies on the availability of causal event tracking and versioned state updates in the underlying infrastructure.

We evaluate a prototype implementation of *SANS-SOUCI* using a portable FaaS infrastructure specifically designed for distributed IoT applications. It implements append-only state update semantics (which *SANS-SOUCI* treats as a history of application state updates) and causal event tracking (to facilitate debugging of a highly concurrent distributed IoT applications). Our results show that the *SANS-SOUCI* can achieve effective data repair while introducing overheads typically less than 10%.

ACKNOWLEDGEMENT

This work is funded in part by NSF (CNS-1703560, OAC-1541215, CCF-1539586, ACI-1541215), ONR NEEC (N00174-16-C-0020), and the AWS Cloud Credits for Research program.

REFERENCES

[1] P. Alvaro, S. Galwani, and P. Bailis. 2017. Research for Practice: Tracing and Debugging Distributed Systems; Programming by Examples. In *CACM*.
[2] I. Beschastnikh, Y. Brun, M. D. Ernst, A. Krishnamurthy, and T. E. Anderson. 2012. Mining Temporal Invariants from Partially Ordered Logs. *SIGOPS Oper. Syst. Rev.* 45, 3 (Jan. 2012).
[3] I. Beschastnikh, P. Wang, Y. Brun, and M. Ernst. 2016. Debugging distributed systems. In *CACM*.
[4] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica. 2007. Friday: Global Comprehension for Distributed Replay. In *NSDI*.
[5] Dennis Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. 2006. Replay Debugging for Distributed Applications. In *ATC*.
[6] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. 2016. Serverless Computation with OpenLambda. In *HotCloud*.
[7] Silu Huang, Liqi Xu, Jialin Liu, Aaron J. Elmore, and Aditya Parameswaran. 2017. OrpheusDB: Bolt-on Versioning for Relational Databases. *VLDB* (June 2017).
[8] L. Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (July 1978), 558–565.
[9] H. Lee, K. Satyam, and G. Fox. 2018. Evaluation of Production Serverless Computing Environments. In *IEEE Cloud*.
[10] W-T. Lin, C. Krintz, and R. Wolski. 2018. Tracing Function Dependencies Across Clouds. In *IEEE Cloud*.
[11] W-T. Lin, C. Krintz, R. Wolski, and M. Zhang. 2018. Tracking Causal Order in AWS Lambda Applications. In *IEEE International Conference on Cloud Engineering*.
[12] Xuezheng Liu, Wei Lin, Aimin Pan, and Zheng Zhang. 2007. WiDS Checker: Combating Bugs in Distributed Systems. In *NSDI*.
[13] T. Lynn, P. Rosati, A. Lejeune, and V. Emeakaroha. 2017. A Preliminary Review of Enterprise Serverless Cloud Computing (Function-as-a-Service) Platforms. In *IEEE International Conference on Cloud Computing*.
[14] J. Mace, R. Roelke, and R. Fonseca. 2018. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. *ACM Trans. Comput. Syst.* 35, 4 (Dec. 2018).
[15] G. McGrath and P. R. Brenner. 2017. Serverless Computing: Design, Implementation, and Performance. In *International Conference on Distributed Computing Systems Workshops*.
[16] Dominik Meissner, Benjamin Erb, Frank Kargl, and Matthias Tichy. 2018. Retro-Lambda: An Event-sourced Platform for Serverless Applications with Retroactive Computing Support. In *Intl. Conf. on Distributed and Event-based Systems*.
[17] Faisal Nawab, Vaibhav Arora, Divyakant Agrawal, and Amr El Abbadi. 2015. Chariots: A Scalable Shared Log for Data Management in Multi-Datacenter Cloud Environments.. In *EDBT*.
[18] Daniele Romano and Martin Pinzger. 2011. Using Vector Clocks to Monitor Dependencies Among Services at Runtime. In *QASBA*.
[19] B. Sigelman, L. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. 2010. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure.* Technical Report. Google, Inc.
[20] R. Wolski and C. Krintz. 2018. *CSPOT: A Serverless Platform of Things.* Technical Report 2018-01. UC Santa Barbara.