Tracking Causal Order in AWS Lambda Applications

Wei-Tsung Lin, Chandra Krintz, Rich Wolski, and Michael Zhang
Dept. of Computer Science
Univ. of California, Santa Barbara

Xiaogang Cai, Tongjun Li, and Weijin Xu
Huawei Technologies Co. Inc.

UCSB Technical Report 2017-03, Sept. 20, 2017

***Abstract*—Serverless computing is a new cloud programming and deployment paradigm that is receiving wide-spread uptake. Serverless offerings such as Amazon Web Services (AWS) Lambda, Google Functions, and Azure Functions automatically execute simple functions uploaded by developers, in response to cloud-based event triggers. The serverless abstraction greatly simplifies integration of concurrency and parallelism into cloud applications, and enables deployment of scalable distributed systems and services at very low cost.**

Although a significant first step, the serverless abstraction requires tools that software engineers can use to reason about, debug, and optimize their increasingly complex, asynchronous applications. Toward this end, we investigate the design and implementation of *GammaRay*, a cloud service that extracts causal dependencies across functions and through cloud services, without programmer intervention. We implement *GammaRay* for AWS Lambda and evaluate the overheads that it introduces for serverless micro-benchmarks and applications written in Python.

I. Introduction

Serverless computing [1], [2] (also known as cloud functions or functions-as-a-service (FaaS) [3], [4]), is an emerging paradigm for cloud software development and deployment in which software engineers express arbitrary computations as simple functions that are automatically invoked by a cloud platform in response to cloud events (e.g. HTTP requests, performance or availability changes in the infrastructure, data storage and production, log activity, etc.). Serverless platforms automatically set up and tear down function execution environments on-demand (typically using Linux containers), precluding the need for developers explicitly to provision and manage servers and configure software stacks. Developers construct and upload functions and specify triggering events. Functions are typically written in high level languages including Python, Java, or Node.js, leverage cloud services for their implementation, and communicate via HTTP or similar protocols.

Serverless applications are characterized by large numbers of transient, short-lived, concurrent functions. Because the cloud (and not the developer) provisions the necessary resources, and such functions (by definition) can tolerate a high degree of multi-tenancy, application owners pay a very small fee (after any "free tier" usage) for CPU, memory, and cloud service use (e.g. $0.20 per 1M invocations per month, and $0.00001667 per memory * execution time). To facilitate scale at a low price point relative to virtual server rental, cloud providers restrict function size (i.e., memory, code size, disk) and execution duration (e.g. 5 minutes maximum).

Amazon Web Services (AWS) released the first commercially viable FaaS, called AWS Lambda, in 2014 [5], [6]. Since this time, the model has received wide-spread adoption because of its simplicity, low-cost, scalability, and fine-grained resource control versus traditional cloud services. Its popularity has spawned similar offerings in other public clouds (e.g. [7], [8]) as well as in open source and private cloud settings (e.g. [9], [10], [11], [12], [13], [14]). Today, serverless is used to implement a wide range of scalable, event-driven, distributed cloud applications, including web sites and cloud APIs, big data analytics, microservices, image and video processing, log analyses, data synchronization and backup, and real-time stream processing.

The serverless programming paradigm simplifies parallel and concurrent programming and thus is a significant step toward enabling efficiency and scale for the next-generation (post-Moore's-Law era) of advanced applications, such as those that interact with data and the physical world (e.g. the Internet of Things (IoT)) [15], [16], [17], [18]. However, the complexity of asynchronous programming that these new applications embody requires tools that developers can use to reason about, debug, and optimize their applications. Today, such tooling for FaaS applications is nascent with only simple logging services available. Logging forces developers to write complex secondary applications that download, aggregate, analyze, and provide

effective anomaly alerts or feedback. Such effort is error prone, takes focus away from innovation, and must be repeated for every application.

To address some of these needs for Lambda, Amazon has developed AWS X-Ray [19]. X-Ray links function activities together using unique identifiers per function invocation and presents performance and dependency data to developers as logs and service graph summaries for each application. Although a good first step, X-Ray is limited in that (i) it does not provide causal ordering of events, (ii) it does not trace *through* cloud services (i.e. to capture dependency A→B, for a function A that updates a DynamoDB table that triggers function B), (iii) it performs sampling (missing events), and (iv) its history is limited to 24 hours.

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. Causality is an important tool employed in concurrent and distributed systems that facilitates reasoning about, analyzing, and drawing inferences from a computation [20], [21], [22], [23]. In particular, causal order is required for function design (to enable mutual exclusion, consistency, deadlock detection), for distributed debugging, failure recovery, and inconsistency detection, for reasoning about progress (termination detection, collection of obsolete data and state), and for measuring and optimizing concurrency. This lack of support in AWS Lambda limits the degree to which developers can identify the root cause of errors, performance bottlenecks, cost anomalies, and optimization opportunities for Lambda applications.

To address these limitations, we present *GammaRay*, a cloud service for AWS Lambda applications that provides a holistic view of causal application behavior and performance end-to-end. *GammaRay* requires no developer intervention and works across AWS regions and AWS cloud services. *GammaRay* intercepts Lambda function entry points and calls to AWS services made by the application. It records these events synchronously using transactional database streams (to guarantee causal consistency) and processes them off line, in near real-time, to provide developers with service graphs and analysis data at both the function aggregate and instance level. As such, *GammaRay* precludes the need for developers to write their own CloudWatch and X-Ray log parsing and aggregation tools for each application, and provides causal ordering for concurrent, multi-function Lambda applications.

This paper investigates three implementation alternatives for *GammaRay*. Two of these alternatives are full X-Ray replacements that collect both performance data and causal relationships using static and dynamic instrumentation. The third is a hybrid approach that leverages X-Ray for performance monitoring (incurring some of its limitations) in exchange for lower runtime performance overhead. We investigate the overhead of each alternative using micro-benchmarks and multi-function serverless applications. We find that the hybrid approach performs the best and that its implementation introduces 17ms per API call, 419ms on function startup, and 5MB of memory, on average, over X-Ray.

II. Background

Serverless computing [1], [24], [4], [25], [26] is a cloud computing execution model in which a cloud service invokes functions that comprise an application, on behalf of an application owner and in response to cloud service events. Event triggers include HTTP requests, database or object store updates, invocation by other functions, performance or availability changes in the infrastructure, log and queue activity, and publish/subscribe notifications, among others. Because the service automates and abstracts away the details of function invocation, resource allocation and deallocation, and runtime triggers for these applications, the model is referred to as `serverless`, even though, behind the scenes, servers are still involved. In this section, we provide background on the AWS Lambda service which we build upon and extend in this paper.

A. AWS Lambda

AWS Lambda was made available to the public as an AWS service in 2014. The service provides support for Lambda functions written in Python, Java, C#, and Node.js, which access AWS cloud services via AWS software development kits (SDKs) for these languages. Each Lambda function has a single entry point (specified in its deployment configuration) and is deployed in a particular AWS region. A Lambda function can invoke other Lambda functions (including themselves) in the same region. They can also be invoked automatically (i.e. triggered) by updates made to AWS "event sources" including DynamoDB, Simple Storage Service (S3 object storage) Simple Notification Service (SNS), CloudWatch, Alexa, and Kinesis. Some services serving as event sources pass details about the triggering action (e.g. the key that was updated in DynamoDB, the bucket and file prefix that was deleted or modified in S3, the SNS topic to which a post occurred, etc.). Other event sources such as function invocation via the command line interface (CLI) and the API Gateway [27], that trigger functions asynchronously, include no trigger-identifying information in the callee.

AWS deploys Lambda functions via isolated Linux containers and may (or may not) reuse containers for repeat executions of the same, recently executed, function [28]. Functions can access parts of the container file system (e.g. deployment directory and /tmp), environment variables, and the network. Functions can integrate libraries and binary programs, but the resulting deployment package is size-constrained. AWS Lambda also limits on the number of concurrent function executions, disk usage, and execution duration of

functions (e.g. to 5 minutes), among other restrictions. The latest AWS Lambda limits can be found at http://docs.aws.amazon.com/lambda/latest/dg/limits.html.

Moreover, some event sources (e.g. DynamoDB, Kinesis, CloudWatch logs) can trigger multiple functions in the same region for the same event (i.e. events have "fan-out" dependencies). Alternatively, S3 and API Gateway trigger a single function in the same region per path or route, respectively. Finally, a SNS notification can trigger one or more Lambda functions in any region and is an ideal event source for cross-region interoperation within region-distributed Lambda applications.

B. Monitoring AWS Lambda Applications

There are two performance monitoring services available to AWS application developers: CloudWatch [29] and X-ray [19]. CloudWatch is a service that collects information about AWS service and resource use. It also includes the ability for applications to write their own performance records and an API for filtering, downloading, and reading CloudWatch logs. Accessing CloudWatch via the API however, is limited (e.g. 5 transactions per second per region) with commonly long delays (on the order of seconds) between event execution and the availability of its log record, for scaling and system stability purposes. The latest CloudWatch limits can be found at http://docs.aws.amazon.com/AmazonCloudWatch/latest/logs/cloudwatch_limits_cwl.html.

CloudWatch logging is available for AWS Lambda functions in all AWS regions, however log streams are local to a region and may or may not be distinct for concurrent invocations of the same function. Developers must write complex applications (potentially as Lambda applications themselves) to extract actionable insights about the performance and behavior of their Lambda applications, which is tedious and error prone given the use limits, region isolation, and eventual consistency of CloudWatch logging. Such efforts are infeasible and costly for even medium-scale AWS Lambda applications, which can consist of hundreds to thousands of function instances.

To address some of these limitations for highly dynamic Lambda applications, web applications, and microservices, AWS more recently introduced X-Ray. X-Ray automatically samples the entry and exit of function instances, called segments, using unique trace identifiers (`trace_id`). When a sample is taken, X-Ray records function duration and container startup overhead, and records and times SDK calls and HTTP accesses that a function makes, as *subsegments*. Users can define, annotate, and record their own subsegments. X-Ray data is sent to an X-Ray daemon running in the container with the function via UDP. The daemon buffers and sends monitoring data (when sampled) to the X-Ray logging service.

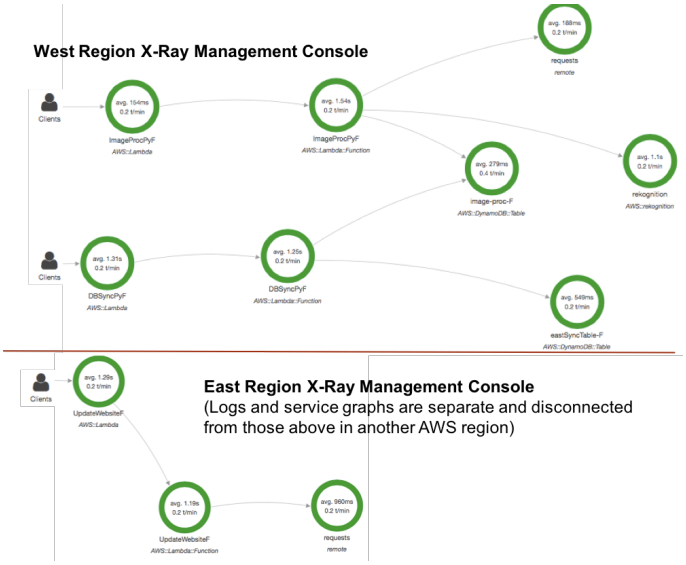

Fig. 1: X-Ray service graph for ImgProc [30].

The X-Ray logging service presents data to developers as logs and dependency trees, called service graphs. X-Ray links the segment and its subsegments (per `trace_id`) into an application's service graph as leaf nodes with meta-information, such as the table name of the DynamoDB table that the function updated and the region in which it is located. Service graphs visualize X-Ray log data for specified time durations (aggregating multiple invocations of the applications).

Figure 1 shows the service graph for a multi-function, concurrent Lambda application that integrates two popular Lambda use cases (image processing and geo-replication of data). The application, called ImgProc (ImageProcPyF), is triggered by a user uploading a photo to an S3 bucket (we configure updates to this bucket as an event source for the function). Via the AWS SDK, the function invokes the AWS Rekognition image processing service on the photo and writes the labels that it returns to a DynamoDB table (image-proc-F). The function then reports its findings to a dynamic web page (cf the requests object) and exits. They table write triggers a second function (DBSyncPyF), which concurrently synchronizes the table across regions (reading table image-proc-F in the west region and writing to table eastSyncTable-F in the east region). This second table write triggers a third function in the east region which updates a mirrored web page local to its region. ImgProc was originally developed by AWS engineers [30]; it is one of the applications with which we empirically evaluate our work.

When a function is triggered by an unknown source, the service graph represents this via a `Clients` icon. X-Ray divides the function into two parts (subsegments): its startup overhead (type AWS::Lambda) and its execution time (AWS::Lambda::Function). Multiple instances of functions are combined into aggregate service graphs by X-Ray. However, within the raw data of the logs from which the service graphs are

drawn, there is a segment with a unique `trace_id` for each service graph with unknown source. The segment consists of metadata and subsegments (for SDK calls, HTTP requests, and any user-defined operations). The metadata includes start and end times, `trace_id`, and details about the operation type and outcome (e.g. error status, if any). Segments and subsegments are linked via `Id`'s and `parent_id`'s for subsegments with the same `trace_id`. Thus it is possible to construct an ordering of events (subsegments) that originate from the same function (parent) but not across top-level segments.

C. Limitations

The figure above (Fig. 1) reveals multiple limitations of X-Ray. First, even though the ImageProcPyF triggers DBSynchPyF by updating a DynamoDB table, X-Ray log and service graph data does not capture these relationships. Similarly, across regions, functions are disconnected and part of independent X-Ray traces. Moreover, the only option for viewing service graph data is in aggregate; only log data contains per function instance data.

Other X-Ray limitations relate to record loss. X-Ray uses statistical sampling of performance information. Highly scalable applications, "rare" events that exercise code paths that are difficult to test can cause faults that are difficult to reproduce and diagnose. If the events are sufficiently rare, a statistical technique may miss them. In addition, X-Ray uses UDP messages to a separate process to offload logging overhead. Because UDP is an unreliable network transport mechanism, it may be that message are lost before their content can be logged. Given this implementation, it is not possible to use X-Ray alone to construct the causal order of events across Lambda applications.

III. *GammaRay*

Causality (or the causal precedence relation) is a well understood and important tool employed in concurrent and distributed systems that enables developers to reason about, analyze, and draw inferences their applications [22], [31], [32]. Such reasoning and analysis is difficult in these settings because the communication delay between distributed functions that communicate via message passing over a network, is unpredictable. As such, it is impossible for them to agree on the exact time and thus on the total order of events that the application experiences across functions [20]. However, it is possible to establish a partial order on events with causal precedence if we know the order of internal events – and – we synchronously record when messages are passed.

To facilitate causal order tracking for serverless applications in AWS that is *cloud-wide* – across all AWS services and regions – we have developed a cloud service for AWS called *GammaRay* that we depict in Figure 2. *GammaRay* extracts causal precedence for Lambda applications by monitoring both function-internal events (like X-Ray) and the message-passing performed by functions *through* AWS cloud services (which X-Ray does not). Moreover, *GammaRay* augments causal relations with both aggregate and instance-level performance data. To enable this, *GammaRay* automatically injects instrumentation into Lambda functions and into the SDK with which they invoke cloud services (event sources that trigger other functions) upon function deployment to AWS Lambda. *GammaRay* considers messages in this setting to be SDK calls between functions and services. *GammaRay* synchronously records sender and receiver information for each call. *GammaRay* consumes these records off-line to compute causal relations and performance statistics for each event and to construct a service graph that can be easily interrogated by developers and analysis tools.

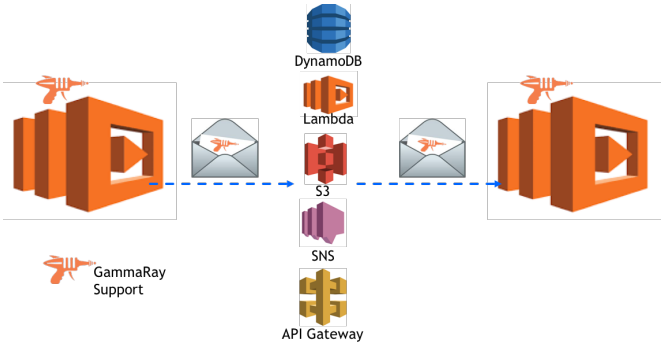

Fig. 2: *GammaRay* Overview. *GammaRay* automatically injects support that captures the entry/exit and SDK (cloud service access) of AWS Lambda applications. Each SDK invocation carries a trace identifier through the call. *GammaRay* extracts the identifier upon function entry for any triggered functions.

The *GammaRay* design consists of three components: a Lambda function deployment tool, *GammaRay* runtime support, and *GammaRay* event processing engine. The function deployment tool takes a code directory and a list libraries and builds a code package that it then uploads to Lambda using the developer's credentials. The tool filters out unused libraries to minimize package size. If *GammaRay* support is requested (a command line option), the tool injects *GammaRay* instrumentation by replacing the function entry point with the *GammaRay* entry point and by "wrapping" AWS SDK and HTTP operations.

When a function executes, the *GammaRay* runtime assumes control via this instrumentation, upon entry and exit of function, SDK, and HTTP calls. The runtime records information about each event synchronously in a database. The record contains information including a timestamp and a unique ID for the function instance. Upon entry, *GammaRay* stores the unique ID of the

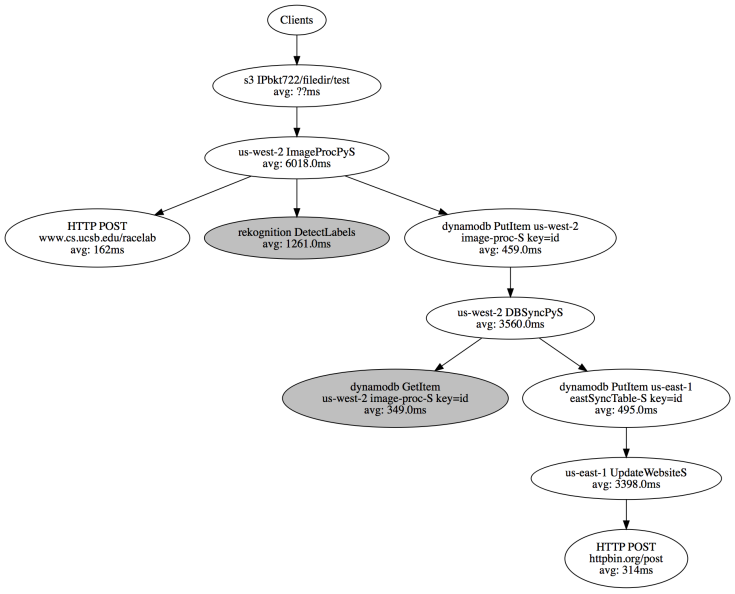

Fig. 3: *GammaRay* service graph for ImgProc [30]. *GammaRay* captures causal dependencies and performance through AWS services and across regions.

function instance, for inclusion in downstream records for the function. It also invokes the original function. When the original function returns, *GammaRay* records its duration its return/response, and any errors/exceptions that occur, as part of the exit record. For SDK calls, *GammaRay* records information about cloud service access such as table name and keys for DynamoDB updates, bucket name, prefix, and key for S3 updates, SNS topics, and HTTP URLs.

As mentioned previously, when a Lambda function invokes another (via the SDK or HTTP) there is no trigger-identifying information available in the callee. To overcome this limitation, *GammaRay* injects the caller's unique (request) ID into the payload of the invocation as a hidden argument. This data is later used by the event processing engine to map cloud service updates (event sources) to function invocations and produce causal relations across the application.

The *GammaRay* event processing engine runs in the background in response to a transactional database stream, to construct a service graph using the causal order and performance of events across an application [1]. Using the *GammaRay* API, this data can be queried and analyzed by downstream data analysis tools, e.g. those for anomaly detection and root cause analysis [33].

[1]Note that DynamoDB Stream semantics (employed by *GammaRay* to implement the transactional database stream) enables multiple agents to agree upon a single shared *total order* on events (when/if connected).

A. Implementations

We next investigate three alternative implementations of *GammaRay*: `G-Ray-D`, `G-Ray-S`, and `G-Ray-H`. In all three, *GammaRay* automatically inserts "hidden" arguments into function invocations as needed, and processes all function arguments upon function entry. Additionally, all configurations implement the *GammaRay* log via a shared DynamoDB table and stream. DynamoDB Streams record the sequence of record-level DynamoDB table modifications [34] and thus enable *GammaRay* to extract the causal relationships across events that it records (in the order they occur).

`G-Ray-D` injects the necessary *GammaRay* instrumentation dynamically using a library that "monkey-patches" [35] AWS Lambda SDK calls made by the function to invoke the *GammaRay* runtime before and after the call. It represents the most flexible, portable, and application-transparent implementation strategy. Alternatively, `G-Ray-S` implements the same functionality by adding the instrumentation code *statically* to the AWS Lambda Python SDK. It increases the size of an instrumented Lambda program (both in terms of memory and package size) but avoids dynamic runtime instrumentation overhead.

`G-Ray-D` and `G-Ray-S` are full replacements for AWS X-Ray. They improve upon X-Ray in two ways. First, they track causal order across AWS regions and across service invocations. Secondly, they track *all* events (rather than a statistical sample) so they can be better used for performance debugging activities such as diagnosis of faults due to rare events (X-Ray uses statistical sampling). Moreover, because they only depend on AWS's scalable database (DynamoDB), and this functionality is relatively common among public cloud providers, in theory, these implementations could be ported to other public clouds.

Alternatively, `G-Ray-H` is an AWS-specific implementation of *GammaRay* that makes maximal use of extant AWS services, including X-Ray and CloudWatch. It implements the same causal-ordering tracking as *G-Ray-D* and *G-Ray-S*, but because `G-Ray-H` relies on AWS for function timings, its performance data is sampled.

All implementations use the AWS SDK (boto [36]) and `G-Ray-H` and `G-Ray-D` rely on the Fleece library for X-Ray daemon support [37] for Python. The *GammaRay* deployment tool also uses the SDK to upload the compressed Lambda package to AWS Lambda, to set up the necessary policy and permissions for the function, and to configure any event sources that trigger function invocation. We have experimented with implementing *GammaRay* for Java as well. In this paper, however, we report on our experiences with Python exclusively.

B. GammaRay Event Processing Engine

The *GammaRay* event processing engine runs offline – in the background and so does not introduce

App	Description
empty	Micro: Returns immediately
DDB read	Micro: 100 random reads of DynamoDB table
DDB write	Micro: 100 random writes to DynamoDB table
S3 read	Micro: 100 random reads of random S3 object
S3 write	Micro: 100 creates of a new S3 object
SNS	Micro: 100 postings to SNS
Map-Reduce	A Big-Data-Benchmark [39], [40] app implemented in Lambda by AWS Engineers [41]
ImgProc	Image Processing app [30]. Images uploaded to S3 trigger a function which extracts labels using AWS Rekognition service, and reads and writes DynamoDB tables within and across regions (performing geo-replication), and triggering a cross-region function

TABLE I: Micro-benchmarks (demarked Micro) and Multi-Function Lambda Apps used to evaluate *GammaRay*. All are available from our project repository.

overhead on serverless applications. The engine processes the table data in append-order via the DynamoDB Stream. From this information, it constructs a service graph containing causal order dependencies for each application across AWS services and regions. It presents this data to users as graph aggregates (as X-Ray does) or for individual function instances (which X-Ray does not) and annotates the graph with performance data. The amount and type of data with which *GammaRay* annotates its graphs is configurable.

Figure 3 shows the service graph for the ImgProc application for one run of the `G-Ray-S` configuration. *GammaRay* leverages graphviz [38] for its service graph implementation. In this configuration, the engine displays SDK operation names and key names, and average performance across event instances. Because the S3 write is performed by a user (Clients) directly, the average time is not available (denoted ??ms in the figure). *GammaRay* displays non-event-source operations (e.g. DB reads) in gray and errors in red.

IV. Evaluation

To evaluate *GammaRay*, as well as to illuminate the source of the overhead it introduces, we employ both multi-function Lambda applications and micro-benchmarks. We first overview these applications and our empirical methodology and then present our empirical results. We used only the AWS Free Tier for implementation and evaluation of this study (i.e. no costs were incurred for function invocation).

A. Methodology

The applications and micro-benchmarks that we use in this study are listed in Table I. We present the baseline timings in milliseconds (ms) and memory used in megabytes (MB) for each in Figure 4. For the micro-benchmarks, the DB payload is 4 bytes; the S3 operations are on empty files. We execute both sets of Lambda applications multiple times and compute the

TTime (ms)	Clean	X-RayND	X-Ray
empty	6.825	10.304	12.888
DDB read	2,434.524	2,760.705	4,745.461
DDB write	2,392.259	2,926.633	4,754.816
S3 read	2,841.425	3,134.483	5,215.324
S3 write	5,354.460	6,073.092	8,727.316
SNS	4,217.200	4,327.122	6,432.616
Map-Reduce	124,582.122	122,156.327	114,006.962
ImgProc	3,417.780	3,047.135	3,067.145

Memory (MB)	Clean	X-RayND	X-Ray
empty	20.990	24.980	40.960
DDB read	24.980	40.960	44.410
DDB write	40.960	44.410	64.860
S3 read	44.410	64.860	63.560
S3 write	64.860	63.560	48.625
SNS	30.960	34.840	46.720
Map-Reduce	1,175.333	1,203.825	1,231.351
ImgProc	107.920	113.360	114.440

Fig. 4: Baseline performance data for the micro-benchmarks and Lambda apps. The top table shows the total time in microseconds; the bottom table shows the total memory consumed in MB, on average across runs.

average and standard deviation. We execute the micro-benchmarks 200 times and the Lambda applications 50 times unless otherwise noted.

The baseline configurations, which we use for comparison and which include no *GammaRay* functionality, are `Clean`, `X-RayND` and `X-Ray`. `Clean` captures the performance of the application with all tracing turned off for all functions. `X-RayND` shows the performance of the stock AWS X-Ray with tracing turned on, but the data capture mechanisms do not use a separate X-Ray "daemon". Without this daemon option, X-Ray logs function entry and exit calls in Python applications but not the Python SDK calls that the function makes. `X-Ray` is full AWS X-Ray support for Python applications using the X-Ray daemon implemented via the Fleece library [37].

The baseline measurements reveal interesting characteristics about AWS Lambda. First, the `empty` micro-benchmark results (in which the function simply returns) show no statistical difference in either the mean or the variance of their execution times, either with or without tracing. This result seems to indicate that the X-Ray logs are updated asynchronously (i.e. there is intermediate buffering for which users are not charged).

Full X-Ray introduces overhead for both the DDBread and DDBwrite benchmark. Each benchmark reads/writes DynamoDB 100 times. In the case of `X-RayND`, only the start and exit of the benchmark are logged. For `X-Ray`, each of the internal 100 SDK calls to DynamoDB are also logged. Since the mean execution time approximately doubles, we conclude that internal SDK logging for DynamoDB using the X-ray daemon

requires approximately 1/100 the time required for entry and exit logging. The same seems to hold for S3 reads but not for S3 writes (which take more time than reads. For long-running X-Ray does not wait but instead posts records that the operation is "in-progress" [42] potentially incurring more overhead for multiple log records.

For the multi-function Lambda applications, Map-Reduce and ImgProc, `X-Ray` executes in less total time than `Clean` (last two rows of top table in Figure 4). We ran a Student's t-test [43] on the datasets and find their means to be different. We do not have a good explanation as to why `X-Ray` is faster but believe that it is related to the AWS implementation and deployment of X-Ray. In our evaluation, we compare *GammaRay* to X-Ray for these applications.

B. Application Performance

We first empirically evaluate *GammaRay* for our long-running Lambda application: Map-Reduce. This application was written by AWS engineers and is based on one of the Big Data Benchmark programs [39]. This application implements the map-reduce protocol but relies only on AWS Lambda and S3 for its implementation, i.e. it does not use HDFS [44], Hadoop [45], Spark [46], or Amazon Elastic Map Reduce (EMR). We use the pavlo/text/1node/uservisits [40] dataset which is 24MB in size and contains IP addresses that have visited particular websites. The application invokes 29 mappers, which read their portion of the input from S3. Mappers count the number of access per IP prefix for a range of IPs and store the results in S3. A coordinator monitors this progress (via triggers from S3 writes) and invokes a single reducer function when all mappers complete. The reducer downloads the intermediate results and performs a reduction across them to produce the final per-IP count which its stores in S3 (which again triggers the coordinator one final time).

Recall from Figure 4, that the application without *GammaRay* or X-Ray completes in approximately 125 seconds (cf Map-Reduce row, `Clean` column of Figure 4) and 114 seconds with full X-Ray enabled (shown in the `X-Ray` column of Figure 4 in the Map-Reduce row). Figure 5 shows the percentage overhead versus full X-Ray introduced by *GammaRay* on the map-reduce application. On total time, `G-Ray-D` introduces 25.1%, `G-Ray-S` introduces 15.3%, and `G-Ray-H` introduces 11.9% overhead versus *X-Ray*. On memory use, `G-Ray-D` introduces 3.7%, `G-Ray-S` introduces 7.5%, and `G-Ray-H` introduces 4.4% overhead.

This overhead is primarily due to the instrumentation performed by each variant. On function entry, *GammaRay* parses and logs (in DynamoDB) function input data. `G-Ray-D` and `G-Ray-S` both also log exit events to DynamoDB (to record timings). `G-Ray-D` and `G-Ray-S` log before and after each SDK call to capture timings and causal dependencies; `G-Ray-H` records a

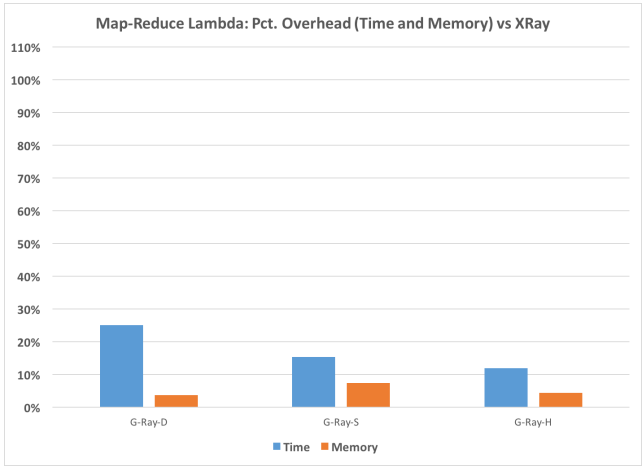

Fig. 5: Percent overhead versus X-Ray for *GammaRay* for the Map-Reduce Lambda application. For each *GammaRay* variant (G-Ray-D=dynamic, G-Ray-S=static, and G-Ray-H=hybrid), we present the percent overhead on total time across functions and on memory used across functions, on average for 50 runs of the application.

log entry before each SDK call that might trigger other Lambda functions, to track causal dependencies. Moreover, X-Ray tracing is turned off for `G-Ray-D` and `G-Ray-S` (because it is not needed) and turned on for `G-Ray-H` which uses X-Ray data to annotate the causal service graph with performance data (offline).

The overhead of *GammaRay* is low for this application because the time spent not executing event-source-triggering calls is large relative (the application executes for over 124 seconds) to the number of calls that *GammaRay* instruments. For this app, `G-Ray-S` and `G-Ray-D` generate over 840 *GammaRay* tracing records; `G-Ray-H` generates 125 records. As a result, much of the time is spent in mapper and reducer functions for data processing.

We next evaluate the overhead of *GammaRay* for the short running ImgProc application. ImgProc performs image processing and geo-replication of database tables; we describe this application fully in Section II. The application consists of three dependent functions (two in the east region and one in the west) that trigger each other via DynamoDB table updates (in both regions). Application execution is initiated by file being placed in an S3 bucket.

Figure 6 shows the percentage overhead of *GammaRay* versus X-Ray for the ImgProc application. As shown in the baseline data, one instance of the app completes in 3.1 seconds and uses 114MB of memory for `X-Ray`. Because a single instance of this application is very short running, *GammaRay* consumes a significantly larger overall percentage of total time than it did for Map-Reduce. For this application, `G-Ray-D` introduces 92.3%, `G-Ray-S` introduces 66.8%, and `G-Ray-H` introduces 42.9% execution overhead. In terms of mem-

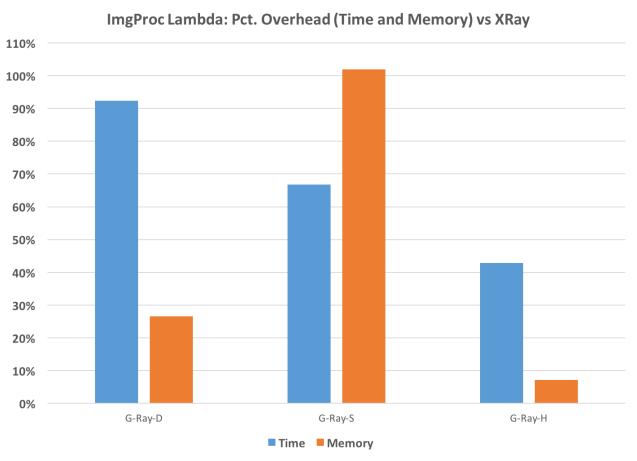

Fig. 6: Percent overhead versus X-Ray for *GammaRay* for the Image Processing (ImgProc) Lambda application. For each *GammaRay* variant (G-Ray-D=dynamic, G-Ray-S=static, and G-Ray-H=hybrid), we present the percent overhead on total time across functions and on memory used across functions, on average for 50 runs of the application.

Storage MB	Local Project Directory		/tmp Directory	
	compressed	uncompressed	compressed	uncompressed
Clean	0.0002	0.0001		
X-RayND	0.0002	0.0001		
X-Ray	0.7311	2.2188		
G-Ray-D	0.7325	2.2266		
G-Ray-S	0.0021	0.0117	3.4353	23.8320
G-Ray-H	0.7330	2.9609		

Fig. 7: Container disk space usage for GammaRay wrapper and library support

ory use, `G-Ray-D` introduces 26.5%, `G-Ray-S` introduces 101.1%, and `G-Ray-H` introduces 7.2% overhead.

The implementation of `G-Ray-S` adds more memory overhead than the other variants. We believe this is because of the additional code footprint that we require for the *GammaRay* library extensions (we measure and discuss disk space usage further below). Moreover, a single invocation of the ImgProc application comprises 18 events that consume most of the execution time. *GammaRay* writes database records for all 18 events including entry/exit for configurations `G-Ray-D` and `G-Ray-S`. Configuration `G-Ray-H` posts only 5 records at during execution (those sufficient to capture the causal ordering). `Clean`, `X-RayND`, and `X-Ray` post no records during execution – all performance data is recorded via unreliable communication and eventually consistent, non-order preserving logs, asynchronously.

From the results of these two Lambda applications, we conclude that the execution overhead associated with tracking causal ordering across regions and AWS service invocations is lowest for the *GammaRay*–X-Ray hybrid (configuration `G-Ray-H`). This configuration enables *GammaRay* to use less memory and record the minimal set of events (required to identify causal relations across events) synchronously, and all other events asynchronously via the X-Ray daemon. We next investigate the overhead that *GammaRay* introduces at a finer grain using micro-benchmarks (the first six programs in Table I).

C. Container Disk Space Usage

As discussed previously, the empty micro-benchmark returns immediately when invoked. We use this micro-benchmark to evaluate the storage overhead *GammaRay* imposes on container disk space given the minimal `Clean` code package of this benchmark. The table in Figure 7 reports disk space usage in megabytes (MB) for the Lambda function package (the function code and its libraries) that is downloaded and decompressed upon container instantiation when a function is invoked. Columns 2 and 3 shows the size in MB for the package compressed and uncompressed. On average *GammaRay* increases compressed package size by less than 1% for `G-Ray-H`.

The size of the package is limited by AWS to have a maximum of 50MB compressed and 250MB uncompressed. Large package sizes also slow down function deployment times (including version replacement and code update). To keep deployment times low, libraries in the package can be placed in the `tmp` file system in the container. To use this option, developers package this code separately and upload it to S3. Upon invocation the developer adds code to the start of the function that downloads, extracts, and links the code into the application. The *GammaRay* tool performs these operations automatically. AWS limits the maximum size of the `tmp` file system to 500MB.

We use this option for the `G-Ray-S` configuration. We do so because this configuration rewrites a small portion of the AWS SDK (botocore). AWS provides the SDK in the container for free. Because of the rewrite, *GammaRay* must include botocore in the deployment (to replace the default container version). By doing so, `G-Ray-S` has a very small project package and a large (compressed and uncompressed) `tmp` file system component as shown in the table using columns 3 and 4. We include the time required to download from S3 and uncompress the package in all `G-Ray-S` experiments. We find that if the function is executed repeatedly and AWS reuses the container, we can avoid this overhead. To do so, *GammaRay* first checks whether the downloaded package exists and if so, performs only library loading and linking.

We also believe that the additional `G-Ray-S` library code increases the overall memory footprint at runtime (cf `G-Ray-S` Memory in Figures 5 and 6). On average, however, *GammaRay* introduces a small overhead on container storage for its wrappers and additional libraries for both `G-Ray-D` and `G-Ray-H` versus X-Ray

TTime	Startup	SDK (Overhead per operation)					
Ohead (ms)	(wrapper)	DDB Read	DDB Write	S3 Read	S3 Write	SNS	Avg
X-Ray (over Clean)	6.063	47.326	47.419	52.024	87.144	64.197	59.622
G-Ray-H (over X-Ray)	418.850	1.458	29.474	2.700	19.347	33.846	17.365
G-Ray-H (Total over Clean)	424.913	48.783	76.894	54.724	106.492	98.043	76.987

Memory Ohead (MB)		SDK (Total per benchmark)					
X-Ray (Over clean)		17.285	16.910	16.495	14.150	14.150	15.798
G-Ray-H (over X-Ray)		1.985	5.010	8.270	7.855	4.360	5.496
G-Ray-H (Total over Clean)		19.270	21.920	24.765	22.005	18.510	21.294

Fig. 8: Micro-benchmark Result Summary. The top table shows the overhead on total time (TTime) in milliseconds (ms) and the bottom table shows overhead on memory in megabytes (MB) for the micro-benchmark programs. The first row of data is the overhead that X-Ray introduces for performance monitoring. The second row of data is the additional overhead on top of X-Ray that *GammaRay* introduces. In the top table, the overhead is broken down by startup time and per-SDK operation. At startup, the *GammaRay* wrapper introduces 125ms for obtaining a handle to the *GammaRay* database table from AWS and just under 300ms for processing function inputs and storing them in the table. On average, X-Ray adds 60ms per SDK operation and 16MB of memory overall. *GammaRay* adds another 17ms and 5MB of memory, respectively.

because it is able to leverage the same libraries as X-Ray for their implementation.

D. Micro-Benchmark Performance

We next breakdown the overhead of tracing on the remaining micro-benchmarks. We only consider `G-Ray-H` as it is the best performing *GammaRay* configuration. `G-Ray-H` keeps its overhead low by relying on X-Ray to collect performance statistics. Thus X-Ray must be turned on in this configuration (introducing some overhead itself). Moreover, since X-Ray only performs sampling and its logs are eventually consistent (with delays of seconds in many cases), the performance information on the *GammaRay* service graphs is also subject to these disadvantages. However, *GammaRay* guarantees causal order for service graph connectivity through AWS services.

The two DDB micro-benchmarks execute random 100 reads and 100 writes to different AWS DynamoDB tables, respectively. The two S3 micro-benchmarks execute random 100 reads and 100 writes to AWS S3 buckets, respectively. And the SNS micro-benchmark posts 100 notifications to AWS SNS. The performance data for the `Clean` and `X-Ray` configurations to which we compare is shown in the baseline data (Figure 4). The performance results for the micro-benchmarks is shown in Figure 8. The top table presents data for total time overhead in milliseconds (ms) and the bottom table shows memory overhead in megabytes (MB) for `X-Ray` and `G-Ray-H`.

For total time overhead (the top table), we break out that imposed on function startup from that imposed on SDK calls. The first row of data in the top table shows the number of milliseconds added to `Clean` by `X-Ray`. `X-Ray` adds 6ms at startup and 47-87ms on the different SDK operations evaluated. `X-Ray` tracing is lowest on DynamoDB reads and writes and highest on S3 writes and SNS notifications. We believe that this latter overhead is due to the multiple "in-progress" records that X-Ray posts for longer running operations such as these. We observe many such records for S3 write and SNS operations for these benchmarks.

The second row in the top table shows the additional overhead (over `X-Ray`) that `G-Ray-H` introduces. Since *GammaRay* relies on X-Ray for performance data, the total overhead of *GammaRay* versus `Clean` is a combination of both X-Ray and *GammaRay* (last row in both tables). `G-Ray-H` imposes 419ms on function startup. This overhead consists of obtaining a handle to the *GammaRay* DynamoDB table, parsing the function arguments to extract trigger information (either inserted by *GammaRay* or by AWS automatically), and synchronously writing a record containing this payload to the DynamoDB table. The payload contains a timestamp, the request ID, the function ID, and 4-8 additional strings describing the event source (triggering operation). Depending on the event source this payload can vary in size from 16 bytes to arbitrary length (e.g. a DynamoDB key, S3 bucket name, or SNS subject contains application-specific data which can be large).

In our study, the largest payload we have observed is 4 kilobytes. Thus, table write time by the *GammaRay* wrapper can vary but we observe it to be 293ms on average with a standard deviation of 78ms. We measured the time for obtaining the DynamoDB handle from AWS via repeated executions of it alone in a Lambda function (i.e. as another micro-benchmark). We find that for `Clean`, this operation takes 126ms on average (with a standard deviation of 59ms). Because this startup overhead has a significant impact on short-running applications (as shown previously for

the ImgProc application), we are investigating ways of minimizing payload size in particular, and optimizing the *GammaRay* startup process (wrapper), as part of future work.

As shown in columns 3-7 (row 2 of data) in the top table, *GammaRay* introduces 1-34ms of additional overhead (over X-Ray) on individual SDK operations. Because `G-Ray-H` writes to the DynamoDB table once per operation (immediately prior to the operation), only for events that can potentially trigger other Lambda functions, its overhead is small for DDB Read and S3 Read. Only DDB Write, S3 Write, and SNS include operations that are potentially triggering (i.e. they can be event sources). `G-Ray-H` introduces 29ms on DynamoDB write operations, 19ms on S3 write operations, and 34ms on SNS notifications.

The final column of the table shows the average overhead per operation. `X-Ray` introduces 60ms per operation for tracing and `G-Ray-H` introduces an additional 17ms. The sum of these overheads is what `G-Ray-H` requires to produce causal service graphs, through AWS services, annotated with performance data. This data is shown in the bottom row of both tables. `G-Ray-H` introduces on average, a total of 424ms on function startup and 77ms on each AWS SDK call to achieve this. In terms of memory (bottom table), `X-Ray` introduces 16MB of memory, with `G-Ray-H` adding another 5MB, across the micro-benchmarks on average.

V. Related Work

In this section, we overview related work. We focus on research contributions in the area of debugging serverless applications and tracking causal relationships in distributed systems.

The Serverless framework[12] simplifies the process of developing Lambda applications. With an offline plugin, users debug Lambda functions locally. Docker-lambda[47] is a reverse-engineered sandbox that replicates AWS Lambda. It supports all Lambda runtimes and guarantees the same behavior of on AWS Lambda. Josef Spillner studied FaaS and implemented Snafu[48], a modular system compatible to AWS Lambda, which is useful for debugging Lambda applications. The only other extant debugging and analysis support for AWS Lambda applications comes from logging tools. AWS as mentioned previously, provides CloudWatch and X-Ray logging with their limitations detailed in Section II. New Relic[49] and Dashbird[50] provide AWS Lambda monitoring by collecting data from AWS Cloudwatch. Zipkin[51] is a distributed tracing system based on Google Dapper[52]. It helps gather timing data needed to troubleshoot latency problems in Lambda applications.

Capturing causal ordering in support of debugging and performance analysis is well understood and has been extensively researched. Schwarz et al.[22] shows that characterizing the causal relationship is key to understanding distributed applications. Bailis et al. revisits causality in [53] in the context of real-world applications and proposes a number of interesting extensions to the model.

Many distributed track causal relationships in distributed applications. Google has made its production distributed systems tracing infrastructure, Dapper[52] available for public use. The paper describes how they achieve low overhead, application-level transparency, and ubiquitous deployment in the large scale. Fonseca et al. proposed X-trace[54], a tracing framework that provides a comprehensive view of behavior of a modern Internet system that consists of many applications across multiple administrative domains. Kronos [55] utilizes a separate event ordering service to determine the order of interdependent operations in a distributed system. Escriva et al. demonstrates the benefit of providing a Kronos API via several example applications.

Other research contributes new approaches to achieving causal consistency in distributed and scalable datastore systems. Lloyd et al. proposed COPS[56], a key-value store that delivers causal consistency across the wide-area. They identify and define a new consistency model, causal consistency with convergent conflict handling. Bolt-on[57] is a system proposed by Bailis et al., which provides a shim layer that provides causal consistency on top of general-purpose and widely deployed datastores. Saturn[58] is a metadata service for geo-replicated data management systems. It can be used to ensure that remote operations are in a visible order that respects causality. Saturn has been evaluated in Amazon EC2 and the work demonstrates that weakly consistent datastores can provide an improvement (via causal consistency) over eventually consistent models.

VI. Conclusions

Serverless is an emerging cloud service that simplifies and facilitates development and deployment of highly concurrent and scalable applications. Despite its popularity and wide-spread availability, developer support is limited to only basic logging. With this paper, we take an initial step to address this limitation with *GammaRay*, a cloud service that tracks causal dependencies across functions and through cloud services for serverless applications. Causal dependency analysis is an important tool employed in non-serverless concurrent and distributed systems that links events in an application in “happens-before“ order. Such ordering is key for effective distributed debugging, performance and cloud cost optimization, failure recovery, anomaly detection, and root cause performance analysis.

GammaRay tracks dependencies across serverless applications and *through* cloud services. We investigate three different ways of engineering *GammaRay* and

evaluate the overhead of each using serverless micro-benchmarks and applications. We implement *GammaRay* for AWS Lambda Python applications and show that it is possible to leverage existing cloud services for much of its implementation. For the applications, *GammaRay* introduces 12-43% execution overhead and 1-7% memory overhead on average. This translates to approximately 17ms per API call, 419ms on function startup, and 5MB of memory, on average, over X-Ray. Finally, the entirety of this study (development and empirical evaluation triggering hundreds of thousands of lambda functions) was performed using only the AWS Free Tier (i.e. no costs were incurred).

References

[1] "Amazon Serverless computing or Google Function as a Service (FaaS) vs Microservices and Container Technologies," https://www.yenlo.com/blog/amazon-serverless-computing-or-google-function-as-a-service--faas-vs-microservices-and-container-technologies [Online; accessed 1-Nov-2016].

[2] "Serverless Computing," https://en.wikipedia.org/wiki/Serverless_computing [Online; accessed 15-September-2017].

[3] A. Avram, "FaaS, PaaS, and the Benefits of the Serverless Architecture," https://www.infoq.com/news/2016/06/faas-serverless-architecture, [Online; accessed 15-Nov-2016].

[4] "FaaS, PaaS, and the Benefits of the Serverless Architecture," https://www.infoq.com/news/2016/06/faas-serverless-architecture [Online; accessed 1-Nov-2016].

[5] "AWS Lambda," https://aws.amazon.com/lambda/, [Online; accessed 15-Nov-2016].

[6] "AWS Lambda Pricing," https://aws.amazon.com/lambda/pricing/, [Online; accessed 14-September-2017].

[7] "Google Cloud Functions," https://cloud.google.com/functions/docs/, [Online; accessed 15-Nov-2016].

[8] "Azure Functions," https://azure.microsoft.com/en-us/services/functions/, [Online; accessed 15-Nov-2016].

[9] "IBM OpenWhisk," https://developer.ibm.com/openwhisk/, [Online; accessed 15-Nov-2016].

[10] "Iron.io," https://www.iron.io, [Online; accessed 15-Nov-2016].

[11] "WebTask," https://webtask.io, [Online; accessed 15-Nov-2016].

[12] "Serverless Framework," https://serverless.com/, [Online; accessed 11-September-2017].

[13] "Apex," http://apex.run/, [Online; accessed 14-September-2017].

[14] "Go Sparta," http://gosparta.io/, [Online; accessed 14-September-2017].

[15] "Exploiting Parallelism and Scalability: Report on an NSF-Sponsored Workshop," http://people.duke.edu/~bcl15/documents/xps2015-report.pdf, [Online; accessed 14-September-2017].

[16] G. McGrath and P. R. Brenner, "Serverless Computing: Design, Implementation, and Performance," in Distributed Computing Systems Workshops, 2017.

[17] Amazon, "Amazon GreenGrass," 2017, "https://aws.amazon.com/greengrass/" Accessed 15-Sep-2017.

[18] N. Berdy, "How to use Azure Functions with IoT Hub message routing," 2017, "https://azure.microsoft.com/en-us/blog/how-to-use-azure-functions-with-iot-hub-message-routing/".

[19] "AWS X-Ray," https://aws.amazon.com/xray/, [Online; accessed 11-September-2017].

[20] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," Commun. ACM, vol. 21, no. 7, pp. 558–565, Jul. 1978. [Online]. Available: http://doi.acm.org/10.1145/359545.359563

[21] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto, "Causal memory: definitions, implementation, and programming," Distributed Computing, vol. 9, no. 1, pp. 37–49, Mar 1995. [Online]. Available: https://doi.org/10.1007/BF01784241

[22] R. Schwarz and F. Mattern, "Detecting causal relationships in distributed computations: In search of the holy grail," Distrib. Comput., vol. 7, no. 3, Mar. 1994.

[23] N. M. Chakarat Skawratananond and V. K. Garg, "A Lightweight Algorithm for Causal Message Ordering in Mobile Computing Systems," 1999, "http://www.utdallas.edu/ neerajm/publications/conferences/causal.pdf" Accessed 15-Sep-2017.

[24] "Serverless Framework," https://en.wikipedia.org/wiki/Serverless_Framework [Online; accessed 1-Nov-2016].

[25] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Serverless computation with openlambda," in Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing, ser. HotCloud'16. Berkeley, CA, USA: USENIX Association, 2016, pp. 33–39. [Online]. Available: http://dl.acm.org/citation.cfm?id=3027041.3027047

[26] "Function as a Service," https://en.wikipedia.org/wiki/Function_as_a_Service [Online; accessed 15-September-2017].

[27] "Create an API Gateway API for AWS Lambda Functions," http://docs.aws.amazon.com/apigateway/latest/developerguide/integrating-api-with-aws-services-lambda.html, [Online; accessed 14-September-2017].

[28] Amazon Web Services, "AWS DynamoDB Streams Best Practices," "http://docs.aws.amazon.com/amazondynamodb/latest/developer" "guide/Streams.Lambda.BestPracticesWithDynamoDB.html" Accessed 18-Sep-2017.

[29] "Amazon Cloudwatch," https://aws.amazon.com/cloudwatch/, [Online; accessed 11-September-2017].

[30] V. Budilov, "Use Amazon Rekognition to Build an End-to-End Serverless Photo Recognition System," "https://aws.amazon.com/blogs/ai/use-amazon-rekognition-to-build-an-end-to-end-serverless-photo-recognition-system/" Accessed 15-Sep-2017.

[31] M. Raynal, A. Schiper, and S. Toueg, "The causal ordering abstraction and a simple way to implement it," Inf. Process. Lett., vol. 39, no. 6, pp. 343–350, Oct. 1991. [Online]. Available: http://dx.doi.org/10.1016/0020-0190(91)90008-6

[32] A. Schiper, J. Eggli, and A. Sandoz, "A new algorithm to implement causal ordering," in Proceedings of the 3rd International Workshop on Distributed Algorithms. London, UK, UK: Springer-Verlag, 1989, pp. 219–232. [Online]. Available: http://dl.acm.org/citation.cfm?id=645946.675010

[33] H. Jayathilaka, C. Krintz, and R. Wolski, "Performance monitoring and root cause analysis for cloud-hosted web applications," in Proceedings of WWW 2017 (to appear), April 2017.

[34] Amazon Web Services, "Capturing Table Activity with DynamoDB Streams," "http://docs.aws.amazon.com/amazondynamodb/latest/dev" "eloperguide/Streams.html" Accessed 18-Sep-2017.

[35] Wikipedia, "Monkey Patch," "https://en.wikipedia.org/wiki/Mon" "key_patch" Accessed 15-Sep-2017.

[36] "AWS SDK for Python (Boto3)," https://aws.amazon.com/sdk-for-python/, [Online; accessed 14-September-2017].

[37] "Fleece," https://github.com/racker/fleece, [Online; accessed 11-September-2017].

[38] graphviz.org, "Graphviz - Graph Visualization Software," "http://www.graphviz.org" Accessed 18-Sep-2017.

[39] AMPLab, “Big Data Benchmark,” "https://amplab.cs.berkeley.edu/benchmark/" Accessed 15-Sep-2017.

[40] A. P. et al, “A comparison of approaches to large-scale data analysis,” "http://dl.acm.org/citation.cfm?id=1559865" Accessed 15-Sep-2017.

[41] B. Lyston, “Ad Hoc Big Data Processing Made Simple with Serverless MapReduce,” "https://aws.amazon.com/blogs/compute/ad-hoc-big-data-processing-made-simple-with-serverless-mapreduce/" Accessed 15-Sep-2017.

[42] Amazon Web Services, “AWS X-Ray Segment Documents,” "http://docs.aws.amazon.com/xray/latest/devguide/xray-api-segmentdocuments.html" Accessed 18-Sep-2017.

[43] “Student’s T-Test,” https://en.wikipedia.org/wiki/Student’s_t-test [Online; accessed 22-July-2017].

[44] “HDFS,” "https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html" Accessed 15-Sep-2017.

[45] “Apache Hadoop,” http://hadoop.apache.org/ [Online; accessed 2-January-2017].

[46] “Apache Spark,” http://spark.apache.org/ [Online; accessed 2-January-2017], https://spark.apache.org/ [Online; accessed 14-Feb-2015].

[47] “Docker-lambda,” https://github.com/lambci/docker-lambda, [Online; accessed 11-September-2017].

[48] J. Spillner, “Snafu: Function-as-a-service (faas) runtime design and implementation,” CoRR, vol. abs/1703.07562, 2017.

[49] “New Relic Monitors Serverless Computing with AWS Lambda Integration,” https://blog.newrelic.com/2016/11/29/aws-lambda-integration-serverless-infrastructure/, [Online; accessed 11-September-2017].

[50] “Dashbird,” https://dashbird.io/, [Online; accessed 11-September-2017].

[51] “Zipkin,” http://zipkin.io/, [Online; accessed 11-September-2017].

[52] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, “Dapper, a large-scale distributed systems tracing infrastructure,” Google, Inc., Tech. Rep., 2010. [Online]. Available: https://research.google.com/archive/papers/dapper-2010-1.pdf

[53] P. Bailis, A. Fekete, A. Ghodsi, J. Hellerstein, and I. Stoica, “The potential dangers of causal consistency and an explicit solution,” in Proceedings of the Third ACM Symposium on Cloud Computing, ser. SoCC ’12, 2012.

[54] R. Fonseca, G. Porter, R. Katz, S. Shenker, and I. Stoica, “X-trace: A pervasive network tracing framework,” in USENIX Conference on Networked Systems Design and Implementation, 2007.

[55] R. Escriva, A. Dubey, B. Wong, and E. Sirer, “Kronos: The design and implementation of an event ordering service,” in European Conference on Computer Systems, ser. EuroSys ’14, 2014.

[56] W. Lloyd, M. Freedman, M. Kaminsky, and D. Andersen, “Don’t settle for eventual: Scalable causal consistency for wide-area storage with cops,” in ACM Symposium on Operating Systems Principles, 2011.

[57] P. Bailis, A. Ghodsi, J. Hellerstein, and I. Stoica, “Bolt-on causal consistency,” in ACM SIGMOD International Conference on Management of Data, 2013.

[58] M. Bravo, L. Rodrigues, and P. V. Roy, “Saturn: A distributed metadata service for causal consistency,” in European Conference on Computer Systems, 2017.