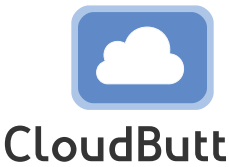


HORIZON 2020 FRAMEWORK PROGRAMME

CloudButton

(grant agreement No 825184)

Serverless Data Analytics Platform

D4.3 Full implementation of the BLOSSOM middleware

Due date of deliverable: 15-12-2021
Actual submission date: 01-07-2022

Start date of project: 01-01-2019 Duration: 36 months

Summary of the document

Document Type	Report
Dissemination level	Public
State	v2.0
Number of pages	56
WP/Task related to this document	WP4 / All tasks
WP/Task responsible	IMT
Leader	Pierre Sutra (IMT)
Technical Manager	Tristan Tarrant (RHAT)
Quality Manager	Marc Sánchez (URV)
Author(s)	Daniel Barcelona-Pons (URV), Anatole Lefort (IMT), Pierre Sutra (IMT), Gerard París Aixalà (URV), Pedro García (URV), Marc Sanchez (URV), Tristan Tarrant (RHAT)
Partner(s) Contributing	IMT, URV, RHAT
Document ID	CloudButton_D4.3_Public.pdf
Abstract	This document describes in detail CRUCIAL, a framework to program efficient stateful serverless applications.
Keywords	serverless, FaaS, distributed storage, big data, data analytics, machine learning

History of changes

Version	Date	Author	Summary of changes
0.1	13-12-2021	Pierre Sutra	Initial version
0.2	10-05-2022	Pierre Sutra	§4 added
0.3	15-05-2022	Pierre Sutra	§5.1 added
0.4	20-05-2022	Pierre Sutra	Modifications to §4
0.5	01-06-2022	Pierre Sutra	Some more edits
0.6	06-06-2022	Tristan Tarrant	Modifications to §5
1.0	27-06-2022	Pierre Sutra	First full draft
1.1	28-06-2022	Daniel Barcelona	Review §3
2.0	05-07-2022	Pierre Sutra	Final version

Table of Contents

1 Executive summary

Serverless computing greatly simplifies the use of cloud resources. In particular, Function-as-a-Service (FaaS) enables programmers to develop applications as individual functions that can run and scale independently. The CloudButton project aims at leveraging this new infrastructure to bring closer to the end user the vast amount of data and computing power available in the cloud.

In this document, we present CRUCIAL, a system to program and execute efficient applications that target serverless platforms.[1] CRUCIAL is built upon the key insight that serverless computing resembles to concurrent programming at the scale of the cloud. As a consequence, a distributed shared memory layer is the right answer to the need for fine-grained state management and coordination in serverless.

The programming model of CRUCIAL keeps the simplicity of serverless computing and allows to write effortlessly parallel code for this new environment. Our system is structured into a compute and a storage tier, while providing a convenient user-facing library to the programmer. The compute tier runs atop a FaaS platform, while storage consists of an efficient in-memory distributed storage layer.

We validate CRUCIAL with the help of micro-benchmarks and various data analytics applications. To update shared data and synchronize cloud functions, CRUCIAL has better performance than alternative solutions. We show that it also simplifies the writing of parallel tasks (e.g., Monte Carlo) and ML algorithms (such as *k*-means clustering and logistic regression). Compared to Apache Spark, it obtains close or better performance at similar cost. We also use CRUCIAL to port a state-of-the-art multi-threaded ML library to serverless, as well as *NIX scripts. CRUCIAL brings elasticity and on-demand capabilities to these traditional single-machine programs. The performance of the serverless versions of these two programs are on par with the one offered by a dedicated cluster of high-end servers. A key feature of CRUCIAL is that it also allows to port programs to serverless in a few lines of code. All the above applications were ported while modifying less than 4% of the original code base. This document describes in full length the CRUCIAL prototype, its programming API and the underlying distributed storage and compute tiers.

Comparison with D4.2 This deliverable is an iteration over the previous deliverable of this work package (WP4). It contains three key novelties: *(i)* The serverless shell (`sshell`) is a new tool to execute *NIX commands remotely in the function-as-a-service platform. `sshell` permits to reuse an existing code base while benefiting from the massive power of serverless and paying only for the resources used. *(ii)* In this document, we also introduce J-NVM, a fully-fledged support for non-volatile persistent memory in the Java language. J-NVM allows serverless functions to quickly access persisted data in the distributed shared memory layer of CRUCIAL. *(iii)* The last novelty is a new Kubernetes operator for the shared memory layer. This operator simplifies data access within a unified modern container-based ecosystem to serverless (and traditional) distributed applications.

[1] In the project proposal, CRUCIAL is codenamed BLOSSOM.

2 Introduction

Context Serverless computing is a paradigm that removes much of the cloud's usage complexity by abstracting away the provisioning of compute resources. This fairly new model was started by services such as Google BigQuery [117] and AWS Glue [13], and evolved into Function-as-a-Service (FaaS) computing platforms, such as AWS Lambda and Google Cloud Functions. In these services, a user-defined function and its dependencies are deployed to the cloud, where they are managed by the provider and executed on demand at scale.

Current practice shows that the FaaS model works well for applications that require a small amount of storage and memory due to the operational limits set by the cloud providers (see, for instance, AWS Lambda [10]). However, there are more limitations. While functions can initiate outgoing network connections, they cannot directly communicate between each other, and have little bandwidth compared to a regular virtual machine [27, 155]. This is because this model was originally designed to execute event-driven, stateless functions in response to user actions or changes in the storage tier (e.g., uploading a file to Amazon S3 [9]). Despite these constraints, recent works have shown how this model can be exploited to process and transform large amounts of data [74, 124, 130], encode videos [48], execute linear algebra tasks [133], and perform Monte Carlo simulations with large amounts of parallelism [70].

Problematic The above works, such as PyWren [74, 130] and ExCamera [48], prove that FaaS platforms can be programmed to perform a wide variety of embarrassingly parallel computations. Yet, these tools face also fundamental challenges when used out-of-the-box for many popular tasks. Although the list is too long to recount here, convincing cases of these ill-suited applications are distributed stateful computations such as machine learning (ML) algorithms. Just an imperative implementation of k-means [96] raises several issues: first, the need to efficiently handle a globally-shared state at fine granularity (the cluster centroids); second, the problem to globally synchronize cloud functions, so that the algorithm can correctly proceed to the next iteration; and finally, the prerogative that the shared state survives system failures.

Current serverless systems do not address these issues effectively. First, due to the impossibility of function-to-function communication, the prevalent practice for sharing state across functions is to use remote storage. For instance, serverless frameworks, such as PyWren and numpywren [133], use highly-scalable object storage services to transfer state between functions. Since object storage is too slow to share short-lived intermediate state in serverless applications [85], some recent works use faster storage solutions. This has been the path taken by Locus [124], which proposes to combine fast, in-memory storage instances with slow storage to scale shuffling operations in MapReduce. However, with all the shared state transiting through storage, one of the major limitations of current serverless systems is the lack of support to handle mutable state at a fine granularity (e.g., to efficiently aggregate small granules of updates). Such a concern has been recognized in various works [27, 75], but this type of fast, enriched storage layer for serverless computing is not available today in the cloud, leaving fine-grained state sharing as an open issue.

Similarly, FaaS orchestration services (such as AWS Step Functions [12] or OpenWhisk Composer [50]) offer limited capabilities to coordinate serverless functions [54, 75]. They have no abstraction to signal a function when a condition is fulfilled, or for multiple functions to synchronize, e.g., in order to guarantee data consistency, or to ensure joint progress to the next stage of computation. Of course, such fine-grained coordination should be also low-latency to not significantly slow down the application. Existing stand-alone notification services, such as AWS SNS [21] and AWS SQS [56], add significant latency, sometimes hundreds of milliseconds. This lack of efficient cloud coordination tools means that each serverless framework needs to develop its own mechanisms. For instance, PyWren enforces the synchronization of map and reduce stages through object storage, while ExCamera has built a notification system using a long-running VM-based rendezvous server. As of today, there is no general way to let multiple functions synchronize via abstractions hand-crafted by users, so that fine-grained coordination can be truly achieved.

Contributions To overcome the aforementioned issues, we propose CRUCIAL, a system for the development of efficient (both stateful and stateless) serverless applications. To simplify the writing of an application, CRUCIAL provides a thread abstraction that maps a thread to the invocation of a serverless function: the *cloud thread*. This abstraction can be extended to build task management systems with serverless thread pools. To support fine-grained state management and coordination, our system builds a distributed shared object (DSO) layer on top of a low-latency in-memory data store. This layer provides out-of-the-box strong consistency guarantees, simplifying the semantics of global state mutation across cloud threads. Since global state is manipulated as remote objects, the interface for mutable state management becomes virtually unlimited, only constrained by the expressiveness of the programming language (Java in our case). The result is that CRUCIAL can operate on small data granules, making it very easy to develop applications that have fine-grained state sharing needs. CRUCIAL also leverages this layer to implement fine-grained coordination. For applications that require longer retention of in-memory state, CRUCIAL ensures data durability through replication. To ensure the consistency of replicas, CRUCIAL uses state machine replication (SMR), so that any acknowledged write can survive failures.

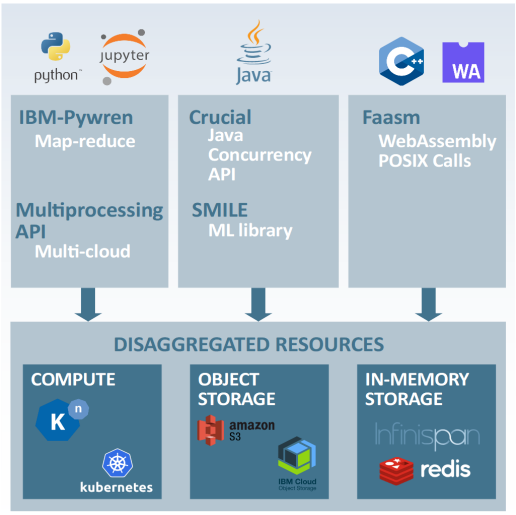


Figure 1: CloudButton architecture

Most importantly, CRUCIAL offers all of the above guarantees with almost no increase in the programming complexity of the serverless model. With the help of a few annotations and constructs, developers can run their single-machine, multi-threaded, stateful code in the cloud as serverless functions. CRUCIAL's programming constructs enable developers to enforce atomic operations on shared state, as well as to finely synchronize functions at the application level, so that (imperative) implementations of popular algorithms such as *k*-means can be effortlessly ported to serverless platforms.

CRUCIAL in CloudButton The CloudButton project uses serverless technologies to simplify data analytics and data processing for everyday programmers. Figure 1 presents a general architecture of the project. CRUCIAL was developped in the context of CloudButton. It provides a general framework to port and develop Java applications for serverless. CRUCIAL is complementary to LITHOPS [122] and FAASM [123]. All of these works are united by a common principle of *transparency*: they offer familiar programming concepts and abstractions, allowing to program serverless platforms much like a regular computer.

Outline The remainder of this document is as follows:

- (§3) First, we detail the core abstractions and internals of the CRUCIAL framework. We show that CRUCIAL is suited for many stateful distributed applications such as traditional parallel computations, machine learning algorithms, and complex concurrency situations. Using extensive evaluation of *k*-means and logistic regression over a 100 GB dataset, we show that CRUCIAL can lead to $18-40\%$ performance improvement over Apache Spark running on dedicated instances at similar cost. Using CRUCIAL, we also port to serverless part of Smile [92], a state-of-the-art multi-threaded ML library. The portage impacts less than 4% of the original code base. It brings elasticity and on-demand capabilities to a traditional single-machine program. Its performance are on par with a dedicated high-end server: using a random forest classification algorithm, the portage with 200 cloud threads is up to 30% faster than a 4-CPU 160-threads dedicated server solution.

- (§4) This section presents the serverless shell (`sshell`). `sshell` is built atop CRUCIAL. It brings the massive copmutation power of the cloud to regular shell scripts. `sshell` is built around a small set of components that includes a new inter-process communication layer for serverless. We evaluate it in AWS Lambda using several micro-benchmarks and a large-scale application. Our results show that `sshell` achieves comparable or better performance than a high-end server. Moreover, it can be faster and more cost-efficient than a cluster-based solution to mine large datasets.
- (§5) CRUCIAL includes a layer of distributed shared objects (DSO). This layer allows serverless functions to communicate one to another and to maintain a persistent state for the application. It is implemented atop Infinispan, an industrial-grade distributed storage system [103]. The CloudButton project brought many improvements to Infinispan. This includes a support for non-volatile memory, ahead-of-time compilation and a new algorithm for data placement. We also provide a new operator for Kubernetes. This operator simplifies data access within a unified modern container-based ecosystem to serverless (and traditional) distributed applications.
- (§6) Complementary to the above works, we also investigated several innovative ideas to improve CRUCIAL. This includes multiple research results on data distribution, replication and persistence. We present the exploratory works conducted in WP4. Part of this work will be included in future releases of CRUCIAL.
- (§7-§8) Before closing, this document reviews the state of the art on serverless computing and make a comparison with CRUCIAL.

Many of the contributions covered in this document appeared to top tier conferences and journals in their respective domains [17, 42, 44, 52, 60, 91, 99, 144]. Most notably: TOSEM'22, SOSP'21, Middleware'21,'19, Eurosys'20,'21 and DSN'19.

3 Stateful serverless programming with CRUCIAL

CRUCIAL is a powerful framework to create and execute complex serverless tasks atop a Function-as-a-Service (FaaS) architecture. This section presents the programming model and the internals of our system in detail; then some evaluation results to assess its capabilities.

3.1 Using CRUCIAL

Below, we present the programming interface of CRUCIAL and illustrate it with several applications. We also provide a methodology to port a conventional single-machine application to serverless with the help of our framework. Note that the programming model of CRUCIAL is also recalled in Deliverable [123, §4].

3.1.1 Programming model

The programming model of CRUCIAL is object-based and can be integrated with any object-oriented programming language. As Java is the language supported in our implementation, the following description considers its jargon.

Overall, a CRUCIAL program is strongly similar to a regular multi-threaded, object-oriented Java one, besides some additional annotations and constructs. Table 1 summarizes the key abstractions available to the programmer that are detailed hereafter.

Table 1: Programming abstractions

Abstraction	Description
`CloudThread`	Cloud functions are invoked like threads.
`ServerlessExecutorService`	A simple executor service for task groups and distributed parallel *for*s.
Shared objects	Linearizable (wait-free) distributed objects (e.g., `AtomicInt`, `AtomicLong`, `AtomicBoolean`, `AtomicByteArray`, `List`, `Map`).
Synchronization objects	Shared objects providing primitives for thread synchronization (e.g., `Future`, `Semaphore`, `CyclicBarrier`).
`@Shared`	User-defined shared objects. Object methods run on the DSO servers, allowing fine-grained updates and aggregates (e.g., `.add()`, `.update()`, `.merge()`).
Data persistence	Long-lived shared objects are replicated. Persistence may be activated with `@Shared(persistence=true)`.

Cloud threads A `CloudThread` is the smallest unit of computation in CRUCIAL. Semantically, this class is similar to a `Thread` in conventional concurrent computing. To write an application, each task is defined as a `Runnable` and passed to a `CloudThread` that executes it. The `CloudThread` class hides from the programmer the execution details of accessing the underlying FaaS platform. This enables access transparency to remote resources [55].

Serverless executor service The `ServerlessExecutorService` class may be used to execute both `Runnable` and `Callable` instances in the cloud. This class implements the `ExecutorService` interface, allowing the submission of individual tasks and fork-join parallel constructs (`invokeAll`). The full expressiveness of the original JDK interface is retained. In addition, this executor also includes a distributed parallel *for* to run n iterations of a loop across m workers. To use this feature, the user specifies the in-loop code (through a functional interface), the boundaries for the iteration index, and the number of workers m.

State handling CRUCIAL includes a library of base shared objects to support mutable shared data across cloud threads. The library consists of common objects such as integers, counters, maps, lists and arrays. These objects are *wait-free* and *linearizable* [97]. This means that each method invocation terminates after a finite number of steps (despite concurrent accesses), and that concurrent method invocations behave as if they were executed by a single thread. CRUCIAL also gives programmers the ability to craft their own custom shared objects by decorating a field declaration with the `@Shared` annotation. Annotated objects become globally accessible by any thread. CRUCIAL refers to an object with a key crafted from the field's name of the encompassing object. The programmer can override this definition by explicitly writing `@Shared(key=k)`. Our framework supports distributed references, permitting a reference to cross the boundaries of a cloud thread. This feature helps to preserve the simplicity of multi-threaded programming in CRUCIAL.

Data Persistence Shared objects in CRUCIAL can be either *ephemeral* or *persistent*. By default, shared objects are ephemeral and only exist during the application lifetime. Once the application finishes, they are discarded. Ephemeral objects can be lost, e.g., in the event of a server failure in the DSO layer, since the cost of making them fault-tolerant outweighs the benefits of their short-term availability [85]. Nonetheless, the annotation `@Shared(persistent=true)` enables to make them persistent. Persistent objects outlive the application lifetime and are only removed from storage by an explicit call.

```
1 public class PiEstimator implements Runnable {
2   private final static long ITERATIONS = 100_000_000;
3   private Random rand = new Random();
4   @Shared(key="counter")
5   AtomicLong counter = new AtomicLong(0);
6
7   public void run() {
8     long count = 0;
9     double x, y;
10    for (long i = 0L; i < ITERATIONS; i++) {
11      x = rand.nextDouble();
12      y = rand.nextDouble();
13      if (x * x + y * y <= 1.0) count++;
14    }
15    counter.addAndGet(count);
16  }
17 }
18
19 List<Thread> threads = new ArrayList<>(N_THREADS);
20 for (int i = 0; i < N_THREADS; i++) {
21   threads.add(new CloudThread(new PiEstimator()));
22 }
23 threads.forEach(Thread::start);
24 threads.forEach(Thread::join);
25 double output = 4.0 * counter.get() / (N_THREADS * ITERATIONS);
```

Listing 1: Monte Carlo simulation to approximate π.

```
1 ExecutorService se = new ServerlessExecutorService();
2 List<Callable> tasks = IntStream.range(0, N_THREADS)
3       .mapToObj(i -> Executors.callable(new PiEstimator())).collect(Collectors.toList());
4 se.invokeAll(tasks);
```

Listing 2: Using the `ServerlessExecutorService` to perform a Monte Carlo simulation.

Synchronization Current serverless frameworks support only uncoordinated embarrassingly parallel operations, or bulk synchronous parallelism (BSP) [63, 75]. To provide fine-grained coordination of cloud threads, CRUCIAL offers several primitives such as cyclic barriers and semaphores. These coordination primitives are semantically equivalent to those in the standard `java.util.concurrent` library. They allow a coherent and flexible model of concurrency for cloud functions that is nonexistent as of today.

3.1.2 Sample applications

Listing 1 presents an application implemented with CRUCIAL. This simple program is a multi-threaded Monte Carlo simulation that approximates the value of π. It draws a large number of random points and computes how many of them fall in the circle enclosed by the unit square. The ratio of points falling in the circle converges with the number of trials toward $\pi/4$ (line 25).

The application first defines a regular `Runnable` class that carries the estimation of π (lines 1 to 17). To parallelize its execution, lines 23 and 24 run a fork-join pattern using a set of `CloudThread` instances. The shared state of the application is a `counter` object (line 5). This counter maintains the total number of points falling into the circle, which serves to approximate π. It is updated by the threads concurrently using the `addAndGet` method (line 15).

The previous fork-join pattern can also be implemented using the `ServerlessExecutorService`. In this case, we simply replace lines 19 to 24 in Listing 1 with the content of Listing 2.

A second application is shown in Listing 3. This program outputs an image approximating the Mandelbrot set (a subset of $\mathbb{C}$) with a gradient of colors. The output image is stored in a CRUCIAL shared object (line 3). To create the image, the application computes the color of each pixel (line 5). The color indicates when the pixel escaped from the Mandelbrot set (after a bounded number of

```
1 public class Mandelbrot implements Serializable {
2   @Shared(key = "mandelbrotImage")
3   private MandelbrotImage image = new MandelbrotImage();
4
5   private static int[] computeRow(int row, int width, int height, int maxIters) {...}
6
7   private void compute() {
8     image.init(COLUMNS, ROWS);
9     ServerlessExecutorService se = new ServerlessExecutorService();
10    se.invokeIterativeTask(row -> image.setRowColor(row, computeRow(row, COLUMNS, ROWS,
          MAX_INTERNAL_ITERS)), N_TASKS, 0, ROWS);
11    se.shutdown();
12  }
13 }
```

Listing 3: Mandelbrot set computation in a distributed parallel *for*.

iterations). The rows of the image are processed in parallel, using the `invokeIterativeTask` method of the `ServerlessExecutorService` class. As seen at line 10, this method takes as input a functional interface (`IterativeTask`) and three integers. The interface defines the function to apply on the index of the *for* loop. The integers define respectively the number of tasks among which to distribute the iterations, and the boundaries of these iterations (`fromInclusive`, `toExclusive`).

This second example illustrates the expressiveness and convenience of our framework. In particular, as in multi-threaded programming, CRUCIAL allows to define concurrent tasks with lambda expressions and pass them shared variables defined in the encompassing class.

3.1.3 Portage to serverless

The previous sections detail the programming interface of CRUCIAL and illustrate it with base applications. In this section, we turn our attention to the problem of porting existing applications to serverless. We first explain the benefits an application may have from a port to serverless and a methodology to achieve it. Further, we present the limitations of this methodology and how the programmer can overcome them. In §3.4.4, we evaluate the successful application of this methodology to port Smile [92], a state-of-the-art machine learning library.

Benefits & Target applications CRUCIAL can be used not only to program serverless-native applications, but also to port existing single-machine applications to serverless. Successfully porting an application comes with several incentives; namely the ability to *(i)* access on-demand computing resources; *(ii)* scale these resources dynamically; and *(iii)* benefit from a fine-grained pricing for their usage. To match the programming model of CRUCIAL, Java applications that can benefit from a portage should be multi-threaded. Moreover, as with other parallel programming frameworks (e.g., MPI [140] or MapReduce [38]), they should be inherently parallel.

Methodology CRUCIAL allows to port an existing Java multi-threaded application to serverless with low effort. To this end, the following steps should be taken: **(1)** The programmer replaces the `ExecutorService` or `Thread` instances with their CRUCIAL counterparts, as listed in Table 1. **(2)** The programmer makes `Serializable` each immutable object passed between cloud threads. **(3)** The programmer substitutes the concurrent mutable objects shared by threads with the equivalent ones provided by the DSO layer. For example, an instance of `java.util.atomic.AtomicBoolean` is replaced with `org.crucial.dso.AtomicBoolean`. **(4)** Regarding synchronization primitives, the programmer must transform them into distributed objects. As an example, a cyclic barrier can be replaced with `org.crucial.dso.CyclicBarrier`, an implementation based internally on a monitor. Another option is `org.crucial.dso.ScalableCyclicBarrier`, that implements the collective described in [65]. **(5)** If the application uses the `synchronized` keyword, some rewriting is necessary. Recall that this keyword is specific to the Java language and allows to use any (non-primitive) object as a monitor [66].

```
1 public class WordCount {
2   private Document document = new Document(LOCATION);
3   private String word = "serverless";
4
5   private void compute() {
6     AtomicLong counter = new AtomicLong("wordcount");
7     ServerlessExecutorService se = new ServerlessExecutorService();
8     se.invokeIterativeTask(i -> counter.addAndGet(countWords(word, document.split(i))), N_TASKS, 0,
          N_TASKS);
9   }
10 }
```

Listing 4: Parallel word count.

CRUCIAL does not support the `synchronized` keyword out of the box since it would require modifying the JVM. Two solutions are offered to the programmer: *(i)* create a monitor object in DSO and use it where appropriate; or *(ii)* create a method for the object used as a monitor that contains all the code in the `synchronized{..}` block. Then, this object is annotated as `@Shared` in the application, and the method called where appropriate. The first solution is simple, but it might not be the most efficient since it requires to move data back and forth the cloud threads that use the monitor. The second solution needs rewriting part of the original application. However, it is more in line with the object-oriented approach in CRUCIAL, where an operation updating a shared object is accessible through a (linearizable) method, and it may perform better.

Limitations & Solutions The above methodology works for most applications, yet it has limitations. First, some threading features are not available in the framework —e.g., signaling a cloud thread. Second, CRUCIAL does not natively support arrays (e.g., `T[] tab`). Indeed, recall that the Java language offers native methods to manipulate such data types. For instance, calling `tab[i]=x` assigns the value (or reference) x to `tab[i]`. Transforming a native call is not possible with just annotations.[2] The solution to these two problems is to rewrite the application appropriately, as in the case of `synchronized`.

Another issue is related to data locality. Typically, a multi-threaded application initializes shared data in the main thread and then makes it accessible to other threads for computation. Porting such a programming pattern to FaaS implies that data is initialized at the machine starting up the application, then serialized to be accessible elsewhere; this is very inefficient. Instead, a better approach is to pass a distributed reference that is lazily de-referenced by the thread. To illustrate this point, consider Listing 4 which counts the number of occurrences of the word ‘‘`serverless`’’ in a document. The application first constructs a reference to the document (line 1). Then, the document is split into chunks. For each chunk, the number of occurrences of the word is counted by a cloud thread (line 2). The results are then aggregated in the shared counter ‘‘`wordcount`’’. Reading the document in full at line 1 and serializing it to construct the chunks is inefficient. Instead, the application should send a distributed reference to the cloud threads at line 2. Then, upon calling `split`, the chunks are created on each thread by fetching the content from remote storage.

3.2 Design

Figure 2 presents the overall architecture of CRUCIAL. In what follows, we detail its components and describe the lifecycle of an application in our system.

CRUCIAL encompasses three main components (from left to right in Figure 2): the client application; the FaaS computing layer that runs the cloud threads; and the DSO layer that stores the shared objects. A client application differs from a regular JVM process in two aspects: threads are executed as cloud functions, and they access shared data using the DSO layer. Moreover, CRUCIAL applications may also rely on external cloud services, such as object storage to fetch input data (not modeled

[2] It is however possible with bytecode manipulation tools (e.g., [24]).

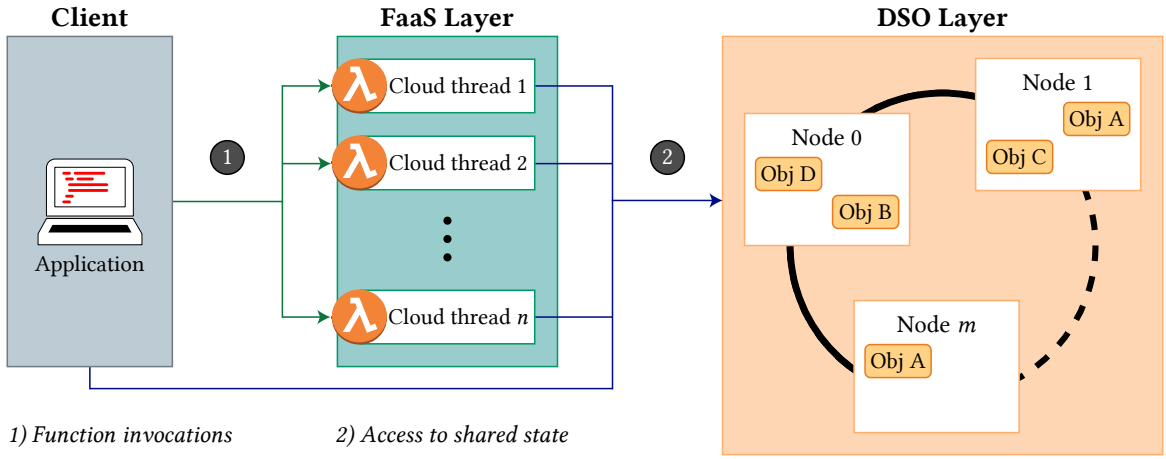


Figure 2: CRUCIAL's overall architecture. A client application runs a set of cloud threads in the FaaS layer. The cloud threads and the client have access to the shared state stored in the DSO layer.

in Figure 2).

3.2.1 The distributed shared objects layer

Each object in the DSO layer is uniquely identified by a reference. Fine-grained updates to the shared state are implemented as methods of these objects. Given an object of type T, the reference to this object is (T, k), where k is either the name of the annotated object field or the value of the parameter *key* in the annotation `@Shared(key=k)`. When a cloud thread accesses an object, it uses its reference to invoke remotely the appropriate method.

CRUCIAL constructs the DSO layer using consistent hashing [82], similarly to Cassandra [86]. Each storage node knows the full storage layer membership and thus the mapping from data to node. The location of a shared object o is determined by hashing the reference (T, k) of o. This offers the following usual benefits: 1. no broadcast is necessary to locate an object; 2. disjoint-access parallelism [71] can be exploited; and 3. service interruption is minimal in the event of server addition and removal. The latter property is useful for persistent objects, as detailed next.

Persistence One interesting aspect of CRUCIAL is that it can ensure durability of the shared state. This property is appealing, for instance, to support the different phases of a machine learning workflow (training and inference). Objects marked as persistent are replicated *rf* (replication factor) times in the DSO layer. They reside in memory to ensure sub-millisecond read/write latency and can be passivated to stable storage using standard mechanisms (marshalling). When a cloud thread accesses a shared object, it contacts one of the server nodes. The operation is then forwarded to the actual replicas storing the object. Each replica executes the incoming call, and one of them sends the result back to the caller. Notice that for ephemeral —non-persistent— objects, *rf* is 1.

Extensions Traditionally, data is persisted on disk asynchronously. This avoids the cost of having a disk access in the critical path but lower durability: if the system stops before data is flushed, one or more updates can be lost. The release of Intel Optane DC, and the global availability of persistent memory, offers the promise to reduce drastically the cost of durability. DSO now integrates a new persistent backend to leverage this medium. The backend is built using the J-NVM framework and we detail it in §5.1.

Consistency CRUCIAL provides linearizable objects and programmers can reason about interleaving as in the shared-memory case. This greatly simplifies the writing of stateful serverless applications. For persistent objects, consistency across replicas is maintained with the help of state machine

replication (SMR) [131]. To handle membership changes, the DSO layer relies on a variation of virtual synchrony [29]. Virtual synchrony provides a totally-ordered set of views to the server nodes. In a given view, for some object x, the operations accessing x are sent using total order multicast. The replicas of x deliver these operations in a total order and apply them on their local copy of x according to this order. A distinct replica (primary) is in charge of sending back the result to the caller. When a membership change occurs, the nodes re-balance data according to the new view. In [17], we provide a full pseudo-code of this construction together with a proof of correctness.

Extensions Weaker consistency models were also investigated through the notion of degradability – the full details appear in §6.2.

3.2.2 Fast aggregates through remote procedure call

Stateful applications commonly aggregate and combine small granules of data (e.g., the training phase of a ML algorithm). Unfortunately, cloud functions are not network-addressable and run separate from data. As a consequence, these applications are routinely left with no other choice but to "ship data to code". This is known as one of the biggest downsides of FaaS platforms [63].

To illustrate this point, consider an `AllReduce` operation where N cloud functions need to aggregate their results by applying some commutative and associative operator f (e.g., a sum). To achieve this, each function first writes its local result in the storage layer. Then, the functions await that their peers do the same, fetch the N results, and apply f sequentially. This algorithm is expensive and entails a communication cost of N^2 messages with the storage layer.

CRUCIAL fully resolves this anti-pattern with minimal efforts from the programmer. Complex computations are implemented as object methods in DSO and called by the cloud functions where appropriate. Going back to the above example, each function simply calls $f(r)$ on the shared object, where r is its local result. This is for instance the case at line 8 in Listing 4 with the method `counter.addAndGet`. With this approach, communication complexity is reduced to $\mathcal{O}(N)$ messages with the storage layer.

We exploit this key feature of CRUCIAL in our serverless implementation of several ML algorithms (e.g., k-means, linear regression, random forest). Its performance benefits are detailed in §3.4.2.

3.2.3 Lifecycle of an application

The lifecycle of a CRUCIAL application is similar to that of a standard multi-threaded Java one. Every time a `CloudThread` is started, a Java thread (i.e., an instance of `java.lang.Thread`) is spawned on the client. This thread pushes the `Runnable` code attached to the `CloudThread` to a generic function in the FaaS platform. Then, it waits for the result of the computation before it returns.

Accesses to some shared object of type `T` at cloud threads (or at the client) are mediated by a proxy. This proxy is instantiated when a call to "`new T()`" occurs, and either the newly created object of type `T` belongs to CRUCIAL's library, or it is tagged `@Shared`. As an example, consider the counter used in Listing 1. When an instance of `PiEstimator` is spawned, the field `counter` is created. The "`new`" statement is intercepted and a local proxy for the counter is instantiated to mediate calls to the remote object hosted in the DSO layer. If this object does not exist in the DSO layer, it is instantiated using the constructor defined at line 5. From thereon, any call to `addAndGet` (line 15) is pushed to the DSO layer. These calls are delivered in total order to the object replicas where they are applied before sending back a response value to the caller.

The Java thread remains blocked until the cloud function terminates. Such a behavior gives cloud threads the appearance of conventional threads; minimizing code changes and allowing the use of the `join()` method at the client to establish synchronization points (e.g., fork/join pattern). It must be noted, however, that as cloud functions cannot be canceled or paused, the analogy is not complete. If any failure occurs in a remote cloud function, the error is propagated back to the client application for further processing.

The case of the `ServerlessExecutorService` builds on the same idea as `CloudThread`. A standard Java thread pool is used internally to manage the execution of all tasks. In the case of a callable task, the result is accessible to the caller in a `Future` object.

3.2.4 Fault tolerance

Fault tolerance in CRUCIAL is based on the disaggregation of the compute and storage layers. On the one hand, writes to DSO can be made durable with the help of data replication. In such a case, CRUCIAL tolerates the joint failure of up to $rf - 1$ servers.[3] On the other hand, CRUCIAL offers the same fault-tolerance semantics in the compute layer as the underlying FaaS platform. In AWS Lambda, this means that any failed cloud thread can be re-started and re-executed with the exact same input. Thanks to the cloud thread abstraction, CRUCIAL allows full control over the retry system. For instance, the user may configure how many retries are allowed and/or the time between them. If retries are permitted, the programmer should ensure that the re-execution is sound (e.g., it is idempotent). Fortunately, atomic writes in the DSO layer make this task easy to achieve. Considering the *k*-means example depicted in Listing 5 (or another iterative algorithm), it simply consists in sharing an iteration counter (line 6). When a thread fails and re-starts, it fetches the iteration counter and continues its execution from thereon.

3.3 Implementation

The implementation of CRUCIAL is open source and available online [6]. It consists of around 10K SLOC, including scripts to deploy and run CRUCIAL applications in the cloud. The DSO layer is written atop the Infinispan in-memory data grid [103] as a partial rewrite of the CRESON project [145].

A CRUCIAL application is written in Java and uses Apache Maven to compile and manage its dependencies. It employs the abstractions listed in Table 1 and has access to scripts that automate its deployment and execution in the cloud.

To run cloud threads, our prototype implementation relies on AWS Lambda. Lambda functions are deployed with the help of a Maven plugin [5] and invoked via the AWS Java SDK. To control the replay mechanism, calls to Lambda are synchronous. The adherence of CRUCIAL to Lambda is limited and the framework can execute atop a different FaaS platform with a few changes. In §7.1, we discuss this platform dependency.

The `ServerlessExecutorService` implements the base `ExecutorService` interface. It accepts `Callable` objects and task collections. The invocation of a `Callable` returns a (local) `Future` object. This future is completed once a response from AWS Lambda is received. For `Runnable` tasks, the response is empty unless an error occurs. In that case, the system interprets it and throws an exception at the client machine, referencing the cause.

To create a distributed parallel *for*, the programmer uses an instance of `IterativeTask` (as illustrated at line 10 in Listing 3). This functional interface is similar to `java.util.function.Consumer`, but limited to iteration indexes (i.e., the input parameter must be an integer). Internally, the iterative task creates a collection of `Callable` objects. In our current prototype, the scheduling is static and based on the number of workers and tasks given in parameter.

When an AWS Lambda function is invoked, it receives a user-defined `Runnable` (or `Callable`) object. The object and its content are marshalled and shipped to the remote machine, where they are re-created. Initialization parameters can be given to the constructor. As pointed out in §3.1.1, a distributed reference is sent in lieu of a shared object.

Proxies for the shared objects are waved into the code of the client application using AspectJ [83]. In the case of user-defined objects, the aspects are applied to the annotated fields (see §3.1.1). Such objects must be serializable and they should contain an empty constructor (similarly to a JavaBean). The `jar` archive containing the definition of the objects is uploaded to the DSO servers where it is dynamically loaded.

[3] Synchronization objects (see Table 1) are not replicated. This is not an important issue due to their ephemeral nature.

Synchronization objects (e.g., barriers, semaphores, futures) follow the structure of their Java counterparts. They rely internally on Java monitors. When a client performs a call to a remote object, it remains blocked until the request responds. The server processes the operation with a designated thread. During the method invocation, that thread may suspend itself through a `wait` call on the object until another thread awakes it.

State machine replication (SMR) is implemented using Infinispan's interceptor API. This API enables the execution of custom code during the processing of a data store operation. It follows the visitor pattern as commonly found in storage systems. Infinispan [103] relies on JGroups [62] for total order multicast. The current implementation uses Skeen's algorithm [20].

In our prototype, the deployment of the storage layer is explicitly managed (like, e.g., AWS ElastiCache). Automatic provisioning of storage resources for serverless computing remains an open issue [27, 75], with just a couple works appearing very recently in this area [85, 124].

3.4 Evaluation

This section evaluates the performance of CRUCIAL and its usability to program stateful serverless applications.

Outline We first evaluate the runtime of CRUCIAL with a series of micro-benchmarks (§3.4.1). Then, we focus on fine-grained updates to shared mutable data (§3.4.2) and fine-grained synchronization (§3.4.3). Further, we detail the (partial) portage to serverless of the Smile library [92] (§3.4.4). Finally, we analyze the usability of our framework when writing (or porting) applications (§3.4.5).

Goal & scope. The core objective of this evaluation is to understand the benefits of CRUCIAL to program applications for serverless. To this end, we distinguish two types of applications: serverless-native and ported applications. Serverless-native applications are those written from scratch for a FaaS infrastructure. Ported applications are the ones that were initially single-machine applications and were later modified to execute atop FaaS. For both types of applications, our evaluation campaign aims at providing answers to the following questions:

- *How easy is it to program with* CRUCIAL*?* (§3.4.2 and §3.4.3). In addressing this question, we specifically focus on the following applications: machine learning , data analytics and synchronization tasks. These applications are parallel and stateful, that is they contain parallel components that need to update a shared state and synchronize to make progress.
- *Do applications programmed with* CRUCIAL *benefit from the capabilities of serverless (e.g., scalability and on-demand pricing)?* (§3.4.4)
- *How efficient is an application programmed with* CRUCIAL*?* (§3.4.2) For serverless-native applications, we compare CRUCIAL to PyWren, a state-of-the-art solution for serverless programming. We also make a comparison with Apache Spark, the de facto standard approach to program stateful cluster-based programs. For ported applications, we compare CRUCIAL to a scale-up approach, using a high-end server.
- *How costly is* CRUCIAL *with respect to other solutions?* (§3.4.5) Here we are interested both in the programming effort to code a serverless application and its monetary cost when running atop a FaaS platform. Again, answers are provided for both serverless-native and ported applications.

Experimental setup. All the experiments are conducted in Amazon Web Services (AWS), within a Virtual Private Cloud (VPC) located in the `us-east-1` region. Unless otherwise specified, we use `r5.2xlarge` EC2 instances for the DSO layer and 3 GB AWS Lambda functions. Experiments with concurrency over 300 cloud threads are run outside the VPC due to service limitations.

The code of the experiments presented in this section is available online [6].

Table 2: Average latency comparison – 1 KB payload

	PUT	GET
S3	34,868μs	23,072μs
Redis	232μs	229μs
Infinispan	228μs	207μs
CRUCIAL	231μs	229μs
CRUCIAL ($rf = 2$)	512μs	505μs

3.4.1 Micro-benchmarks

As depicted in Figure 2, the runtime of CRUCIAL consists of two components: a Function-as-a-Service (FaaS) platform and the DSO layer. In this section, we evaluate the performance of this runtime across several micro-benchmarks.

Latency Table 2 compares the latency to access a 1 KB object sequentially in CRUCIAL (DSO), Redis, Infinispan, and S3. We chose Redis because it is a popular key-value store available on almost all cloud platforms, and it has been extensively used as storage substrate in prior serverless systems [74, 85, 124]. Each function performs 30K operations and we report the average access latency. In Table 2, CRUCIAL exhibits a performance similar to other in-memory systems. In particular, it is an order of magnitude faster than S3. This table also depicts the effect of object replication. When data is replicated, SMR adds an extra round-trip, doubling the latency perceived at a client. The number of replicas does not affect this behavior, as shown in the next experiment.

Throughput We measure the throughput of CRUCIAL and compare it against Redis. For an accurate picture, replication is enabled in both systems to capture their performance under scenarios of high data availability and durability.

In this experiment, 200 cloud threads access 800 shared objects during 30 s. The objects are chosen at random. Each object stores an integer offering basic arithmetic operations. We consider simple and complex operations. The simple operation is a multiplication. The complex one is the sequential execution of 10K multiplications. In Redis, these operations require several commands which run as Lua scripts for both consistency and performance.

To replicate data, Redis uses a master-based mechanism. By default, replication is *asynchronous*, so the master does not wait for a command to be processed by the replicas. Consequently, clients can observe stale data. In our experiment, to minimize inconsistencies and offer guarantees closer to CRUCIAL, functions issue a `WAIT` command after each write [127]. This command flushes the pending updates to the replicas before it returns.

We compare the average throughput of the two systems when the replication factor (rf) of a datum varies as follows: ($rf = 1$) Both CRUCIAL and Redis (2 shards with no replicas) are deployed over a 2-node cluster; ($rf = 2$) In the same 2-node cluster, Redis nows uses one master and one replica; ($rf = 3$) We add a third node to the cluster and Redis employs one master and two replicas. In Figure 3, "Redis `WAIT` r" indicates that r is the number of synchronously replicated copies of shared objects.

The experimental results reported in Figure 3 show that CRUCIAL is not sensitive to the complexity of operations. Redis is 50% faster for simple operations because its implementation is optimized and written in C. However, for complex operations, CRUCIAL is almost five times better than Redis. Again, implementation-specific details are responsible for this behavior: while Redis is single-threaded, and thus concurrent calls to the Lua scripts run sequentially, CRUCIAL benefits from disjoint-access parallelism [71]. When objects are replicated, the comparison is similar. In particular, Figure 3 shows that CRUCIAL and Redis have close performance when Redis operates in synchronous mode.

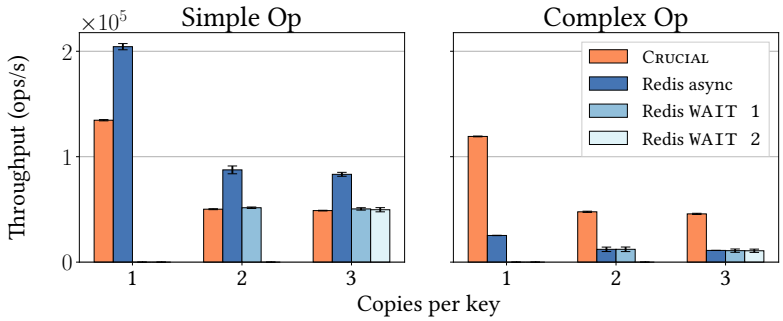


Figure 3: Operations per second performed in CRUCIAL and Redis (with and without replication). Cloud threads access uniformly at random 800 different keys/objects.

This experiment also verifies that the performance of CRUCIAL is not sensitive to the number of replicas. Indeed, the throughput in Figure 3 is roughly equivalent for all values of $rf \geq 2$. This comes from the fact that CRUCIAL requires a single RTT to propagate an operation to the replicas.

Parallelism We first evaluate our framework with the Monte Carlo simulation presented in Listing 1. This algorithm is embarrassingly parallel, relying on a single shared object (a counter). The simulation runs with 1 to 800 cloud threads and we track the total number of points computed per second. The results, presented in Figure 4a, show that our system scales linearly and that it exhibits a 512× speedup with 800 threads.

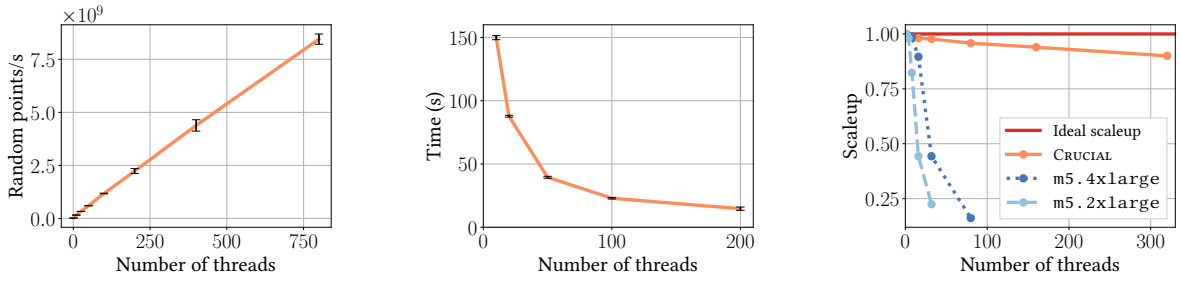
We further evaluate the parallelism of CRUCIAL with the code in Listing 3. This second experiment computes a 30K×30K projection of the Mandelbrot set, with (at most) 1000 iterations per pixel. As shown in Figure 4b, the completion time decreases from 150 s with 10 threads to 14.5 s with 200 threads: a speedup factor of 10.2× over the 10-thread execution. This super-linear speedup is due to the skew in the coarse-grained row partitioning of the image. It also underlines a key benefit of CRUCIAL. If this task is run on a cluster, the cluster is billed for the entire job duration, even if some of its resources are idle. Running atop serverless resources, this implementation ensures instead that row-dependent tasks are billed for their exact duration.

Takeaways The distributed shared objects (DSO) layer of CRUCIAL is on par with existing in-memory data stores in terms of latency and throughput. For complex operations, it significantly outperforms Redis due to data access parallelism. CRUCIAL scales linearly to hundreds of cloud threads. Applications written with the framework benefit from the serverless provisioning and billing model to match irregularities in parallel tasks.

3.4.2 Fine-grained state management

This section shows that CRUCIAL is efficient for parallel applications that access shared state at fine granularity. We detail the implementation of two machine learning algorithms in the framework. These algorithms are evaluated against a single-machine solution, as well as two state-of-the-art frameworks for cluster computing (Apache Spark) and FaaS-based computation (PyWren).

A serverless *k*-means Listing 5 details the code of a *k*-means clustering algorithm written with CRUCIAL. This program computes *k* clusters from a set of points across a fixed number of iterations, or until some convergence criterion is met (line 21). The algorithm is iterative, with recurring synchronization points (line 19), and it uses a small mutable shared state. Listing 5 relies on shared objects for the convergence criterion (line 4), the centroids (line 8), and a synchronization object to



(a) (b) (c)

Figure 4: (a) Scalability of a Monte Carlo simulation to approximate π. CRUCIAL reaches 8.4 billion random points per second with 800 threads. (b) Scalability of a Mandelbrot computation with CRUCIAL. (c) Scalability of the *k*-means clustering algorithm with CRUCIAL versus single-machine multi-threading.

```
1  public class KMeans implements Runnable {
2    private CyclicBarrier barrier = new CyclicBarrier();
3    @Shared(key = "delta")
4    private GlobalDelta globalDelta = new GlobalDelta();
5    @Shared(key = "iterations")
6    private AtomicInteger globalIterCount = new AtomicInteger();
7    // Wraps a list of @Shared centroids
8    private GlobalCentroids centroids = new GlobalCentroids();
9
10   public void run() {
11     loadDatasetFragment();
12     int iterCount = globalIterCount.intValue();
13     do {
14       correctCentroids = globalCentroids.getCorrectCoordinates();
15       resetLocalStructures();
16       localDelta = computeClusters();
17       globalDelta.update(localDelta);
18       centroids.update(localCentroids, localSizes);
19       barrier.await();
20       globalIterCount.compareAndSet(iterCount, iterCount++);
21     } while (iterCount < maxIterations && !endCondition());
22   }
23 }
```

Listing 5: *k*-means implementation with CRUCIAL.

coordinate the iterations (line 2). At each iteration, the algorithm needs to update both the centroids and the criterion. The corresponding method calls (lines 14, 17 and 18) are executed remotely in DSO.

Figure 4c compares the scalability of CRUCIAL against two EC2 instances: `m5.2xlarge` and `m5.4xlarge`, with 8 and 16 vCPUs respectively. In this experiment, the input increases proportionally to the number of threads. We measure the *scale-up* computed with respect to that fact: $\textit{scale-up} = T_1/T_n$, where T_1 is the execution time of Listing 5 with one thread, and T_n when using n threads.[4] Accordingly, $\textit{scale-up} = 1$ means a perfect linear scale-up, i.e., the increase in the number of threads keeps up with the increase in the workload size (top line in Figure 4c). The scale-up is sub-linear when $\textit{scale-up} < 1$. As expected, the single-machine solution quickly degrades when the number of threads exceeds the number of cores. The solution using CRUCIAL is within 10% of the optimum. For instance, with 160 threads, the scale-up factor is approximately 0.94. This lowers to 0.9 for 320 threads due to the overhead of creating the cloud threads.

[4]In Figure 4c, threads are AWS Lambda functions for CRUCIAL, and standard Java threads for the EC2 instances.

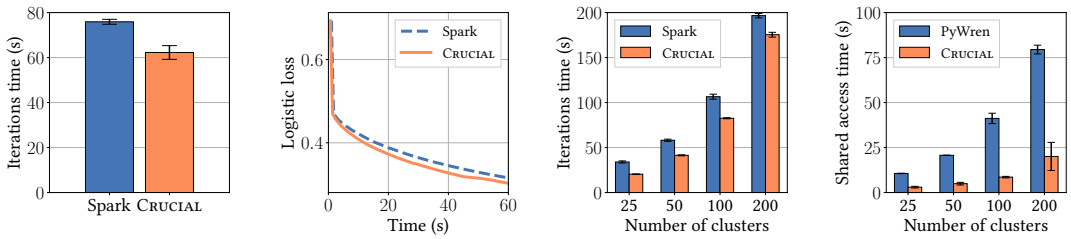


Figure 5: Comparison of CRUCIAL and the state-of-the-art. (a) Average logistic regression iterative phase completion time (100 iterations). (b) Logistic regression performance. (c) Average *k*-means iterative phase completion time (10 iterations) with varying number of clusters. (d) Average *k*-means shared state access time.

Comparison with Spark Apache Spark [160] is a state-of-the-art solution for distributed computation in a cluster. As such, it is extensively used to scale many kinds of applications in the cloud. One of them is machine learning (ML) training, as enabled by Spark's MLlib [106] library. Most ML algorithms are iterative and share a modest amount of state that requires per-iteration updates. Consequently, they are a perfect fit to assess the efficiency of fine-grained updates in CRUCIAL against a state-of-the-art solution. This is the case of logistic regression and *k*-means clustering, which we use in this section to compare CRUCIAL and Spark.

(Setup) For this comparison, we provide equivalent CPU resources to all competitors. In detail, CRUCIAL experiments are run with 80 concurrent AWS Lambda functions and one storage node. Each AWS Lambda function has 1792 MB and 2048 MB of memory for logistic regression and *k*-means, respectively. These values are chosen to have the optimal performance at the lowest cost (see §3.4.5).[5] The DSO layer runs on a `r5.2xlarge` EC2 instance. Spark experiments are run in Amazon EMR with 1 master node and 10 `m5.2xlarge` worker nodes (*Core nodes* in EMR terminology), each having 8 vCPUs. Spark executors are configured to utilize the maximum resources possible on each node of the cluster. To improve the fairness of our comparison, the time spent in loading the dataset from S3 and parsing it is not considered for both solutions. For Spark, the time to provision the cluster is not counted. Regarding CRUCIAL, FaaS cold starts are also excluded from measurements due to a global barrier before starting the computation.

(Dataset) The input is a 100 GB dataset generated with spark-perf [37] that contains 55.6M elements. For logistic regression, each element is labeled and contains 100 numeric features. For *k*-means, each element corresponds to a 100-dimensional point. The dataset has been split into 80 equal-size partitions to ensure that all partitions are small enough to fit into the function memory. Each partition has been stored as an independent file in Amazon S3.

(Logistic regression) We evaluate a CRUCIAL implementation of logistic regression against its counterpart available in Spark's MLlib [106]: `LogisticRegressionWithSGD`. A key difference between the two implementations is the management of the shared state. Each iteration, Spark broadcasts the current weight coefficients, computes, and finally aggregates the sub-gradients in a MapReduce phase. In CRUCIAL, the weight coefficients are shared objects. Each iteration, a cloud thread retrieves the current weights, computes the sub-gradients, updates the shared objects, and synchronizes with the other threads. Once all the partial results are uploaded to the DSO layer, the weights are recomputed, and the threads proceed to the next iteration.

In Figures 5a and 5b, we measure the running time of 100 iterations of the algorithm and the logistic loss after each iteration. Results show that the iterative phase is 18% faster in CRUCIAL

[5] Starting with a configuration of 1792 MB, an AWS Lambda function has the equivalent to 1 full vCPU (`https://docs.aws.amazon.com/lambda/latest/dg/resource-model.html`). Also, with this assigned memory, the function uses a full Elastic Network Interface (ENI) in the VPC.

(62.3 s) than with Spark (75.9 s), and thus the algorithm converges faster. This gain is explained by the fact that CRUCIAL aggregates and combines the sub-gradients in the storage layer. On the contrary, each iteration in Spark requires a reduce phase that is costly both in terms of communication and synchronization.

(k-means) We compare the *k*-means implementation described in §3.4.2 to the one in MLlib. For both systems, the centroids are initially at random positions and the input data is evenly distributed among tasks. Figure 5c shows the completion time of 10 iterations of the clustering algorithm. In this figure, we consider different values of k to assess the effectiveness of our solution when the size of the shared state varies. With $k = 25$, CRUCIAL completes the 10 iterations 40% faster (20.4 s) than Spark (34 s). The time gap is less noticeable with more clusters because the time spent synchronizing functions is less representative. In other words, the iteration time becomes increasingly dominated by computation. As in the logistic regression experiment, CRUCIAL benefits from computing centroids in the DSO layer, while Spark requires an expensive reduce phase at each iteration.

Comparison with PyWren We close this section by comparing CRUCIAL to a serverless-native state-of-the-art solution. To date, the most evaluated framework to program stateful serverless applications is PyWren [74]. Its primitives, such as `call_async` and `map` are comparable to CRUCIAL's cloud thread and serverless executor abstractions. Our evaluation employs Lithops, a recent and improved version of PyWren (see [122] for the full details). PyWren is a MapReduce framework. Thus, it does not natively provide advanced features for state sharing and synchronization. Therefore, following the recommendations by Jonas et al. [74], we use Redis for this task.

(Setup) We employ the same application, dataset, and configuration as in the previous experiment. The two frameworks use AWS Lambda for execution. A single `r5.2xlarge` EC2 instance runs DSO for CRUCIAL, or Redis for PyWren.

(k-means) Implementing *k*-means above PyWren requires to store the shared state in Redis, that is the centroids and the convergence criterion. Following Jonas et al. [74], we use a Lua script to achieve this. At the end of each iteration, every function updates (atomically) the shared state by calling the script. This approach is the best solution in terms of performance. In particular, it is more efficient than using distributed locking due to the large number of commands needed for the updates. To synchronize across iterations, we use the Redis barrier covered in §3.4.3.

The CRUCIAL and PyWren *k*-means applications are written in different languages (Java and Python, respectively). Consequently, the time spent in computation for the two applications is dissimilar. For that reason, and contrary to the comparison against Spark, Figure 5d does not report the completion time. Instead, this figure depicts the average time spent in accessing the shared state during the *k*-means execution for both CRUCIAL and PyWren. This corresponds to the time spent inside the loop in Listing 5 (excluding line 16).

In Figure 5d, we observe that the solution combining PyWren and Redis is always slower than CRUCIAL. This comes from the fact that CRUCIAL allows efficient fine-grained updates to the shared state. Such results are in line with the ones presented in §3.4.1.

Takeaways The distributed shared objects (DSO) layer of CRUCIAL offers abstractions to program stateful serverless applications. DSO is not only convenient but, as our evaluation confirms, efficient. For two common machine learning tasks, CRUCIAL is up to 40% faster than Spark, a state-of-the-art cluster-based approach, at comparable resource usage. It is also faster than a solution using jointly PyWren, a well-known serverless framework, and the Redis data store.

3.4.3 Fine-grained synchronization

This section analyzes the capabilities of CRUCIAL to coordinate cloud functions. We evaluate the synchronization primitives available in the framework and compare them to state-of-the-art solutions. We then demonstrate the use of CRUCIAL to solve complex coordination tasks by considering a traditional concurrent programming problem.

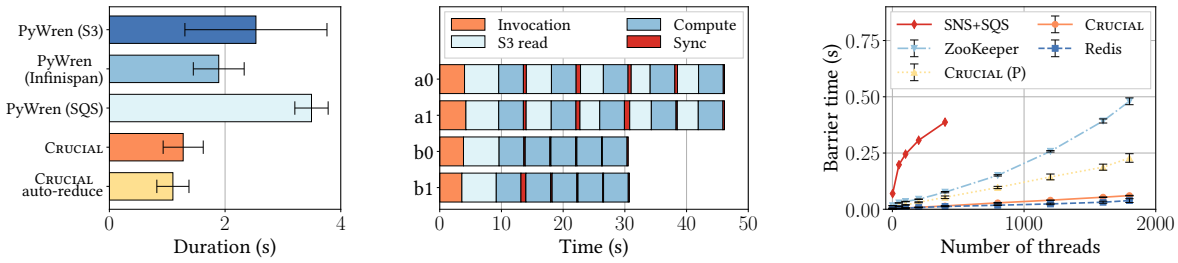


Figure 6: (a) Synchronizing a map phase in MapReduce with PyWren, Amazon SQS and CRUCIAL. (b) Performance breakdown of an iterative task using either multiple stages (a0/a1), or a single stage with a CRUCIAL barrier (b0/b1). (c) Average time threads spend waiting on a barrier.

Map phase Many algorithms require synchronization at various stages. In MapReduce [38], this happens between the map and reduce phases, and it is known as shuffle. Shuffling ensures that the reduce phase starts when all the appropriate data was output in the preceding map phase. Shuffling the map output is a costly operation in MapReduce, even if the reduce phase is short. For that reason, when data is small and the reduction operation simple, it is better to skip the reduce phase and instead aggregate the map output directly in the storage layer [40]. CRUCIAL allows to easily implement this approach.

In what follows, we compare different techniques to synchronize cloud functions at the end of a map. Namely, we compare 1. the original solution in PyWren, based on polling S3; 2. the same mechanism but using the Infinispan in-memory key-value data store; 3. the use of Amazon SQS, as proposed in some recent works (e.g., [84]); and 4. two techniques based on the `Future` object available in CRUCIAL. The first solution outputs a future object per function, then runs the reduce phase. The second aggregates all the results directly in the DSO layer (called auto-reduce).

We compare the above five techniques by running back-to-back the Monte Carlo simulation in Listing 1. The experiment employs 100 cloud functions, each doing 100M iterations. During a run, we measure the time spent in synchronizing the functions. On average, this accounts for 23% of the total time.

Figure 6a presents the results of our comparison. Using Amazon S3 proves to be slow, and it exhibits high variability —some experiments being far slower than others. This is explained by the combination of high access latency, eventual consistency, and the polling-based mechanism. The results improve with Infinispan, but being still based on polling, the approach induces a noticeable overhead. Using Amazon SQS is the slowest approach of all. It needs a polling mechanism that actively reads messages from the remote queue. The solution based on `Future` objects allows to immediately respond when the results are available. This reduces the number of connections necessary to fetch the result and thus translates into faster synchronization. When the map output is directly aggregated in DSO, CRUCIAL achieves even better performance, being twice as fast as the polling approach atop S3.

Synchronization primitives Cloud functions need to coordinate when executing parallel tasks. This section evaluates some of the synchronization primitives available in CRUCIAL to this end.

For starters, we study the performance of a barrier when executing an iterative task. In Figure 6b, we depict a breakdown of the time spent in the phases of each iteration (Invocation, S3 read, Compute, and Sync). The results are reported for 2 cloud functions out of 10 —the other functions behave similarly.

The breakdown in Figure 6b considers two approaches. The first one launches a new stage of functions (a0 and a1) at each iteration that do not use the barrier primitive. The second launches

a single stage of functions (b0 and b1) that run all the iterations and use the barrier primitive to synchronize. In the first case, data must be fetched from storage at each iteration, while in the second approach it is only fetched once. Overall, Figure 6b shows that this latter mechanism is clearly faster. In particular, the total time spent in coordinating the functions is lower when the barrier is used (Sync).

Figure 6c draws a comparison between two barrier objects available in CRUCIAL and several state-of-the-art solutions. More precisely, the figure reports the performance of the following approaches: 1. a pure cloud-based barrier, which combines Amazon SNS and SQS services to notify the functions; 2. a ZooKeeper cyclic barrier based on the official double barrier [51] in a 3-node cluster; 3. a non-resilient barrier using the Redis `BLPOP` command ("blocking left pop") on a single server; 4. the default cyclic barrier available in CRUCIAL, with a single server instance; and 5. a resilient, poll-based (P) barrier implementing the algorithm in [65] on a 3-node cluster with replication.

To draw this comparison, we measure the time needed to exit 1000 barriers back-to-back for each approach. An experiment is run 10 times. Figure 6c reports the average time to cross a single barrier for a varying number of cloud functions.

The results in Figure 6c show that the single server solutions, namely CRUCIAL and Redis, are the fastest approaches. With 1800 threads, these barriers are passed after waiting 68 ms on average. The fault-tolerant barriers (CRUCIAL (P) and ZooKeeper) create more contention, incurring a performance penalty when the level of parallelism increases. With the same number of threads, passing the poll-based barrier of CRUCIAL takes 287 ms on average. ZooKeeper requires twice that time. The solution using Amazon SNS and SQS is an order of magnitude slower than the rest.

It is worth noting the difference between the programming complexity of each barrier. Both barriers implemented in CRUCIAL take around 30 lines of basic Java code. The solution using Redis has the same length, but it requires a proper management of the connections to the data store as well as the manual creation/deletion of shared keys. ZooKeeper substantially increases code complexity, as programmers need to deal with a file-system-like interface and carefully set watches, requiring around 90 lines of code. Finally, the SNS and SQS approach is the most involved technique of all, necessitating 150 lines of code and the use of two complex cloud service APIs.

A concurrency problem Thanks to its coordination capabilities, CRUCIAL can be used to solve complex concurrency problems. To demonstrate this feature, we consider the Santa Claus problem [150]. This problem is a concurrent programming exercise in the vein of the dining philosophers, where processes need to coordinate in order to make progress. Common solutions employ semaphores and barriers, while others, actors [18].

(Problem) The Santa Claus problem involves three sets of *entities*: Santa Claus, nine reindeer and a group of elves. The elves work at the workshop until they encounter an issue that needs Santa's attention. The reindeer are on vacation until Christmas eve, when they gather at the stable. Santa Claus sleeps, and can only be awakened by either a group of three elves to solve a workshop issue, or by the reindeer to go delivering presents. In the first case, Santa solves the issues, and the elves go back to work. In the second, Santa and the reindeer execute the delivery. The reindeer have priority if the two situations above occur concurrently.

(Solution) Let us now explain the design of a common solution to this problem [18]. Each entity (Santa, elves, and reindeer) is a thread. They communicate using two types of synchronization primitives: *groups* and *gates*. Elves and reindeer try to join a group when they encounter a problem or Christmas is coming, respectively. When a group is full —either including three elves or nine reindeer—, the entities enter a room and notify Santa. A room has two gate objects: one for entering and one for exiting. Gates act like barriers, and all the entities in the group wait for Santa to open the gate. When Santa is notified, he looks whether a group is full (either of reindeer or elves, prioritizing reindeer). He then opens the gate and solves the workshop issues or goes delivering presents. This last operation is repeated until enough deliveries, or *epochs*, have occurred.

We implemented the above solution in three flavors. The first one uses plain old Java objects (POJOs), where groups and gates are monitors and the entities are threads. Our second variation is a

Table 3: Santa Claus problem's completion time (in seconds) on a single machine vs. CRUCIAL.

	Threads	Threads + DSO	CRUCIAL
p50	20.15	20.91	21.97
p99	21.09	22.03	22.66
Overhead	–	3.8%	9.0%

refinement of this base approach, where the synchronization objects are stored in the DSO layer. The conversion is straightforward using the API presented in §3.1. In particular, the code of the objects used in the POJO solution is unchanged. Only adding the `@Shared` annotation is required. The last refinement consists in using CRUCIAL's cloud threads instead of the Java ones.

(Evaluation) We consider an instance of the problem with 10 elves, 9 reindeer and 15 *deliveries* (epochs of the problem). Table 3 presents the completion time for each of the above solutions.

The results in Table 3 show that CRUCIAL is efficient in solving the Santa Claus problem, being at most 9% slower than a single-machine solution. In detail, storing the group and gate objects in CRUCIAL induces an overhead of around 4% on the completion time. When cloud threads are used instead of Java ones, a small extra time is further needed —less than a second. This penalty comes from the necessary remote calls to the FaaS platform to start computation.

Takeaways The fine-grained synchronization capabilities of CRUCIAL permit cloud functions to coordinate efficiently. The synchronization primitives available in the framework fit iterative tasks well and perform better than state-of-the-art solutions at large scale while being simpler to use. This allows CRUCIAL to solve complex concurrency problems efficiently.

3.4.4 Smile library

The previous section presented the portage to serverless of a solution to the Santa Claus problem. In what follows, we further push this logic by considering a complex single-machine program. In detail, we report on the portage to serverless of the random forest classification algorithm available in the Smile library. Smile [92] is a multi-threaded library for machine learning, similar to Weka [67]. It is widely employed to mine datasets with Java and contains around 165K SLOC. In what follows, we first describe the steps that were taken to conduct the portage using CRUCIAL. Then, we present performance results against the vanilla version of the library.

Porting `smile.classification.RandomForest` The portage consists in adapting the random forest classification algorithm [23] with the help of our framework. In the training phase, this algorithm takes as input a structured file (commonly, `.csv` or `.arff`) which contains the dataset description. It outputs a random forest, i.e., a set of decision trees. During the classification phase, the forest is used to predict the class of the input items. Each decision tree is calculated by a training task (`Callable`). The tasks are run in parallel on a multi-core machine during the training phase. During this computation, the algorithm also extracts the out-of-bag (OOB) precision, that is the forest's error rate induced by the training dataset.

To perform the portage, we take the following three steps. First, a proxy is added to stream input files from a remote object store (e.g., Amazon S3). This proxy lazily extracts the content of the file, and it is passed to each training task at the time of its creation. Second, the training tasks are instantiated in the FaaS platform. With CRUCIAL, this transformation simply requires calling a `ServerlessExecutorService` object in lieu of the Java `ExecutorService`. Third, the shared-memory matrix that holds the OOB precision is replaced with a DSO object. This step requires to change the initial programming pattern of the library. Indeed, in the original application, the `RandomForest` class creates a matrix using the metadata available in the input file (e.g., the number of features). If this

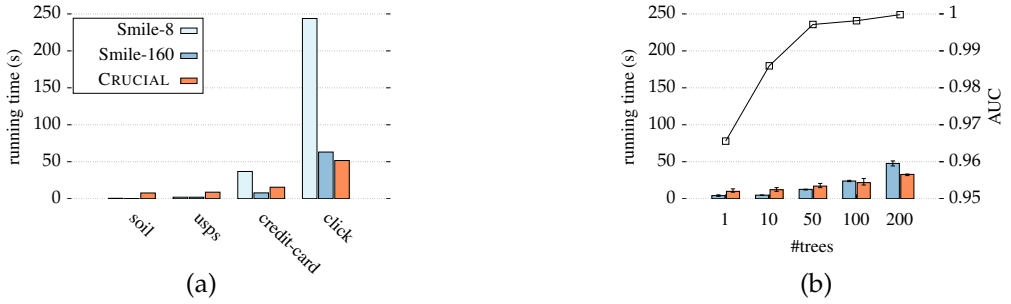


Figure 7: Smile portage. (a) Performance per dataset using 50 trees. (b) Varying the number of trees for the credit-card dataset [121].

pattern is kept, the application must load the input file to kick off the parallel computation, which is clearly inefficient. In the portage, we instead use a barrier to synchronize the concurrent tasks. The first task to enter the barrier is in charge of creating the matrix in the DSO layer.[6]

For performance reasons, Smile uses Java arrays (mono or multi-dimensional) and not object-oriented constructs (such as `ArrayList`). As pointed out previously in §3.1.3, it is not possible to build proxies for such objects in Java without changing the bytecode generated during compilation. Thus, the portage necessitates to transform these arrays into high-level objects. These objects are then replaced with their CRUCIAL counterparts.

Overall, the portage modifies 378 SLOC in the Smile library (version 1.5.3). This is less than 4% of the original code base to run the random forest algorithm. We also added scripts to deploy and run the serverless solution in AWS Lambda, and performance benchmarks (see below), for a total of around 1K SLOC. Notice that the portage does not preclude local (in-memory) execution, e.g., for testing purpose. This is possible by switching a flag at runtime.

Evaluation results In Figure 7, we compare the vanilla version of Smile to our CRUCIAL portage. To this end, we use 4 datasets: (*soil*) is built using features extracted from satellite observations to categorize soils [59]; (*usps*) was published by Le Cun et al. [104] and it contains normalized handwritten digits scanned from envelopes by the U.S. Postal Service; (*credit-card*) is a set of both valid and fraudulent credit card transactions [121]; (*click*) is a 1% balanced subset of the KDD 2012 challenge (Task 2) [69].

We report the performance of each solution during the learning phase. As previously, CRUCIAL is executed atop AWS Lambda. The DSO layer runs with $rf = 2$ in a 3-node (4 vCPU, 16 GB of RAM) Kubernetes cluster. For the vanilla version of Smile, we use two different setups: an hyperthreaded quad-core Intel i78550U laptop with 16 GB of memory (tagged Smile-8 in Figure 7), and a quad-Intel CLX 6230 hyperthreaded 80-core server with 740 GB of memory (tagged Smile-160 in Figure 7).[7]

As expected for small datasets (*soil* and *usps*), the cost of invocation out-weights the benefits of running over the serverless infrastructure. For the two large datasets, Figure 7a shows that the CRUCIAL portage is up to 5x faster. Interestingly, for the last dataset the performance is 20% faster than with the high-end server.

In Figure 7b, we scale the number of trees in the random forest, from a single tree to 200. The second y-axis of this figure indicates the area under the curve (AUC) that captures the algorithm's accuracy. This value is the average obtained after running a 10-fold cross-validation with the training dataset. In Figure 7b, we observe that the time to compute the random forest triples from around 10 to 30 s. Scaling the number of trees helps improving classification. With 200 trees, the AUC of the

[6] This pattern is reminiscent of a Phaser object in Java.

[7] In this last case, the JVM executes with additional flags (+XX:+UseNUMA -XX:+UseG1GC) to leverage the underlying hardware architecture.

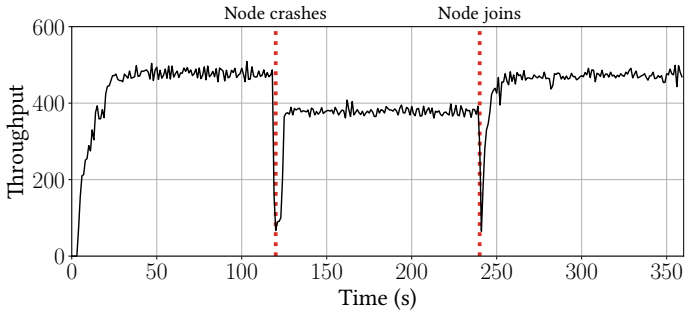


Figure 8: Inferences per second performed on a *k*-means model during 6 minutes. Up to 100 concurrent FaaS functions connecting to the shared model on up to 3 DSO nodes with $rf = 2$. Note the FaaS cold start at the beginning.

computed random forest is 0.9998. This result is in line with prior reported measures [121] and it indicates a strong accuracy of the classifier. Figure 7b indicates that training a 200-trees forest takes around 30 s with CRUCIAL. This computation is around 50% slower with the 160-threads server. It takes 20 minutes on the laptop test machine (not shown in Figure 7b).

Takeaways Overall, the above results show that the portage is efficient, bringing elasticity and on-demand capabilities to a traditional monolithic multi-threaded library. We focused on the random forest classification algorithm in Smile, but other algorithms in this library can be ported to FaaS with the help of CRUCIAL.

3.4.5 Usability of CRUCIAL

This section evaluates how CRUCIAL simplifies the writing of stateful serverless applications and their deployment and management in the cloud.

Data availability Our first experiment assesses that CRUCIAL indeed offers high availability to data persisted in the DSO layer. To this end, the membership of DSO is changed during the execution of the serverless *k*-means. Figure 8 shows a 6-minute run during which inferences are executed with the model trained in §3.4.2. The model is stored in a cluster of 3 nodes with $rf = 2$. The inferences are performed using 100 cloud threads. Each inference executes a read of all the objects in the model, i.e., the 200 centroids.

During the experiment, at 120 s and 240 s, we crash and add, respectively, a storage node to the DSO layer. Figure 8 shows that our system is elastic and resilient to such changes. Indeed, modifications to the membership of the DSO layer affect performance but never block the system. The (abrupt) removal of a node lowers performance by 30%. The initial throughput of the system (490 inferences per second) is restored 20 s after a new storage node is added.

Notice that handling catastrophic (or cascading) events is possible by running DSO across several availability zones, or even datacenters. In such cases, SMR can be tailored to accommodate with the increased latency between data replicas [108]. Evaluating these geo-distributed scenarios is however outside of the scope of this paper.

Programming simplicity Each of the applications used in the evaluation is initially a single-machine program. Table 4 lists the modifications that were necessary to move each program to serverless with CRUCIAL. The differences between the single-machine, parallel code and its serverless counterpart

[8] Transitive closure of the dependencies of `smile.classification.RandomForest` in the Smile library.

Table 4: Lines of code changed in each application to move it to FaaS with CRUCIAL.

Application	Total lines	Changed lines	
Monte Carlo	44	2	(4.5%)
Mandelbrot	88	3	(3.4%)
Logistic regression	430	10	(2.3%)
k-means	329	8	(2.4%)
Santa Claus problem	255	15	(5.9%)
Random Forest [8]	9882	378	(3.8%)

Table 5: Monetary costs of the experiments

		Total time (s)	Total cost ($)	Iterations cost ($)
Logistic regression	Spark	192	0.282	0.111
	CRUCIAL	122	0.302	0.154
k-means ($k = 25$)	Spark	168	0.246	0.050
	CRUCIAL	87	0.244	0.057
k-means ($k = 200$)	Spark	330	0.484	0.288
	CRUCIAL	234	0.657	0.492

are small. In the case of Smile, as mentioned earlier, they mainly come from the use of low-level non-OOP constructs in the library (e.g., Java arrays). For the other programs, e.g., the logistic regression algorithm detailed in §3.4.2, the changes account for less than 3%.

Starting from a conventional OOP program, CRUCIAL requires only a handful of changes to port it to FaaS. We believe that this smooth transitioning can help everyday programmers to start harvesting the benefits of serverless computing.

Cost comparison Although one may argue that the programming simplicity of serverless computing justifies its higher cost [74], running an application serverless should not significantly exceed the cost of running it with other cloud appliances (e.g., VMs).

Table 5 offers a cost comparison between Spark and CRUCIAL based on the experiments in §3.4.2. The first two columns list the time and cost of the entire experiments, including the time spent in loading and parsing input data, but not the resource provisioning time. The last column lists the cost that can be attributed to the iterative phase of each algorithm. To compare fairly the two approaches, we only consider the pricing for on-demand instances.

With the current pricing policy of AWS [10], the cost per second of CRUCIAL is always higher than Spark: 0.25 and 0.28 cents per second for 1792 MB and 2048 MB function memory, respectively, against 0.15 cents per second. Thus, when computation dominates the running time, as in *k*-means clustering with $k = 200$, the cost of using CRUCIAL is logically higher. This difference disappears when CRUCIAL is substantially faster than Spark (e.g., $k = 25$).

To give a complete picture of this cost comparison, there are two additional remarks to make here. First, the solution provided with CRUCIAL using 80 concurrent AWS Lambda functions employs a larger aggregated bandwidth from S3 than the solution with Spark. This reduces the cost difference between the two approaches. Second, as pointed in §3.4.1, CRUCIAL users only need to pay for the execution time of each function, and not the time the cluster remains active. This includes bootstrapping the cluster as well as the necessary trial-and-error processes found, for instance, in machine learning training or hyper-parameter tuning [154].[9]

[9] Provisioning the 11-machine EMR cluster takes 2 minutes (not billed) and bootstrapping requires an extra 4 minutes. A DSO node starts in 30 seconds.

Takeaways The programming model of CRUCIAL allows to easily write conventional object-oriented applications for serverless. Starting from a single-machine code, the changes are minimal. In particular, the distributed shared objects (DSO) layer offers the exact same semantics for state sharing and synchronization as a conventional multi-threaded library (e.g., *java.util.concurrent*). Being serverless, applications written with CRUCIAL are scalable. Moreover, they execute at a comparable cost than cluster-based solutions without high upfront investment.

4 The Serverless Shell

With the advent of new computing platforms, it is of interest to port legacy software. A successful portage allows to reuse a code base and benefits from technological advances. One of these software is the traditional Unix shell [114, 128]. Much of the success of *NIX operating systems comes from this command-line interpreter. The Unix shell offers a minimalist yet powerful syntax to write programs. Shell scripting covers many usages, from traditional system administrative tasks to data analytics. It is particularly efficient to manipulate files, query datasets and execute basic repetitive tasks.

In this section, we detail our adaptation of the Unix shell for serverless using the CRUCIAL framework. The new tool, named the serverless shell (`sshell`), allows to execute commands remotely in the serverless platform. Each command runs isolated in the platform as a serverless function. `sshell` leverages the simplicity and scalability of serverless computing to give access to large computing power on-demand and in a pay-per-use manner. It is built around a small set of components that includes a new inter-process communication layer for serverless. This layer allows to communicate through synchronization primitives (pipes, rendez-vous) and share data (counters, maps, arrays) at fine grain. Coupled with a remote storage, shell scripts can conveniently leverage the inherent parallelism of serverless computing to process large amount of data.

We evaluate `sshell` with micro-benchmarks and a large-scale application. `sshell` requires minimal efforts to port a legacy script to serverless. Once adapted, a script has comparable or better performance than with a high-end server. It is also up to an order of magnitude more cost-efficient and faster than a cluster-based solution to mine large datasets.

In the follow-up, we explain how to program with `sshell` (§4.1) then present its internals (§4.2). The details of our implementation using the CRUCIAL frameowrk follow (§4.2.6). Further, we evaluate `sshell` in AWS Lambda (§4.3).

4.1 Programming with `sshell`

`sshell` is a program to run shell commands in a serverless platform. To invoke it, the programmer types `sshell` followed by the command to execute. For instance, the code below lists the directories available under the root:

```
1 $> sshell ls -C /
2 RequestId: d6215a3a-a41c-4384-b779-215cfa06b30c
3 Duration: 13 ms Memory Used: 98 MB
4 bin dev home lib64 mnt proc run srv tmp var
5 boot etc lib media opt root sbin sys usr
```

The first two lines of the output (in light gray) are debugging information. They indicate respectively the identifier of the call in the serverless platform, and the time and memory spent to answer it. The directories available under / are then listed in the next two lines. This call was executed on AWS Lambda and thus this corresponds to the content of the Firecracker container where the command was run [7].

Figure 9 details a more complex example. `sshell` is used to compute the average size of a web page by mining part of the Common Crawl dataset produced in April 2021. Calling this script leads to the following result:

```
1 $> average
2 50778
```

```
1 TMP_DIR=/tmp/$(whoami)
2 CCBASE="http://commoncrawl.s3.amazonaws.com"
3 CCMAIN="CC-MAIN-2021-17" # april 2021
4 RANGE="-r 0-10000000"
5 curl -s ${CCBASE}/crawl-data/${CCMAIN}/warc.paths.gz \
6   | zcat | head -n 1000 > ${TMP_DIR}/index
7 average(){
8   while read l; do
9   sshell "curl -s ${RANGE} ${CCBASE}/${l} | 2>/dev/null zcat -q | grep ^Content-Length " &
10  done < ${TMP_DIR}/index | awk '{ sum += $2 } END { if (NR > 0) print int(sum / NR) }'
11 }
```

Figure 9: A script using `sshell`.

In detail, lines 2-3 locate the crawl of interest on the web server. The index of the crawl is then fetched and stored in a temporary file. Each line of the index is an archive of the web pages crawled by the Common Crawl foundation. Function `average` iterates over this index (line 8) and for each archive invokes a `sshell` command in parallel (line 9). The command decompresses the archive and returns the size of the crawled pages using their HTTP headers. These values are then averaged using `awk`.

4.2 System design

This section explains how `sshell` works. We first provide an overview then cover its components in detail.

4.2.1 Overview

`sshell` is built around four components: a serverless platform, an executor, a distributed storage and an inter-process communication layer. The serverless platform executes on-demand isolated functions over a set of distributed machines. Examples of such platforms include AWS Lambda, Google Cloud Functions, OpenFaaS and KNative.

When `sshell` is called with a command at some client machine, it invokes the executor in the serverless platform. The executor is in charge of executing the command and redirecting its output to the client. The executor can access any form of distributed storage reachable from the serverless platform. This includes distributed file systems, key-value stores and remote web resources.

The inter-process communication (IPC) layer allows fine-grained communication and synchronization between `sshell` invocations. Roughly speaking, this layer is equivalent to the common IPC layer available in the operating system. It supports the standard pipe operator, and provides synchronization primitives as well as linearizable shared objects.

4.2.2 Serverless platform

A serverless platform allows to execute user-defined functions at scale in a distributed system. The platform consists of several workers in charge of executing the functions and a scheduler to orchestrate them.

A function targets a specific runtime, e.g., Python, and it follows a well-defined signature, such as "`out f(context,in)`". To make a function executable in the platform, the user first uploads it. Once installed, the function is triggered on-demand by contacting the scheduler. Serverless platforms generally offer two types of invocation: *synchronous* and *asynchronous*. If the output is returned to the client once the functions completes, the invocation is synchronous. The invocation is asynchronous when the output is ignored by the client (e.g., if the function returns `null`).

Functions are stateless, that is they do not keep state from one invocation to another. This comes from the fact that multiple invocations might not hit the same worker and can occur in parallel.

```
1 gathering(){
2   JOBS=100
3   BARRIER=$(uuid)
4   seq 1 1 $((JOBS-1)) | parallel -n0 sshell --async barrier -n ${BARRIER} await -p ${JOBS}
5   sshell barrier -n ${BARRIER} await -p ${JOBS}
6 }
```

Figure 10: Synchronization on a barrier.

Invocation are isolated one from another. To achieve this, serverless platforms usually run functions in a dedicated container or VM.

4.2.3 Executor

The executor is a serverless function installed in the serverless platform. This function takes as input a shell command to execute. At line 9 in Figure 9, the command starts with "`curl -s ${RANGE}..`". `sshell` can also pass a file to the executor using the parameter `-f FILE`.

Once called, the executor spawns a process to execute the command. The standard output (stdout) of this process is redirected to the stdout of `sshell`. Similarly, the standard error (stderr) of the process is the stderr of `sshell`.

By default, the executor is invoked synchronously in the serverless platform. The content of stdout and stderr is sent back to the client once the executor terminates. It is also possible to run the executor in asynchronous mode, using `-async`, and fully ignore both streams.

4.2.4 Distributed storage

Like other serverless functions, `sshell` may use any form of distributed storage accessible from the platform. In particular, it can use cloud storage services using a dedicated client (or simply REST requests), as well as the remote resources available on the web.

In the following, we consider that the executor has access to such a remote distributed storage. Access is made through the file system using the mount/unmount interface. As an example, in our evaluation (§4.3), `sshell` uses the AWS Elastic File Storage (EFS) and AWS Simple Storage Service (S3) to read and write data files.

4.2.5 Inter-process communication

Within an operating system, processes communicate through the inter-process communication (IPC) layer. In particular, shell scripts rely heavily on three mechanisms to communicate: files (`>FILE` , `<FILE`), named pipes (`mkfifo FILE`) and anonymous pipes (`|`).

Files and named pipes are available to `sshell` invocations using the distributed storage mounted in the executor. In addition, `sshell` also offers means to communicate via a new IPC layer for serverless. This layer is similar to the standard IPC layer as found in an operating system. It consists of three communication mechanisms: synchronization primitives, linearizable objects and anonymous pipes. We detail each of these mechanisms in the following.

Synchronization primitives Multiple `sshell` invocations can synchronize using primitives such as (cyclic) barriers, countdown latches and rendez-vous. Calling a primitive abides by the following syntax:

```
1   $> primitive -n name operation [parameters]
```

where `primitive` is the type of the primitive, `name` denotes the name of the instance and `operation` the operation to invoke. The call parameters depend on the operation. For instance, a call to `await`

```
1 average_stateful(){
2   sshell "counter -n average reset"
3   while read l; do
4   sshell "counter -n average increment -i \$(curl -s ${RANGE} ${CCBASE}/${l} | 2>/dev/null zcat | grep
        ^Content-Length | awk '{ sum += \$2 } END { if (NR > 0) print int(sum / NR) }')" 1> /dev/null &
5   done < ${TMP_DIR}/index
6   wait
7   lines=$(wc -l ${TMP_DIR}/index | awk '{print $1}')
8   total_average=$(sshell "counter -n average tally")
9   echo $((total_average/lines))
10 }
```

Figure 11: Distributed stateful computation.

(lines 5-6 in Figure 10) requires the number of participants (`-p $JOBS`). In this figure, function `gathering` first generates a random name for the barrier (line 4). Then, it invokes `sshell` 99 times, and each invocation synchronizes on the barrier. Further, `gathering` also joins the barrier, and once the barrier is lifted, it returns. In this example, `sshell` is invoked using GNU `parallel` [146] and asynchronously (as no result is expected).

Shared objects `sshell` also includes a library of shared objects that may be called within a command. The signature of a call follows the same convention as the one previously described for the synchronization primitives.

The library provides common data structures (hash tables, red-black trees, lists, arrays and counters). As detailed in §4.2.6, it is built atop DSO. This library is easily extensible if needed. These objects are linearizable when called concurrently by multiple `sshell` instances, that is they behave as if a single instance had executed all the calls. This is common in concurrent programming and simplifies the writing of parallel programs.

We illustrate how to use the shared objects library in Figure 11. This script is semantically equivalent to the one in Figure 9. It also averages the sizes of the web pages in the Common Crawl dataset. Differently from Figure 9, the computation is however executed this time in a stateful manner. To this end, each `sshell` command uses a counter named `average` (line 4 in Figure 11). This counter is shared among all the `sshell` invocations that run in parallel. Upon reading a chunk of the dataset, an invocation takes the average size of the web pages it encounters in the chunk, then increments the counter accordingly. At the end of the computation, the script divides the value of the counter by the number of lines in the index to obtain the global average.

Anonymous pipes `sshell` supports anonymous pipes with the help of a rewriting tool. Depending on the capabilities of the serverless platform, the `|` operator is rewritten in two different ways.

If the serverless platform allows direct communication between functions, a socket is used. We illustrate this case in Figure 12(top), where we rewrite `cmdA|cmdB`. In this figure, `cmdB` creates a socket at port 8080 to receive the output of `cmdA` (line 1). Then, it adds its address to a rendez-vous object in the IPC layer. The name of this object is chosen randomly (`de41a38e` in Figure 12). `cmdA` awaits for this information, then connects to the socket at the provided address (line 2).

In case direct communication between functions is forbidden, an anonymous pipe is implemented through the distributed storage layer using the file system interface. This is illustrated in Figure 12(bot). The rewriting tool first creates a shared file with a random name in the file system. Then, it rewrites `cmdA` (line 1) and `cmdB` (line 2) appropriately using `tail` and `sed`. In particular, the rewriting uses a magic (`EOF`) to signal the end of the output of `cmdA` in the shared file.

4.2.6 Implementation

`sshell` is written in Java atop the CRUCIAL framework. The sources cover around 3K SLOC. This code base is open and available online [98]. The software is packaged with Maven and includes a na-

```
1 sshell "nc -N -l 8080 | cmdB& rdv de41a38e -1 $IP" &
2 sshell "HOST=$(rdv de41a38e); exec 3<>/dev/tcp/${HOST}/8080; cmdA >&3; echo EOF >&3"
```

```
1 sshell "cmdA | awk '{print \$0}END{print \"EOF\"}' > /fs/de41a38e" &
2 sshell "tail -n +0 --pid=\$\$ -f --retry /fs/de41a38e 2>/dev/null | { sed \"/EOF/ q\" && kill \$\$ ;} |
      grep -v ^EOF\$ | cmdB"
```

Figure 12: Rewriting `cmdA|cmdB` using (top) direct communication, or (bot) the file system.

```
1 seq 1 1 1000 | parallel -j$JOBS -I,, "sshell \"sleep 10\""
2 seq 1 1 1000 | parallel -j$JOBS -I,, "sshell \"dd if=/dev/zero
      of=/efs/large-file-100mb-,,-\${PARALLEL_SEQ}.txt count=1024 bs=102400\""
3 ls /efs | parallel -j$JOBS -I,, "sshell \"echo ,, ; cat /efs/,, /dev/null\""
```

Figure 13: One-liners used in the evaluation.

tive x86-64 executable built with GraalVM (19.3.0). A native executable is key for performance when multiple `sshell` commands are invoked in parallel. The current distribution includes deployment scripts for AWS Lambda. Porting `sshell` to another serverless platform requires to rewrite these scripts, as well as a handful of Java functions in the executor.

The distributed storage layer is implemented using AWS Elastic File Storage (EFS). When the image is deployed in Lambda, the scripts specify an EFS mounting point in the file system. The inter-process communication (IPC) layer uses DSO (see §3.2.1). The objects are made available in the shell using Picocli [120]. The IPC layer operates outside the serverless platform. To use it, the DSO servers have to be deployed beforehand, for instance using Kubernetes [25]).

The implementation of `sshell` for Lambda uses two layers [11]. The first layer contains the libraries common to all the deployments (e.g., IPC support). The second layer is light-weighted (a few KBs) and tailored for a specific use case. It includes the parameters specific to a end user (for instance, the EFS mounting point).

4.3 Evaluation

This section evaluates the performance and cost of `sshell`. We also compare it against a single-machine solution and a cluster-based approach (Apache Spark).

4.3.1 Experimental setup

The evaluation takes place in AWS (us-east-1). Unless stated otherwise, experiments are run from a t2.2xlarge machine (8 vCPUs - 32 GB RAM) located in the same datacenter—called the *client*. We use the default parameters for AWS Lambda and AWS Elastic File System (EFS). EFS operates in general purpose performance mode with default quotas. Each serverless function in Lambda has 1 GB of memory.

4.3.2 Preliminaries

This section presents preliminary results in which the serverless platform and the distributed storage are evaluated.

Invocation latency First, we evaluate the cost to invoke `sshell` in parallel with the code at line 1 in Figure 13. Figure 14 reports the invocation latency, that is the time spent at the client to execute this

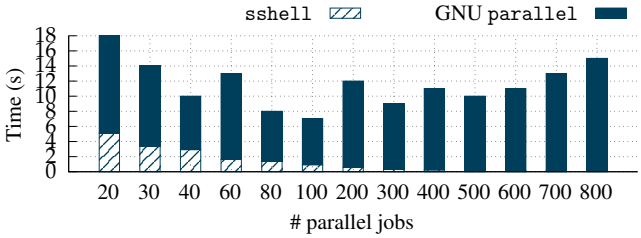


Figure 14: Invocation latency.

one-liner, for increasing values of `$JOBS`. Latency is broken down between the time spent in GNU `parallel` and executing `sshell` per se.

In Figure 14, we observe that the more parallel the task is, the less time is spent in invoking `sshell`. This is expected since increasing the parallelism reduces the number of sequential calls to the program. It also shows that the underlying serverless platform (AWS Lambda) scales well.[10] On the other hand, the total time does not decrease in Figure 14 when `$JOBS` $\geq$ 100. This comes from the fact that the client saturates: the performance of `parallel` drops when the amount of concurrent jobs is too large for the number of vCPUs.

	Download	Upload
Sequential	72 MB/s	77 MB/s
Parallel	3418 MB/s	1333 MB/s

Table 6: Peak transfer rates between EFS and Lambda.

Storage performance Table 6 reports the peak transfer rates between EFS and Lambda. The upload rate is computed by writing from Lambda to EFS using `dd` (line 2 in Figure 13). Piping these files to `/dev/null` provides the download rate (line 3 in Figure 13). As previously, we increment progressively variable `$JOBS` until the system saturates.

The observed peak upload and download rates are respectively 1.333 GB/s and 3.418 GB/s. These values, obtained with around 100 lambdas, are in-line with the documentation [132].

4.3.3 Micro-benchmarks

This section evaluates `sshell` against the micro-benchmarks proposed in [125].

Thumbnails The first benchmark parses a set of 1090 images. For each image, it generates a 10 KB thumbnail with ImageMagick[36]. The input dataset weights 14 GB in total.

Figure 15 presents the benchmark results. The files, both the original images and the thumbnails, are all stored in EFS. The figure details a comparison against a c5d.24xlarge machine (96 VCPUs, 192 GB RAM). In that case, the files are all local to the machine. At the time of writing, this machine is the largest (on-demand) instance available in EC2.

In Figure 15, `sshell` reaches its peak performance with 600 jobs (25 s). This is 14% faster than with the large machine (right of Figure 15). The large machine delivers its peak performance with 200 parallel jobs, a situation in which it is CPU bound.

For `sshell`, both the download and compute time shrinks with more parallel jobs. From 10 to 100 parallel jobs, `sshell` is 4.96x faster. However, as seen in §4.3.2, calling `parallel` with hundreds of jobs is expensive, capping the speedup.

[10] In the experiments, we always operate below the maximum concurrent invocations and invocation rates thresholds of AWS Lambda.

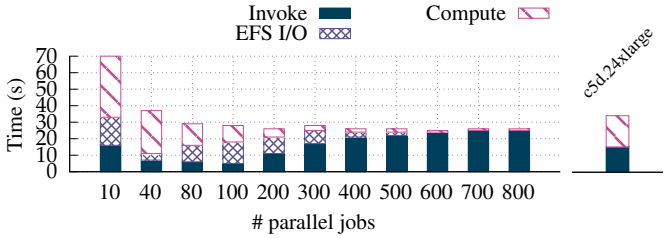


Figure 15: Thumbnails generation.

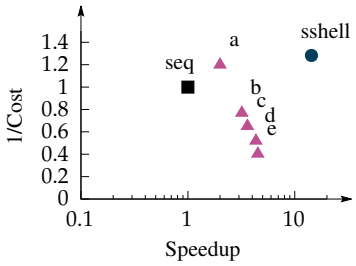


Figure 16: Port scan analysis – performance vs cost comparison between EC2 and `sshell`.

Port scan analysis The second benchmark parses a trace of 40 GB that consists of a full Internet scan of port 80 using `zmap`. The steps below are taken during the processing of the trace: *(1)* clean the raw input data with `zannotate`; *(2)* use `jq` to isolate first, the Internet Protocol (IP), then the Autonomous System (AS) columns; *(3)* call `pr` to merge these two outputs together; and *(4)* run `awk` and `sort` to count the number of IPs per AS.

Figure 16 presents our results for this benchmark. Running the sequential implementation by the authors of the benchmark on a t2.2xlarge machine takes 1803 s for a price of $0.19. The other results in Figure 16 are with a parallel script. Their performance is relative to the sequential implementation ("seq" in Figure 16), and reported along two dimensions: the x-axis measures the speedup, the y-axis is the inverse of the cost. For `sshell`, the cost corresponds to the use of Lambda and EC2. The other implementations use EC2 only. To compute the cost, we apply a per-request rate for Lambda, and a per-second billing for EC2.

When the benchmark is run in parallel, the input file is already split into 10^3 chunks. This pre-processing phase is not considered in Figure 16. In this figure, configurations (a-e) stand for an c5 on-demand EC2 instance with respectively, 16, 36, 48, 72 and 98 vCPUs. These instances are compute optimized which match the benchmark.

In Figure 16, we observe that executing the benchmark in parallel is clearly interesting. EC2 configurations offer different trade-offs in terms of price/performance. For instance, the fastest configuration (e) executes the benchmark in 400 s but it is 3x more expensive that the slowest configuration (a).

In Figure 16, we observe that `sshell` is clearly a better alternative, being cheaper and faster than a single-machine solution. The result reported in Figure 16 for `sshell` are obtained with 100 Lambdas in parallel. Higher parallelism does not improve significantly performance due to the limited time each job executes. Interestingly, as serverless providers bill per call, adding more parallelism has no cost impact, contrary to traditional Infrastructure-as-a-Service solutions.

4.3.4 Large-scale application

In the next experiment, we analyze TBs of data to estimate the popularity of web domains. Our `sshell` script mimics the behavior of LinkRun [138], an application that mines the datasets crawled

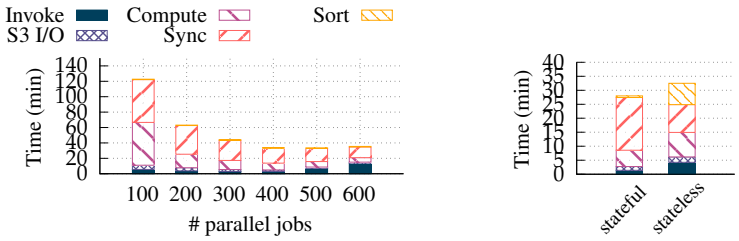


Figure 17: Ranking web domains by popularity.

monthly by the Common Crawl foundation using Apache Spark.

In detail, our script first downloads the web pages in the target dataset, and extracts all the outgoing links. Then, to evaluate the popularity of a domain, the script counts the number of times the domain is mentioned in each page. These results are aggregated over the whole dataset and sorted to construct the output.[11]

The Common Crawl datasets are stored in AWS S3, in the same region as where the experiment runs (us-east-1). A monthly crawl consists of 56,000 WAT files. Each file is an archive of around 400 MB, for a (compressed) total size of 20.17 TB.

Performance analysis Figure 17 reports the completion time of `sshell` to execute the above task. Figure 17(left) presents the performance with increasing levels of parallelism. The performance is split into five parts: (Invoke) invoking `sshell` in parallel; (S3 I/O) downloading the WAT archives; (Compute) extracting then processing these files; (Sync) merging the results between the `sshell` invocations using the IPC layer; and (Sort) ranking the domains based on their popularity. At maximum speed, with 400 jobs, `sshell` completes this task in 28 min. This is close to two times faster than with 200 jobs, and four times faster than with 100 jobs.

Interestingly in Figure 17 (right), sorting the output is always a fast operation (less than 35 s). This comes from the fact that this computation is run in the IPC layer, following a stateful approach (see §4.2.5). In the script, each job extracts locally from the archive a map that stores the popularity of the domains it encountered. The job then merges this local map in a red-black tree stored in the IPC layer and shared across all jobs (precisely, a meargeable map stored in DSO). The content of this tree is then extracted to obtain the final output.

It is possible to follow a stateless approach to sort the domains (similarly to Figure 9 wrt. to Figure 11). The client is then in charge of constructing the final result (e.g., with `awk`). As seen in Figure 17 (right), the sorting phase in this case is 13.8x slower than with the stateful approach.

Comparison with LinkRun In Table 7, we compare `sshell` to the original LinkRun application. For LinkRun, we indicate the time, cost and dataset size, as reported by its author [138].

	SLOC	Pricing	Time	Dataset size (compressed)
Linkrun	716	$200-260	26-48 h	17.62 TB
`sshell`	51	$19	28 min	20.17 TB

Table 7: Comparison with LinkRun.

While providing functionalities close to the original application, our code is more compact, thanks to the minimalist syntax of shell. It is also 56-103x faster to compute the popularity of the web domains. This substantial improvement in speed translates into a large cost difference.

[11] Such a measure of centrality (number of mentions) is a common indicator of the popularity of a node in a graph [47].

5 Extensions

In what follows, we present three extensions to the CRUCIAL framework and more specifically to the DSO layer. These extensions are particularly useful in the context of serverless computing. They are part of the Infinispan code base, and allow to use non-volatile memory to store persistent data (§5.1), compile DSO servers to native for faster execution (§5.2), scale up and down the DSO layer without moving data across servers (§5.3), and add a new operator for the kubernetes containers orchestrator (§5.4).

5.1 Support for non-volatile memory

Modern data stores and big data analytics platforms such as Infinispan are written in Java [19, 22, 39, 87, 110, 129, 156, 161]. Because they manipulate large amount of persistent data, these systems can greatly benefit from the recent technological advances in Non-Volatile Main Memory (NVMM) [72]. Unfortunately, to date, accessing efficiently NVMM from the Java language is still an open challenge.

To access NVMM from Java, two designs exist so far. With the *external design*, NVMM remains outside the Java heap. The Java virtual machine (JVM) accesses it through a file system [78, 94, 159], or using the Java native interface (JNI) [113, 115]. This design is inefficient due to the cost of converting data back and forth between the NVMM and the Java representations. With the *integrated design*, the JVM stores plain Java objects in NVMM and the application directly accesses them with read and write instructions [139, 158]. While this design avoids the conversion cost, it also has a fundamental flaw: the JVM has to run a garbage collector (GC) in NVMM because it now contains Java objects.

Collecting a dataset at the scale of NVMM, that is hundreds of GBs to TBs, is expensive.[12] For instance, we show in [91] that collecting just 80 GB divides the completion time by 3. Moreover, persistent and volatile objects have different life cycles. Applications often contain many allocation and deletion sites for volatile objects. On the contrary, the deletion of a persistent object is often related to a specific event, e.g. discarding a tuple in a relational database. Such events are rare, explicit, and trigger well-defined paths in the application. As we confirm in [91], this makes the number of deletion sites small, and thus the use of a GC for NVMM superfluous.

To remedy these problems, we propose a direct NVMM access, as with the integrated design, but without collecting persistent objects at runtime. Implementing this design is challenging because the Java language was designed for garbage-collected objects. To address this challenge, we revisit how to manage persistent objects for Java in the NVMM era. We introduce a *decoupling principle* between the data structure of a persistent object and its representation in the Java world. Based on this principle, a persistent object now consists of two parts: a data structure stored off-heap in NVMM, and a proxy that remains in volatile memory. The data structure holds the fields of the persistent object, while the volatile proxy provides the methods that manipulate them. Because we store the persistent data structure outside the Java heap, using our own memory layout, they are not collected at runtime. Our design also removes the cost of converting objects by leveraging a JVM interface that inlines the low-level instructions that access NVMM directly in the Java methods.

We implement our decoupling principle in the J-NVM framework. J-NVM is entirely written in Java and it only requires the addition of three NVMM-specific instructions to the Hotspot JVM. Our framework offers to the programmer a low-level interface that focuses on performance and a high-level interface that trades performance for usability. The low-level interface defines the methods that allow a proxy to access the persistent data structure. The high-level interface additionally provides failure-atomic blocks, that is blocks of code executed entirely or not at all [30, 34, 35, 105, 116, 152]. To ease programming, J-NVM includes a code generator which takes as input a legacy Java class and automatically decomposes it into a persistent data structure and a volatile proxy. Our framework also includes the J-PDT library which contains optimized persistent data structures (e.g., arrays, maps and trees) implemented directly atop the low-level library. Internally, these data structures do not rely on failure-atomic blocks for performance, yet they remain consistent when a crash occurs.

[12] The smallest Optane DC holds 128 GB of persistent memory.

We have evaluated J-NVM with micro-benchmarks and by implementing several persistent backends for the Infinispan data store [102]. Our evaluation using a TPC-B like workload [149] as well as YCSB [31] shows that:

- Both the low-level and the high-level interfaces systematically outperform the external design. In YCSB, the low-level interface is at least 10.5x faster than using ext4 atop NVMM or the PCJ library [115], which relies on the native PMDK library [115], except in a single case where it is only 3.6x faster.
- While the failure-atomic blocks of the high-level interface offer an all-around solution, J-PDT, with its hand-crafted persistent data types, executes up to 65% faster. Compared to a volatile implementation, J-PDT is only 45-50% slower.
- Integrating NVMM in the language runtime hurts performance due to the cost of garbage-collecting the persistent objects. For a Redis-like application written with go-pmem [57], increasing the persistent dataset from 0.3 GB to 151 GB multiplies the completion time of YCSB-F by 3.4

In the rest of this section, we explain briefly how to program with J-NVM (§5.1.1). Then, we present some evaluation results (§5.1.2). The interested reader may consult [91] for further details.

5.1.1 Programming with J-NVM

Overview J-NVM decomposes a persistent object into a persistent data structure and a volatile proxy. Persistent data structures live in NVMM, outside the Java heap. Proxies are regular Java objects that intermediate access to the persistent data structures. They implement the `PObject` interface, are instantiated on-demand (e.g., when a persistent object is dereferenced) and managed by the Java runtime. The above decoupling principle avoids running a garbage collector on persistent objects. Based on it, J-NVM implements a complete developer-friendly interface that offers failure-atomic blocks. To construct this interface, J-NVM reuses ideas and principles proposed in prior works, but assembles them differently. Our framework uses a class-centric programming model, that is the property of durability is attached to a class, and not to an instance. As common with prior frameworks (e.g., Thor [95]), a persistent object is live by reachability from a set of user-defined persistent roots. J-NVM garbage collects the unreachable persistent objects at recovery, but avoids running a GC at runtime for performance. Instead, objects are explicitly freed by the developer.

Example usage As illustrated in Figure 18, programming with J-NVM is straighforward. Any class annotated with `@Persistent` is durable. For instance, this is the case of `Simple` in Figure 18 (line 1). To run the application, the developer compiles the sources as usual, then passes a code generator over the bytecode files (the `.class` files). Any class marked with `@Persistent` is transformed.[13] The code generator replaces the volatile fields with persistent ones (lines 3 and 4 in Figure 18). Accordingly, accesses to such fields are replaced by persistent accesses (lines 8, 9, 12, 27 and 28). If a field is marked `transient` (line 5), the code generator keeps it in volatile memory, making no transformation. The developer may use transient fields to optimize the application, e.g., to deduce a volatile value from the persistent state. In addition to the above transformations, the code generator also wraps methods into failure-atomic blocks. The `fa="non-private"` argument of `@Persistent` at line 1 specifies that each non private method has to be wrapped. In the `Main` class of Figure 18, the application manipulates a `Simple` object. It starts by creating (or retrieving) a persistent memory region of 1 MB called `"/mnt/pmem/simple"` (line 17). A persistent memory region contains by default the persistent map `JNVM.root`. This map associates names with the root persistent objects used by the application. The `main` method uses this map to retrieve the persistent object associated with the name `"simple"`. If the object does not exist (line 19), the method allocates a new `Simple` object and records it in the map (line 20). Further, `main` retrieves the `simple` object, increments its `x` field, sets its `y` field and prints its

[13] If the sources are unavailable, instead of relying on the `@Persistent` annotation, the tool takes as input an explicit list of classes to transform.

```
1  @Persistent(fa="non-private")
2  class Simple {
3    PString msg;
4    int x;
5    transient int y;
6
7    Simple(int x) {
8      this.x = x;
9      this.msg = new PString("Hello, NVMM!");
10   }
11
12   void inc() { x++; }
13 }
14
15 class Main {
16   static void main(String args[]) {
17     JNVM.init("/mnt/pmem/simple", 1024*1024);
18
19     if(!JNVM.root.exists("simple"))
20       JNVM.root.put("simple", new Simple(42));
21
22     Simple s = (Simple)JNVM.root.get("simple");
23
24     s.inc();
25     s.y = 42;
26
27     System.out.println(s.x);
28     System.out.println(s.msg);
29
30     JNVM.root.put("simple", new Simple(24));
31     JNVM.free(s.msg);
32     JNVM.free(s);
33   }
34 }
```

Figure 18: How to use J-NVM.

content (lines 22-28). Line 30 creates a second `Simple` object and inserts it in the root map. The code then frees the first object still referenced by the local variable `s` (lines 31-32).

5.1.2 Evaluation

In this section, we present the performance of J-NVM in the YCSB benchmark and provide a detailed comparison against other existing approaches.

Hardware and system The test machine is a quad-Intel CLX 6230 hyperthreaded 80-core server with 128 GB of DRAM and 512 GB of Intel Optane DC (128 GB per socket). It runs Linux 4.19 with gcc 8.3.0, glibc 2.28 and Hotspot 8u232-b03 (commit c5ca527b0afd) configured to use G1. The patch for Hotspot that adds the three NVMM-specific instructions to `Unsafe` (namely, `pwb`, `pfence` and `psync`) contains 200 SLOC. Besides this patch, J-NVM, J-PDT and J-PFA that all together implement our NVMM object-oriented programming framework, encompass about 4000 SLOC. NVMM runs in App Direct mode and is formatted with the ext4 file system. In this mode, software has direct byte-addressable access to NVMM.

Infinispan Our experiments use Infinispan, an open-source industrial-grade data store maintained by Red Hat. Infinispan exposes a cache abstraction to the application that supports advanced operations, such as transactions and JPQL requests. We use Infinispan version 9.4.17.Final [103], which contains around 600,000 SLOC. Infinispan runs either with the application (embedded mode), or as a remote storage (server mode). Unless stated otherwise, we use the embedded mode during our experiments and cache up to 10% of the data items. As seen in [91], a larger ratio would significantly

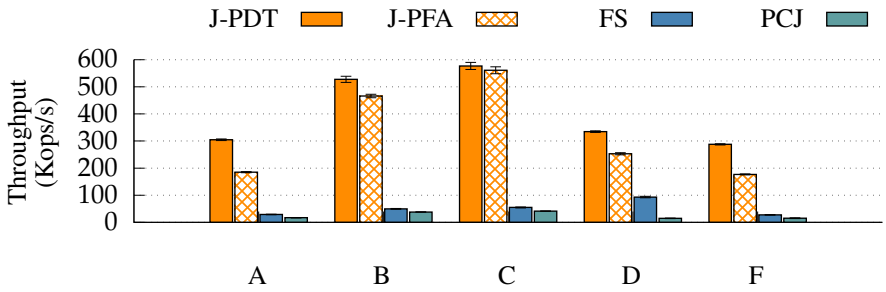


Figure 19: The YCSB benchmark.

harm performance. Accordingly, we also cap the volatile heap to 22 GB. This size gives the best performance with our YCSB workload on a file system backend atop NVMM (precisely, less then 3.7% of the total time is spent in GC in Figure 19).

Persistent backends We evaluate different persistent backends for Infinispan: ***(J-PDT)*** A backend using the J-PDT standalone library. ***(J-PFA)*** A backend built with the failure-atomic blocks of J-NVM. ***(FS)*** The default file system backend of Infinispan using NVMM formatted in ext4. ***(PCJ)*** An implementation that relies on the Persistent Collections for Java library [115]. PCJ uses the native PMDK 1.9.2 library [116] through the Java Native Interface.

YCSB Benchmark We compare J-NVM against the other approaches by running version 0.18 of the Yahoo! Cloud Serving Benchmark (YCSB) [31] YCSB is a key-value store benchmark that consists of six workloads (A to F) with different access patterns. A client can execute six types of operations (`read`, `scan`, `insert`, `update` and `rmw`) on the key-value store. Workload A is update-heavy (50% of `update`), B is read-heavy (95% of `read`) and C is read-only. Workload D consists of repeated reads (95% of `read`) followed by insertions of new values. In the workload E, the client executes short scans. Workload F is a mix of read and read-modify-write (`rmw`) operations. We evaluate all workloads except E. Infinispan only provides `scan` through the JPQL interface, hence workload E is not comparable with the others that use a direct interface. If not otherwise specified, YCSB executes in sequential mode (single-threaded client).

YCSB associates a key with a data record that contains fixed length fields. Unless otherwise stated, we use the default parameters of 3M records, each having 10 fields of 100 B. YCSB runs with the default access patterns (namely, zipfian and latest). Compared to a uniform distribution, these patterns improve the cache hit ratio, and makes thus the FS backend more efficient.

J-PDT, J-PFA and PCJ all require to use persistent keys and values in YCSB. To achieve this, we modified the Infinispan client, which represent less than 30 SLOC from the vanilla version.

Results. Figure 19 presents the throughput of the YCSB benchmark with the different persistent backends. In this figure, we observe first that J-PDT systematically outperforms the other approaches. Except in workload D, J-PDT is consistently 10.5x faster than FS. In comparison to PCJ, the difference ranges between 13.8x and 22.7x faster. In workload D, J-PDT executes at least 3.6x more operations per second than FS and PCJ. The low performance of FS comes from the cost of marshalling persistent

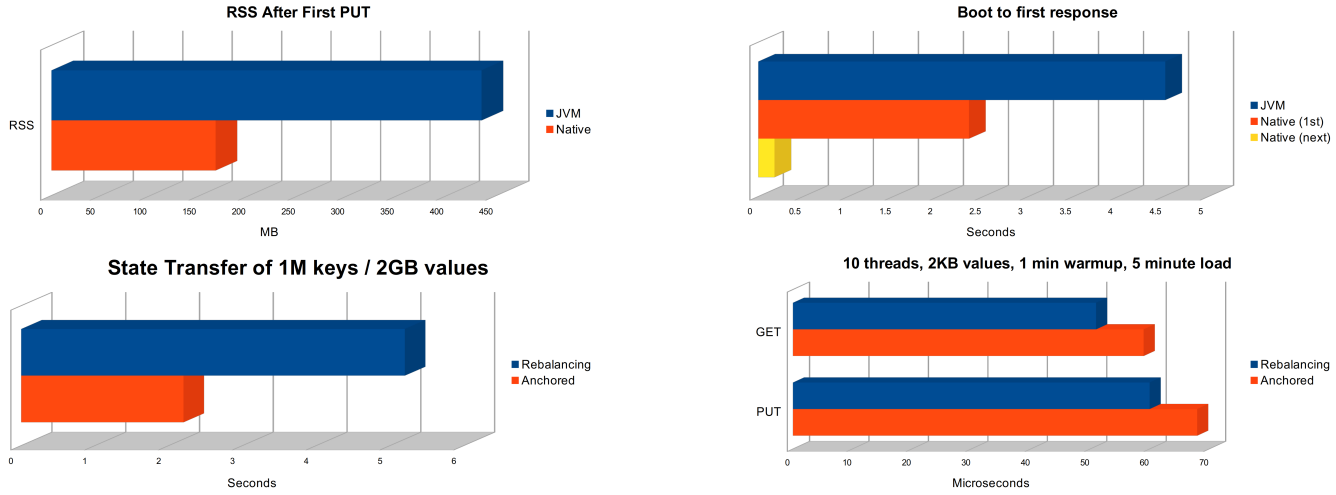


Figure 20: Illustration of the performance improvements in recent Infinispan releases—*(top)* native compilation, *(bot)* anchored keys.

objects back and forth between their file system and Java representations. In Figure 19, the lower performance of PCJ is due to the Java native interface that requires heavy synchronization to call a native method [113]. J-NVM avoids this cost by leveraging the `Unsafe` interface, which does not have to synchronize the whole JVM to escape the Java world. Overall, the results in Figure 19 outline that NVMM drastically changes the way to access persistent data from the Java runtime: while JNI calls or marshalling/unmarshalling operations were negligible with slow storage devices, this is no more the case with NVMM. They must be avoided where possible. In Figure 19, J-PFA also systematically outperforms FS and PCJ for the same reasons as mentioned above. Nevertheless, J-PDT is still up to 65% faster. This result shows that hand-crafted crash-consistent data structures can be more efficient than a generic approach.

5.2 Ahead-of-time compilation

One of the concerns of using Java in high-density environments is the overhead of the Java Virtual Machine (JVM) both in terms of memory usage as well as in startup and warming-up time. Most of the blame for this doesn't actually lie in the JVM itself, which is still one of the best available optimizing virtual machines, but in the typical dynamic approach of many development frameworks which rely on runtime class reflection, annotation scanning and bytecode enhancements.

However, the overhead of the JVM can still be drastically reduced by using ahead-of-time compilation (AOT), where Java source code can be directly compiled to machine code. Oracle has recently released the GraalVM project, which, among other things, delivers a "native-image" tool which generates a native binary from a Java application which does not require a JVM at runtime. This kind of native binary is ideal for applications which need very fast startup time with low memory overhead, which is typical in short-lived execution scenarios like FaaS.

Real-world examples have demonstrated a 100-fold speedup in startup time for a microservice-style application (from 9 seconds to less than 1/10 of a second) and a 10-fold reduction in RSS (Resident Set Size) memory usage (from 250MB to 25MB). Because the native-image tool imposes some limitations on the kind of code that can be compiled and executed, traditional Java applications and libraries need to be altered.

While it was possible to compile Infinispan to native code with very few modifications, the decision was made to replace a lot of the code that performed injection and marshalling at runtime with completely refactored components that performs these tasks at compile-time. For example, the reflection-based Java class marshalling was substituted with a "Protocol Buffers" [151] implementa-

tion which generates serialization/deserialization code during the compilation phase.

Figure 20(top) illustrates the (drastic) performance improvements brought by the native compilation. In particular, and as expected, the boot time is reduced to a very small fraction of what it was previously (top right figure).

Results Every variant of Infinispan (e.g., embedded and server), as well as the Java-based clients now compiles ahead-of-time. This work is now part of the main development tree [147]

5.3 Anchored keys

Infinispan's horizontal scaling has always been driven by three main goals: *(i)* high-availability, i.e. the ability to recover from the loss of one ore more nodes by keeping multiple copies of data across the cluster; *(ii)* balancing load evenly across all nodes by ensuring that data is evenly distributed across the cluster, and *(iii)* allowing intelligent clients to retrieve data from the nodes owning it in a single hop.

As pointed out in §3.2.1, the key to achieving the above goals has been consistent-hashing: Nodes which own a particular item of data are computed based on a hashing algorithm known by both the server and the clients. When a topology change happens (nodes joining/leaving the cluster), the consistent-hash mapping of data to nodes (called ownership) is recomputed according to the available nodes. If necessary, entries are moved around to ensure high-availability and even distribution and load-balancing.

While the rebalancing algorithm tries to perform as little moving of data as possible and in a non-blocking fashion, it still may have a non-negligible impact on throughput and latency while it is running. For this reason a new optional data distribution algorithm, named "Anchored Keys", has been implemented since Infinispan 11. This algorithm has been designed to minimize the data which is exchanged by nodes in case of a topology change. When anchored keys are enabled, Infinispan no longer uses consistent-hashing to determine data ownership but uses the last (most recently added) node in the cluster. However, in order for all nodes to know which node owns a specific key, the ownership information needs to be propagated to all nodes. This information is stored in the extended metadata of each entry, which is then replicated to all nodes.

The current implementation of anchored keys sacrifices availability by not maintaining multiple copies of each item, but this will be amended in a future version by maintaining one or more symmetrical copies of each node.

Figure 20(bot) illustrates the interest of using anchored keys to scale horizontally Infinispan without paying the price of moving data across servers. The time to bootstrap a server is reduced (bottom left figure), with a low impact on performance (bottom right figure).

Results The new anchored-keys module is now part of Infinispan 11 and will evolve to add more capabilities, such as high-availability [147].

5.4 Kubernetes operator

Provisioning, monitoring, scaling and upgrading an Infinispan cluster is a complex undertaking, which involves configuring multiple subsystems, including memory and CPU resources, networking, security and persistent storage. Some of the functionalities required by these operations, such as node discovery, clustered configuration, automatic data rebalancing, was already handled by Infinispan's internal clustering and configuration logic, but it still required a lot of manual uplift. Kubernetes is a managed, containerized orchestrator, which has been designed and developed to handle most of the complexity of a distributed system involving multiple components, each with their own lifecycle and management requirements. Kubernetes offers infrastruture, APIs and policies that can handle the most common distributed deployments. However, for systems which do not fit into the standard behaviours, an extensible layer, known as the Operator Lifecycle Management (for short,

Operator), has been implemented. It is therefore possible to install Operators, which are custom applications whose task is to monitor the global Kubernetes configuration namespace for specific types of Custom Resources (CRs).

Infinispan Operator, extends the Kubernetes API with Custom Resource Definitions (CRDs) for deploying and managing Infinispan clusters on Kubernetes. To interact with Infinispan Operator, Kubernetes users apply CRs through the Kubernetes Dashboard, or directly by using the `kubectl` client. Infinispan Operator listens for Infinispan CRs and automatically provisions native resources, such as StatefulSets and Secrets, that the Infinispan deployment requires. Infinispan Operator also configures Infinispan services according to the specifications in Infinispan CRs, including the number of pods for the cluster and backup locations for cross-site replication. A single operator installation can manage multiple Infinispan clusters in separate namespaces. Each time a user applies CRs to modify the deployment, the operator applies the changes globally to all Infinispan clusters. Infinispan Operator reconciles CRs such as the Cache CR with resources on an Infinispan cluster. Bidirectional reconciliation synchronizes CRs with changes that are made to Infinispan resources through the Infinispan Console, command line interface (CLI), or other client application and vice versa. For example, if a cache is created through the Infinispan Console then Infinispan Operator adds a declarative Kubernetes representation. If the Kubernetes cluster has a Prometheus and Grafana installation, then the operator can also automatically setup metrics scraping and provides a default dashboard for monitoring Infinispan.

Results Infinispan Operator allows to integrate Infinispan into the Kubernetes ecosystem. This benefits to (serverless and traditional) applications that use it to persist and cache data within a unified modern container-based ecosystem.

6 Exploratory work

Exploratory work is an important part of a European-funded RIA because it investigates and develops ideas that advance the state of the art both in research and industry. This section explains the exploratory work that has been conducted during the project. It also provides indications on how to decide whether this work will make it into a future version of CRUCIAL.

This work is risky in the sense that not all of it will become part of the reference architecture. Such risk-taking is necessary in all successful research. Success of this work is to be measured not on how much of the work becomes part of the reference architecture, but on whether sufficiently innovative exploration is done and on whether the reference architecture itself is sufficiently innovative.

6.1 T4.2 - Degradable objects

In a distributed system, data is replicated for availability and to boost performance (typically, with more read replicas). When replicated data is mutable, it is necessary to maintain consistency with the help of a concurrency control mechanism. Due to the CAP and FLP impossibility results [45, 58], orchestrating data replicas is notably difficult and moreover subject to conflicting requirements. On the one hand, strong consistency maintains the sequential invariants of the applications and is well understood. On the other hand, performance and scalability suggest to use of a weaker consistency criterion, yet this requires considerable programming skills. A key challenge is thus to find a good balance between the programming model of the target distributed application, and its deployment constraints and performance requirements.

To reconcile programming model and data consistency, Task T4.2 investigates the notion of degradable object. A degradable object is a mutable shared data type whose behavior varies to match the requirements of an application. More precisely, a degradable object is a hierarchy of object types all having the same signature, but with varying pre- and post-conditions for their operations and that abide by different consistency criteria. Each level of this hierarchy is called a degradation level. The

key principle is that the degradation level L+1 requires less synchrony to implement than the level L. Thus, it is more efficient and more scalable, but also less convenient to program with.

The programmer specifies the degradation level to use according both to the invariants of the application and its performance requirements. Finding the appropriate level for a given application pattern is an iterative process. At first glance, a programmer may use strong consistency, then later refines her choices based on the fact that some interleavings and/or inconsistencies are acceptable. Our key insight here is that this iterative process will offer a principled and pedagogical approach to understand and use (weak to strong) data consistency in distributed applications.

With more details, our efforts have been conducted so far on three fronts.

- First, we are collaborating with the H2020 LightKone project [93] on introducing a new communication primitive in AntidoteDB [14]. AntidoteDB is a distributed database of conflict-free replicated data types (CRDTs). In the traditional CRDT approach, operations that are mutating a replica are executed in the background, outside the critical path. Their side effect (aka., the effector [134]) is then propagated eventually to all the replicas, for instance an epidemic protocol. Our new primitive will maintain this behavior, but will also offer better properties if needed (e.g., on the delivery order of effectors). The end goal is to allow some operations to execute under stricter consistency conditions than strong eventual consistency, the default criteria of CRDTs [136].
- Our second effort is on the specification and definition of degradable objects. We investigate the link between the specification of a sequential data type and the need for process synchronization. Typically, process synchronization is measured by the consensus power of a given data type. The consensus power is the largest number of processes that are able to solve consensus with this data type and registers. Starting from base shared objects, we are investigating how consistency degradation reduces the consensus power.
- The consensus power is formulated with linearizable objects, that is, in the classical shared memory model. As a consequence, this hierarchy does not fully capture the need for synchronization in a distributed message-passing system. To close this gap, we investigate alternative definitions to characterize process synchronization. In particular, our investigation covers the k-set agreement hierarchy and the link between failure detectors and quorums of data replicas [53].

Results We are currently writing a research article covering the notions of consistency degradation and its interest wrt. both strong and weak consistency [81]. A prototype library of degradable objects is also being implemented [80]. Preliminary results show a up to 10x improvement over regular objects in the Apache Cassandra key-value store.

6.2 T4.3 - Just-right synchronization

The classical way of maintaining shared objects strongly consistent is state-machine replication (SMR) [131]. In SMR, an object is defined by a deterministic state machine, and each replica maintains its own local copy of the machine. An SMR protocol coordinates the execution of commands at the replica, ensuring that they stay in sync. This requires to execute a sequence of consensus instances each agreeing on the next state-machine command. The resulting system is linearizable, providing an illusion that each command executes atomically throughout the system.

Strong consistency is necessary to help transitioning legacy code from shared-memory to serverless architecture. As pointed out in §3.2.1, it also helps the programmer to use a distributed programming framework by providing a familiar semantic. For both of these reasons, it was a key concern when developping CRUCIAL.

On the other hand, it is well-known that the above classical SMR scheme limits scalability. To sidestep this performance problem and further scale the size of the data sets that CRUCIAL is able to process, we investigated (i) how to improve the scalability of SMR with leaderless consensus, and

(ii) the design of an efficient atomic multicast protocol to deal with partial replication. These two lines of work are detailed below.

6.2.1 Leaderless consensus

To date, SMR protocols do not scale, that is when more replicas are added to the system, the performance of the replicated service degrades. This situation results from the conjunction of several pitfalls:

- First of all, a large spectrum of protocols, e.g., Paxos [88], Raft [112] or Zab [77], funnel commands through a leader (aka. primary) replica. This approach increases latency for clients far away from the leader and decreases availability because if the leader fails, the system halts to elect a new one. To mitigate these drawbacks, leaderless approaches [101, 108] allow each replica to contact a quorum of its peers to execute a command.
- A second concern is that many standard solutions rely on large quorums to make progress. For instance, in a system of n replicas, Fast Paxos [90] accesses at least $\frac{2n}{3}$ replicas, EPaxos [108] $\frac{3n}{4}$, and Mencius [101] contacts them all. Large quorums harms system reliability and scalability because more replicas have to participate to the ordering of each command.
- A last concern is the communication delay to execute a command. To minimize service latency SMR protocols should leverage non-conflicting commands, that is commands which are not concurrent to any other non-commuting command. These commands are frequent in distributed applications [26, 33] and can execute in a single round-trip [89].

Refining the above observations, we have introduced a set of desirable requirements for SMR: Reliability, Optimal Latency and Leaderlessness (ROLL), We have shown that attaining all the ROLL properties is subject to a trade-off between fault-tolerance and scalability. More specifically, in a system of n processes, the ROLL theorem states that every leaderless SMR protocol that tolerates f failures must contact at least $(n - \frac{(n-f)}{2})$ processes to execute a command in a single round-trip.

Simultaneous failures and/or asynchrony periods are however a rare event. Leveraging this fact, we have proposed a novel SMR protocol named Atlas which, based on the ROLL theorem, is optimal. In particular, Atlas offers two distinguishable unique features.

- First, it executes a command by contacting the closest $\lfloor \frac{n}{2} \rfloor + f$ processes. For small values of f, this implies that the protocol scales.
- The protocol applies commands using a fast path that completes after one round trip, or a slow path, which completes after two round trips. We introduce a new condition that allows commands to take the fast path even in the presence of conflicts. In particular, when $f = 1$, the protocol always takes the fast path.

We have experimentally compared Atlas against Paxos [88], EPaxos [108] and Mencius [101] on Google Cloud Platform using the YCSB benchmark [32]. Our results show that our approach consistently outperforms these protocols. In particular, the protocol scales when f is small in the sense that adding more nodes close to the clients improves latency.

Results The Atlas protocol is detailed in the proceedings of the Eurosys 2020 conference [42]. Its code base [43] is available upon request. In a recent work, that appeared in the Information Processing Letter [144], we study the correctness of the Egalitarian Paxos protocol. The work on the ROLL theorem [52] was accepted to DISC 2020.

6.2.2 Atomic multicast

Atomic multicast is a communications primitive that allows a group of processes to receive messages in an acyclic delivery order. This primitive is a useful building block for distributed storage systems that enforce strong consistency properties. As an example, it is used in Infinispan to implement distributed transactions. The main difference with atomic broadcast, which serves a similar purpose, is that a message can be addressed to a subset of the processes. To be scalable, atomic multicast

protocols must be genuine, that is only the destination group of a message should be involved in its ordering.

The standard fault-tolerant genuine solution layers Skeen's multicast protocol on top of Paxos to replicate each destination group. Recent improvements decrease the latency of this standard solution by adding a parallel speculative execution path. Under normal operation, the standard protocol can deliver multicast message in 6 communication delays and such an optimized version in 4 communication delays.

Standard protocols employ the Paxos consensus protocol as a blackbox. Departing from this traditional way of guaranteeing fault-tolerance, we propose a new solution that weaves Paxos together with Skeen's multicast. The resulting white-box multicast protocol embeds its own replication logic, enabling message ordering and delivery in 3 communication delays under normal operation.

Our protocol offers better theoretical performance. We have experimentally assessed that such characteristics pay-off in practice. We implemented our protocol in the same framework as Skeen's and its optimized variation and conducted a comparative performance analysis of the three protocols. Our protocol offer better latency than prior works (up to 2x faster than the optimized Skeen variation). It also sustains a much higher number of concurrent client requests, thanks to its lower message complexity.

Results The work on white-box atomic multicast [60] was presented at DSN 2019. Its implementation is open source [61].

7 State of the Art

Serverless computing brings cost-efficiency and elasticity to software development. This new paradigm has gained traction recently and many works have been proposed in this area. In what follows, we cover runtimes (§7.1), programming frameworks (§7.2) and storage (§7.3) for cloud functions. The bottom of this section (§7.4) focuses on (serverless and non-serverless) solutions to the problem of stateful distributed computation. Table 8 outlines our survey, comparing CRUCIAL to other existing serverless systems that address the problem of state sharing and coordination.

7.1 Runtimes

Serverless computing has appealing characteristics, based on simplicity, high scalability and fine grained execution. It has seduced both industry [10, 107, 118] and academia [64]. This enthusiasm has also led to a blossom of open-source systems (e.g., [1–4, 64] to cite a few).

At core, a cloud function runtime is in charge of maintaining the user-defined functions, executing them upon request. It must ensure strong isolation between function instances and low startup latency for performance. Many works propose to tackle these two central challenges.

Micro-kernels [100] offer a solid basis to quickly kick-off a function and attain sub-millisecond startup time. Catalyser introduces the `sfork` system call to reuse the state of a running sandbox. Similarly, Firecracker [7] makes containers more lightweight and faster to spawn. SOCK [111] is a serverless-specialized system that uses a provisioning mechanism to cache and clone function containers. SAND [8] exploits function interaction in FaaS to improve the performance applications. The system relaxes isolation at the application level, enabling functions from the same application to share memory and communicate through a hierarchical message bus. This allows better latency and resource efficiency when combining functions. Faasm [137] offers similar guarantees using a language-agnostic runtime built atop WebAssembly. User functions use all the same substrate to execute, allowing fast initialization. They can access a distributed key-value store cached locally and shared across functions located on the same physical machine.

Two recent works [63, 75] coincide with our view that existing runtimes do not support *mutable shared state* and *coordination* across cloud functions. Hellerstein et al. [63] underline that the model is

System	Shared state	Synchronization	Durability	Consistency
PyWren	Object store	coarse-grained	replication	weak
ExCamera	rendezvous	rendezvous	—	—
Ripple	composition	composition	—	—
Pocket/Crail	multi-tiered	coarse-grained	ephemeral	—
Cloudburst	FaaS + cache	coarse-grained	replication	weak
Faasm	shared memory	fine-grained	—	—-
CRUCIAL	DSO	fine-grained	replication	strong

Table 8: Serverless solutions for state sharing and coordination.

a data-shipping architecture that imposes indirect communication and hinders coordination. Jonas et al. [75] highlight the lack of adequate storage for fine-grained operations and the inability to coordinate functions at fine granularity.

7.2 Programming frameworks

Several works that address the above two challenges confront them from a function composition perspective: a scheduler orchestrates the execution of stateless functions and shares information between them.

Many public cloud services support function composition. AWS allows creating state machines with Step Functions [12]. IBM Composer [50] offers a similar solution. Google Cloud Composer [119] allows to easily create and run a DAG of tasks in the cloud. Azure Durable Functions [107] enables to programmatically coordinate function calls. AWS has its own Amazon States Language to define the state machines. Unfortunately, the JSON-based language may become complicated for complex workflows. IBM's solution facilitates coding with a JavaScript API that is later on transformed into a state machine. While both enable state combinations, the expressiveness is very limited. Google Cloud Composer is based on Apache Airflow. It runs a per-user dedicated server that acts as a scheduler. Composition is limited to a DAG of tasks, which is inherently less expressive than state machines. Azure Durable Functions is the most complete solution among all, allowing to directly write imperative code. Asynchronous calls to functions are expressed in C# permitting a function to wait explicitly prior results.

All the above services struggle to execute embarrassingly-parallel tasks [16, 54]. To sidestep this limitation, PyWren [74] pioneered the idea to use FaaS for bulk synchronous parallel (BSP) computations. The paper shows the elasticity and scalability of FaaS and demonstrates with a base Python prototype how to run MapReduce workloads. PyWren uses a client-workers architecture where stateless functions read and write data to cloud storage (mainly Amazon S3). Numpywren [133] is a framework for linear algebra computations over cloud functions. Like with PyWren, functions are managed as a pool of stateless workers and tasks are managed in a queue. IBM-PyWren [130] evolves the PyWren model with new features and enhancements. Locus [124] enhanced PyWren for analytics computation on top of cloud functions. It focuses on the shuffling phase of the MapReduce scheme and combines cheap but slow storage with fast but expensive storage to explore a cost-performance trade-off. ExCamera [48] is another system atop FaaS, more focused on video encoding and low latency. Its computing framework (*mu*) is designed to run thousands of threads (as an abstraction for cloud functions) and manages inter-thread communication through a rendezvous server. *gg* [49] keeps *mu*'s line for running serverless parallel threads but taken to a broader audience.

Ripple [76] is a programming framework to take single-machine applications and allow them to benefit from serverless parallelism. Users rely on a simple interface to express the high-level dataflow of their applications. Ripple automates resource provisioning and handles fault tolerance by eagerly detecting straggler tasks. With the user definition, the framework is able to apply heuristics to abstract data partitioning, task scheduling, resource provisioning and fault tolerance. Before the full computation run, the framework performs a series of dry runs to test and find the best resource

provisioning for the job.

Some recent works attempt to build theoretical foundations for programming with cloud functions. Jangda et al. Jangda et al. [73] propose a concise calculus to capture the operational semantics of serverless computing. Baldini et al. [15] detail the serverless trilemma: functions should be black boxes, composition should be also a function and invocations should not be double-billed. They present a solution to the trilemma for sequential compositions called IBM Sequences.

7.3 Storage

Many frameworks focus on cloud function scheduling and coordination, while using disaggregated storage to manage data dependencies. In particular, they opt to write shared data to slow, highly-scalable storage [74, 130, 133]. To hide latency, they perform coarse-grained accesses, resort to in-memory stores, or use a combination of storage tiers [124].

Pocket [85] is a distributed data store that scales on demand to tightly match the space needs of serverless applications. It leverages multiple storage tiers and right-sizes them offline based on the application requirements. Crail [143] presents the NodeKernel architecture with similar objectives. These two systems are designed for ephemeral data, which are easy to distribute across a cluster. They do not use a distributed hash table that would require data movement when the cluster topology changes, but instead use a central directory. Both systems scale down to zero when computation ends.

InfiniCache [153] is an in-memory cache built atop cloud functions. The system exploits FaaS to store objects in a fleet of ephemeral cloud functions. It uses erasure coding and a background rejuvenation mechanism to maintain data available despite the churn. Similarly to a traditional distributed in-memory cache, InfiniCache is designed to hold objects but not to facilitate their update.

The above works do not allow fine-grained updates to a mutable shared state. Such a feature can be abstracted in various ways. CRUCIAL chooses to represent state as objects, and keeps the well-understood semantics of linearizability. This approach is in-line with the simplicity of serverless computing.

Existing storage systems such as Memcached [46], Redis [126], or Infinispan [103] cannot readily be used as a shared object layer. They either provide too low-level abstractions or require server-side scripting. Coordination kernels such as ZooKeeper [68] can help synchronizing serverless functions. However, their expressiveness is limited and they do not support partial replication [41, 79]. We show these problems in §3.4.

CRUCIAL borrows the concept of callable objects from CRESON [145]. It simplifies its usage (`@Shared` annotation), provides control over data persistence and offers a broad suite of synchronization primitives. While CRUCIAL implements strong consistency, some systems [135, 141, 148] rely instead on weak consistency, trading ease of programming for performance. Weak consistency has been used to implement distributed stateful computation in FaaS, as detailed in the next section.

7.4 Distributed stateful computation

Cloudburst [142] is a stateful serverless computation service. State sharing across cloud functions is built atop Anna [157], an autoscaling key-value store that supports a lattice put/get CRDT data type. Cloudburst offers repeatable read and consistent snapshot consistency guarantees for function composition—something that is not achievable, for instance, when using AWS Lambda in conjunction with S3 (i.e., computing $x + f(x)$ is not possible if x mutates).

Cirrus [28] is a machine learning framework that leverages cloud functions to efficiently use computing resources. This system specializes in iterative training tasks and asynchronous stochastic gradient descent. The initial motivation for Cirrus is much in line with CRUCIAL, however the solution is quite different. Cirrus relies on a distributed data store that does not allow custom shared objects and/or computations. Furthermore, distributed workers cannot coordinate as they do in CRUCIAL.

Besides serverless systems, there exist many frameworks for machine clusters that target stateful distributed computation.

Ray [109] is a recent specialized distributed system mainly targeting AI applications (e.g., Reinforcement Learning). It offers a unified interface for both stateless task-parallel and stateful actor-based computations. Applications use both types combined and the system runs a single dynamic execution engine. Ray achieves high-scalability with a bottom-up distributed scheduler and fault-tolerance using a chain-replicated key-value store. Its architecture is based on cluster provisioning and does not fit the serverless model. CRUCIAL shares Ray's motivation for the need of a specialized system that combines stateful and stateless computations. However, Ray couples both models in the same system and is built for a provisioned resource environment where stateless tasks and actors live co-located. CRUCIAL is built with serverless in mind and separates the two types of computation. Our system uses the highly scalable capabilities of FaaS platforms for stateless tasks and a layer of shared objects for data sharing and coordination. The programming model is also consequently different: while Ray exposes interfaces to code tasks and actors, CRUCIAL uses a traditional shared-memory model where concurrent tasks are expressed as threads.

Other systems with a focus on stateful computations, such as Dask and PyTorch, usually build on low-level technologies (e.g., MPI) to communicate among nodes. These frameworks rely on clusters with known topology and struggle to scale elastically. Such a design is at odds with the FaaS model, where functions are forbidden to communicate directly.

Specialized distributed big data batch processing frameworks, like MapReduce, are available as a service in the cloud (e.g. AWS EMR). We explore such alternatives in the evaluation section (§3.4.2), where we compare against Apache Spark.

8 Conclusion

This document presents CRUCIAL, a powerful system to write efficient serverless programs. CRUCIAL was developped in the context of CloudButton, a joint European effort to simplify data analytics and data processing with serverless technologies.

CRUCIAL offers a simple interface to serverless, allowing to write (or port) effortlessly parallel code for this new environment. It is structured into a compute tier running atop a FaaS platform and a dedicated in-memory distributed storage tier (DSO). We demonstrate how to use CRUCIAL in the context of data-intensive applications (e.g., bulk processing, parallel processing, ML). CRUCIAL allows to move to serverless existing parallel shared-memory code bases in a few modifications. The performance (and costs) are on par with a cluster of high-end servers running a dedicated complex software (such as Apache Spark).

This document also presents the serverless shell (`sshell`), an adaptation of the Unix shell for serverless. `sshell` brings the massive copmutation power of the cloud to regular shell scripts.

In addition, this document presents the recent additions to the distributed shared object (DSO) layer of CRUCIAL: a library to use non-volatile memory in Java, a support for native compilation, as well as, a new kubernetes operator.

Last, we covered the exploratory work conducted during the CloudButton project in WP4. Part of this exploratory work may be included into future versions of the CRUCIAL framework.

References

[1] Serverless functions for kubernetes - fission, 2016. URL https://fission.io/.

[2] Kubeless, 2016. URL https://kubeless.io/.

[3] Openfaas, 2016. URL https://www.openfaas.com/.

[4] Apache openwhisk is a serverless, open source cloud platform, 2016. URL https://openwhisk.apache.org/.

[5] lambda-maven-plugin. https://github.com/SeanRoy/lambda-maven-plugin, 2019.

[6] https://github.com/crucial-project, 2020.

[7] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434, Santa Clara, CA, February 2020. USENIX Association. ISBN 978-1-939133-13-7. URL https://www.usenix.org/conference/nsdi20/presentation/agache.

[8] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. Sand: Towards high-performance serverless computing. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, pages 923–935, Berkeley, CA, USA, 2018. USENIX Association. ISBN 978-1-931971-44-7. URL http://dl.acm.org/citation.cfm?id=3277355.3277444.

[9] Amazon. Aws simple storage service. https://aws.amazon.com/s3, 2008. retrieved Aug. 2019.

[10] Amazon. Aws lambda. https://states-language.net/spec.html, 2014. retrieved Aug. 2019.

[11] Amazon. Aws lambda layer. https://docs.aws.amazon.com/lambda/latest/dg/configuration-layers.html, 2014.

[12] Amazon. Aws step functions. https://aws.amazon.com/step-functions, 2016.

[13] Amazon. Aws glue. https://aws.amazon.com/glue/, 2017. retrieved Aug. 2019.

[14] AntidoteDB. https://www.antidotedb.eu.

[15] Ioana Baldini, Perry Cheng, Stephen J. Fink, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Philippe Suter, and Olivier Tardieu. The serverless trilemma: Function composition for serverless computing. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2017, page 89–103, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450355308. doi: 10.1145/3133850.3133855. URL https://doi.org/10.1145/3133850.3133855.

[16] Daniel Barcelona-Pons, Pedro García-López, Álvaro Ruiz, Amanda Gómez-Gómez, Gerard París, and Marc Sánchez-Artigas. Faas orchestration of parallel workloads. In *Proceedings of the 5th International Workshop on Serverless Computing*, WOSC '19, page 25–30, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450370387. doi: 10.1145/3366623.3368137. URL https://doi.org/10.1145/3366623.3368137.

[17] Daniel Barcelona-Pons, Pierre Sutra, Marc Sánchez-Artigas, Gerard París, and Pedro García-López. Stateful serverless computing with crucial. *ACM Trans. Softw. Eng. Methodol.*, 31(3), mar 2022. ISSN 1049-331X. doi: 10.1145/3490386. URL https://doi.org/10.1145/3490386.

[18] Mordechai Ben-Ari. How to solve the santa claus problem. *Concurrency: Practice and Experience*, 10, 2001. doi: 10.1002/(SICI)1096-9128(199805)10:63.0.CO;2-2.

[19] Shamim Bhuiyan, Michael Zheludkov, and Timur Isachenko. *High Performance In-Memory Computing with Apache Ignite*. Lulu.com, 2017. ISBN 1365732355.

[20] Kenneth P. Birman and Thomas A. Joseph. Reliable Communication in the Presence of Failures. *ACM Transactions on Computers Systems*, 5(1):47–76, January 1987. ISSN 0734-2071. doi: 10.1145/7351.7478. URL `http://doi.acm.org/10.1145/7351.7478`.

[21] Stephen Blum. Amazon sns vs pubnub: Differences for pub/sub. `https://www.pubnub.com/blog/2014-08-21-amazon-sns-pubnub-differences-pubsub/`, 2014.

[22] Dhruba Borthakur, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, Samuel Rash, Rodrigo Schmidt, and Amitanand Aiyer. Apache hadoop goes realtime at facebook. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD'11*, 2011.

[23] Leo Breiman. Random forests. *Mach. Learn.*, 45(1):5–32, October 2001. ISSN 0885-6125. doi: 10.1023/A:1010933404324. URL `https://doi.org/10.1023/A:1010933404324`.

[24] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. Asm: A code manipulation tool to implement adaptable systems. In *Proceedings of the Adaptable and extensible component systems*, 2002.

[25] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *Queue*, 14(1):10:70–10:93, January 2016. ISSN 1542-7730.

[26] Michael Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2006.

[27] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew M Zhang, and Randy Katz. A case for serverless machine learning. In *Workshop on Systems for ML and Open Source Software at NeurIPS*, 2018.

[28] Joao Carreira, Pedro Fonseca, Alexey Tumanov, Andrew Zhang, and Randy Katz. Cirrus: A serverless framework for end-to-end ml workflows. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '19, page 13–24, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450369732. doi: 10.1145/3357223.3362711.

[29] Gregory V. Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: A comprehensive study. *ACM Comput. Surv.*, 33(4):427–469, 2001.

[30] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. Nv-heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'11*. ACM, 2011.

[31] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the Symposium on Cloud Computing, SoCC'1*. ACM, 2010.

[32] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Symposium on Cloud Computing (SoCC)*, 2010.

[33] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally-Distributed Database. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2012.

[34] Andreia Correia, Pascal Felber, and Pedro Ramalhete. Romulus: Efficient algorithms for persistent transactional memory. In *Proceedings of the Symposium on Parallelism in Algorithms and Architectures, SPAA'18*. ACM, 2018.

[35] Andreia Correia, Pascal Felber, and Pedro Ramalhete. Persistent memory and the rise of universal constructions. In *Proceedings of the EuroSys European Conference on Computer Systems, EuroSys'20*. ACM, 2020.

[36] John Cristy. Imagemagick, 1990. URL `https://imagemagick.org/`.

[37] Databricks. spark-perf. `https://github.com/databricks/spark-perf`, 2014.

[38] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008. ISSN 0001-0782. doi: 10.1145/1327452.1327492. URL `https://doi.org/10.1145/1327452.1327492`.

[39] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of the Symposium on Operating Systems Principles, SOSP'7*. ACM, 2007.

[40] David J. DeWitt and Michael Stonebraker. MapReduce: A major step backwards, 2008. DatabaseColumn Blog. `http://www.databasecolumn.com/2008/01/mapreduce-a-major-step-back.html`.

[41] Tobias Distler, Christopher Bahn, Alysson Bessani, Frank Fischer, and Flavio Junqueira. Extensible distributed coordination. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, pages 10:1–10:16, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3238-5. doi: 10.1145/2741948.2741954. URL `http://doi.acm.org/10.1145/2741948.2741954`.

[42] Vitor Enes, Carlos Baquero, Tuanir França Rezende, Alexey Gotsman, Matthieu Perrin, and Pierre Sutra. State-machine replication for planet-scale systems. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450368827. doi: 10.1145/3342195.3387543. URL `https://doi.org/10.1145/3342195.3387543`.

[43] Vitor Enes, Carlos Baquero, Tuanir França Rezende, Alexey Gotsman, Matthieu Perrin, and Pierre Sutra. State-machine replication for planet-scale systems. `https://github.com/imdea-software/VCD-broadcast`, 2020. retrieved Aug. 2020.

[44] Vitor Enes, Carlos Baquero, Alexey Gotsman, and Pierre Sutra. Efficient replication via timestamp stability (extended version). *CoRR*, abs/2104.01142, 2021. URL `https://arxiv.org/abs/2104.01142`.

[45] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, April 1985. ISSN 0004-5411. doi: 10.1145/3149.214121. URL `http://doi.acm.org/10.1145/3149.214121`.

[46] Brad Fitzpatrick. Distributed caching with memcached. *Linux J.*, 2004(124):5–, August 2004.

[47] Santo Fortunato, Marián Boguñá, Alessandro Flammini, and Filippo Menczer. Approximating pagerank from in-degree. In William Aiello, Andrei Broder, Jeannette Janssen, and Evangelos Milios, editors, *Algorithms and Models for the Web-Graph*, pages 59–71, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-78808-9.

[48] Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*, 2017.

[49] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 475–488, Renton, WA, July 2019. USENIX Association. ISBN 978-1-939133-03-8. URL `https://www.usenix.org/conference/atc19/presentation/fouladi`.

[50] The Apache Software Foundation. Openwhisk composer. `https://github.com/apache/openwhisk-composer`, 2017.

[51] The Apache Software Foundation. Zookeeper barrier recipe, 2019. URL `https://zookeeper.apache.org/doc/current/recipes.html#sc_recipes_eventHandles`.

[52] Tuanir França and Pierre Sutra. Leaderless State-Machine Replication: Specification, Properties, Limits. 2020. to appear.

[53] Felix C. Freiling, Rachid Guerraoui, and Petr Kuznetsov. The failure detector abstraction. *ACM Comput. Surv.*, 43(2):9:1–9:40, February 2011. ISSN 0360-0300. doi: 10.1145/1883612.1883616. URL `http://doi.acm.org/10.1145/1883612.1883616`.

[54] Pedro García López, Marc Sánchez-Artigas, Gerard París, Daniel Barcelona Pons, Álvaro Ruiz Ollobarren, and David Arroyo Pinto. Comparison of faas orchestration systems. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, pages 148–153. IEEE, 2018.

[55] Pedro García-López, Aleksander Slominski, Simon Shillaker, Michael Behrendt, and Barnard Metzler. Serverless end game: Disaggregation enabling transparency, 2020.

[56] Simson L. Garfinkel. An evaluation of amazon's grid computing services: Ec2, s3, and sqs. Technical Report TR-08-07, Harvard Computer Science Group, 2007. URL `http://nrs.harvard.edu/urn-3:HUL.InstRepos:24829568`.

[57] Jerrin Shaji George, Mohit Verma, Rajesh Venkatasubramanian, and Pratap Subrahmanyam. go-pmem: Native support for programming persistent memory in go. In *Proceedings of the USENIX Annual Technical Conference, ATC'20*, 2020.

[58] Seth Gilbert and Nancy A. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.

[59] Markus Goldstein and Seiichi Uchida. A comparative evaluation of unsupervised anomaly detection algorithms for multivariate data. *PLOS ONE*, 11(4):1–31, 04 2016. doi: 10.1371/journal.pone.0152173. URL `https://doi.org/10.1371/journal.pone.0152173`.

[60] Alexey Gotsman, Anatole Lefort, and Gregory V. Chockler. White-box atomic multicast (extended version). *CoRR*, abs/1904.07171, 2019. URL `http://arxiv.org/abs/1904.07171`.

[61] Alexey Gotsman, Anatole Lefort, and Gregory V. Chockler. White-box atomic multicast. `https://github.com/imdea-software/atomic-multicast`, 2020. retrieved Aug. 2020.

[62] Red Hat. Reliable group communication with jgroups. http://jgroups.org/manual/#TOA, 2015.

[63] Joseph M. Hellerstein, Jose M. Faleiro, Joseph Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless computing: One step forward, two steps back. In *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*, 2019. URL http://cidrdb.org/cidr2019/papers/p119-hellerstein-cidr19.pdf.

[64] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Serverless computation with openlambda. In *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'16, pages 33–39, Berkeley, CA, USA, 2016. USENIX Association. URL http://dl.acm.org/citation.cfm?id=3027041.3027047.

[65] Debra Hensgen, Raphael Finkel, and Udi Manber. Two algorithms for barrier synchronization. *Int. J. Parallel Program.*, 17(1):1–17, February 1988. ISSN 0885-7458. doi: 10.1007/BF01379320. URL https://doi.org/10.1007/BF01379320.

[66] C. A. R. Hoare. Monitors: An operating system structuring concept. *Commun. ACM*, 17(10): 549–557, October 1974. ISSN 0001-0782. doi: 10.1145/355620.361161. URL https://doi.org/10.1145/355620.361161.

[67] G. Holmes, A. Donkin, and I. H. Witten. Weka: a machine learning workbench. In *Proceedings of ANZIIS '94 - Australian New Zealnd Intelligent Information Systems Conference*, pages 357–361, 1994.

[68] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX Annual Technical Conference*, USENIX ATC. USENIX Association, 2010.

[69] Tencent Inc. Kdd cup - 2012. https://www.openml.org/d/1220, 2014.

[70] Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. Serving deep learning models in a serverless platform. *CoRR*, abs/1710.08460, 2017. URL http://arxiv.org/abs/1710.08460.

[71] Amos Israeli and Lihu Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC'94, pages 151–160, 1994. doi: 10.1145/197917.198079.

[72] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module, 2019.

[73] Abhinav Jangda, Donald Pinckney, Yuriy Brun, and Arjun Guha. Formal foundations of serverless computing. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019. doi: 10.1145/3360575. URL https://doi.org/10.1145/3360575.

[74] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. Occupy the cloud: Distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC'17, 2017. doi: 10.1145/3127479.3128601.

[75] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Menezes Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph

Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud programming simplified: A berkeley view on serverless computing. Technical Report UCB/EECS-2019-3, EECS Department, University of California, Berkeley, Feb 2019. URL http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.html.

[76] Shannon Joyner, Michael MacCoss, Christina Delimitrou, and Hakim Weatherspoon. Ripple: A practical declarative programming framework for serverless compute, 2020.

[77] Flavio Paiva Junqueira, Benjamin C. Reed, and Marco Serafini. Zab: High-performance broadcast for primary-backup systems. In *International Conference on Dependable Systems and Networks (DSN)*, 2011.

[78] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. SplitFS: Reducing software overhead in file systems for persistent memory. In *Proceedings of the Symposium on Operating Systems Principles, SOSP'19*. ACM, 2019.

[79] Babak Kalantari and André Schiper. *14th International Conference Distributed Computing and Networking*, chapter Addressing the ZooKeeper Synchronization Inefficiency. ICDCN. Springer Berlin Heidelberg, 2013.

[80] Boubacar Kane and Pierre Sutra. Degradability: a Simplified Approach to Data Consistency. https://git.overleaf.com/5e25ea7dcf314c000109a114, 2020. retrieved Aug. 2020.

[81] Boubacar Kane and Pierre Sutra. A library of degradable objects for Apache Cassandra. https://github.com/BoubacarKaneTSP/Application, 2020. retrieved Aug. 2020.

[82] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web. In *29th Annual ACM Symposium on Theory of Computing*, STOC, 1997.

[83] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In *15th European Conference on Object-Oriented Programming*, ECOOP, 2001.

[84] Youngbin Kim and Jimmy Lin. Serverless data analytics with Flint. *CoRR*, abs/1803.06354, 2018. URL http://arxiv.org/abs/1803.06354.

[85] Ana Klimovic, Yawen Wang, Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, and Christos Kozyrakis. Pocket: Elastic ephemeral storage for serverless analytics. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 427–444, Carlsbad, CA, 2018. USENIX Association. ISBN 978-1-931971-47-8. URL https://www.usenix.org/conference/osdi18/presentation/klimovic.

[86] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2), April 2010.

[87] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2), April 2010.

[88] Leslie Lamport. The Part-Time Parliament. *ACM Trans. Comput. Syst.*, 1998.

[89] Leslie Lamport. Generalized Consensus and Paxos. Technical Report MSR-TR-2005-33, Microsoft Research, 2005.

[90] Leslie Lamport. Fast Paxos. *Distributed Computing*, 2006.

[91] Anatole Lefort, Yohan Pipereau, Kwabena Amponsem, Pierre Sutra, and Gaël Thomas. J-NVM: off-heap persistent objects in java. In Robbert van Renesse and Nickolai Zeldovich, editors, *SOSP '21: ACM SIGOPS 28th Symposium on Operating Systems Principles, Virtual Event / Koblenz, Germany, October 26-29, 2021*, pages 408–423. ACM, 2021. doi: 10.1145/3477132.3483579. URL https://doi.org/10.1145/3477132.3483579.

[92] Haifeng Li. Smile. https://haifengl.github.io, 2014.

[93] LightKone. https://www.lightkone.eu.

[94] Linux. Direct access for files. URL https://www.kernel.org/doc/Documentation/filesystems/dax.txt.

[95] B. Liskov, A. Adya, M. Castro, S. Ghemawat, R. Gruber, U. Maheshwari, A. C. Myers, M. Day, and L. Shrira. Safe and efficient sharing of persistent objects in thor. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*. ACM, 1996.

[96] S. Lloyd. Least squares quantization in pcm. *IEEE Transactions on Information Theory*, 28(2): 129–137, March 1982. ISSN 0018-9448. doi: 10.1109/TIT.1982.1056489.

[97] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996. ISBN 1558603484.

[98] Aurèle Mahéo and Pierre Sutra. The Serverless Shell, 2021. URL https://github.com/crucial-project/serverless-shell.

[99] Aurèle Mahéo, Pierre Sutra, and Tristan Tarrant. The serverless shell. In Kaiwen Zhang, Abdelouahed Gherbi, Nalini Venkatasubramanian, and Luís Veiga, editors, *Middleware '21: Proceedings of the 22nd International Middleware Conference: Industrial Track, Virtual Event / Québec City, Canada, December 6 - 10, 2021*, pages 9–15. ACM, 2021. doi: 10.1145/3491084.3491426. URL https://doi.org/10.1145/3491084.3491426.

[100] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 218–233, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350853. doi: 10.1145/3132747.3132763. URL https://doi.org/10.1145/3132747.3132763.

[101] Yanhua Mao, Flavio Paiva Junqueira, and Keith Marzullo. Mencius: Building Efficient Replicated State Machine for WANs. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[102] Francesco Marchioni and Manik Surtani. *Infinispan Data Grid Platform*. Packt Publishing Ltd, 2012.

[103] Francesco Marchioni and Manik Surtani. *Infinispan Data Grid Platform*. Packt Publishing Ltd, 2012.

[104] Ofer Matan, Henry S. Baird, Jane Bromley, Christopher J. C. Burges, John S. Denker, Lawrence D. Jackel, Yann Le Cun, Edwin P. D. Pednault, William D. Satterfield, Charles E. Stenard, and Timothy J. Thompson. Reading handwritten digits: A zip code recognition system. *Computer*, 25(7):59–63, July 1992. ISSN 0018-9162. doi: 10.1109/2.144441. URL https://doi.org/10.1109/2.144441.

[105] Amirsaman Memaripour, Joseph Izraelevitz, and Steven Swanson. Pronto: Easy and Fast Persistence for Volatile Data Structures. In *Proceedings of the conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'20*. ACM, 2020.

[106] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. Mllib: Machine learning in apache spark. *Journal of Machine Learning Research*, 17(34):1–7, 2016. URL `http://jmlr.org/papers/v17/15-237.html`.

[107] Microsoft. Azure durable functions. `https://functions.azure.com`, 2016.

[108] Iulian Moraru, David G. Andersen, and Michael Kaminsky. There Is More Consensus in Egalitarian Parliaments. In *Symposium on Operating Systems Principles (SOSP)*, 2013.

[109] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging ai applications. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, pages 561–577, Berkeley, CA, USA, 2018. USENIX Association. ISBN 978-1-931971-47-8. URL `http://dl.acm.org/citation.cfm?id=3291168.3291210`.

[110] Arun C. Murthy, Vinod Kumar Vavilapalli, Doug Eadline, Joseph Niemiec, and Jeff Markham. *Apache Hadoop YARN: Moving beyond MapReduce and Batch Processing with Apache Hadoop 2*. Addison-Wesley Professional, 1st edition, 2014. ISBN 0321934504.

[111] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. SOCK: Rapid task provisioning with serverless-optimized containers. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 57–70, Boston, MA, July 2018. USENIX Association. ISBN 978-1-931971-44-7. URL `https://www.usenix.org/conference/atc18/presentation/oakes`.

[112] Diego Ongaro and John K. Ousterhout. In Search of an Understandable Consensus Algorithm. In *USENIX Annual Technical Conference (USENIX ATC)*, 2014.

[113] Oracle. *Java Native Interface Specification*. Java SE 14 edition, 2020.

[114] Elliott I. Organick. *The Multics System: An Examination of Its Structure*. MIT Press, Cambridge, MA, USA, 1972. ISBN 0-262-15012-3.

[115] Persistent Collections for Java. URL `https://github.com/pmem/pcj`.

[116] Persistent Memory Development Kit, 2018. URL `https://pmem.io/pmdk`.

[117] Google Cloud Platform. Bigquery. `https://cloud.google.com/bigquery/`, 2010. retrieved Aug. 2019.

[118] Google Cloud Platform. Cloud functions. `https://cloud.google.com/functions/`, 2016. retrieved Aug. 2019.

[119] Google Cloud Platform. Cloud composer. `https://cloud.google.com/composer`, 2018. retrieved March. 2020.

[120] Remko Popma. Picocli, 2017. URL `https://picocli.info`.

[121] A. D. Pozzolo, O. Caelen, R. A. Johnson, and G. Bontempi. Calibrating probability with undersampling for unbalanced classification. In *2015 IEEE Symposium Series on Computational Intelligence*, pages 159–166, 2015.

[122] The CloudButton project. D3.3 - Serverless Compute Engine Reference Implementation, 2022.

[123] The CloudButton project. D5.3 - CloudButton Toolkit Reference Implementation, 2022.

[124] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 193–206, Boston, MA, 2019. USENIX Association. ISBN 978-1-931971-49-2. URL https://www.usenix.org/conference/nsdi19/presentation/pu.

[125] Deepti Raghavan, Sadjad Fouladi, Philip Levis, and Matei Zaharia. POSH: A data-aware shell. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 617–631. USENIX Association, July 2020. ISBN 978-1-939133-14-4.

[126] Redis. https://redis.io/, 2009.

[127] Redis. Replication, 2019. URL https://redis.io/topics/replication.

[128] Dennis M. Ritchie and Ken Thompson. The unix time-sharing system. *Commun. ACM*, 17(7): 365–375, July 1974. ISSN 0001-0782.

[129] Haytham Salhi, Feras Odeh, Rabee Nasser, and Adel Taweel. Open source in-memory data grid systems: Benchmarking hazelcast and infinispan. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ICPE '17, page 163–164, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450344043. doi: 10.1145/3030207.3053671. URL https://doi.org/10.1145/3030207.3053671.

[130] Josep Sampé, Gil Vernik, Marc Sánchez-Artigas, and Pedro García-López. Serverless data analytics in the ibm cloud. In *Proceedings of the 19th International Middleware Conference Industry*, Middleware '18, pages 1–8, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-6016-6. doi: 10.1145/3284028.3284029. URL http://doi.acm.org/10.1145/3284028.3284029.

[131] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990. ISSN 0360-0300. doi: 10.1145/98163.98167.

[132] Amazon Web Services. Amazon efs quotas and limits, 2021. URL https://docs.aws.amazon.com/efs/latest/ug/limits.html#limits-fs-specific.

[133] Vaishaal Shankar, Karl Krauth, Qifan Pu, Eric Jonas, Shivaram Venkataraman, Ion Stoica, Benjamin Recht, and Jonathan Ragan-Kelley. numpywren: serverless linear algebra. *CoRR*, abs/1810.09679, 2018. URL http://arxiv.org/abs/1810.09679.

[134] Marc Shapiro and Pierre Sutra. Database consistency models. In *Encyclopedia of Big Data Technologies*. 2019. doi: 10.1007/978-3-319-63962-8_203-1. URL https://doi.org/10.1007/978-3-319-63962-8_203-1.

[135] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Convergent and commutative replicated data types. *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, June 2011.

[136] Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. Convergent and commutative replicated data types. *Bulletin of the EATCS*, 104:67–88, 2011. URL http://eatcs.org/beatcs/index.php/beatcs/article/view/120.

[137] Simon Shillaker and Peter R. Pietzuch. Faasm: Lightweight isolation for efficient stateful serverless computing. *CoRR*, abs/2002.09344, 2020. URL https://arxiv.org/abs/2002.09344.

[138] Sergey Shnitkind. Linkrun, 2019. URL https://github.com/trendsci/linkrun/.

[139] Thomas Shull, Jian Huang, and Josep Torrellas. AutoPersist: An easy-to-use java nvm framework based on reachability. In *Proceedings of the conference on Programming Language Design and Implementation, PLDI'19*. ACM, 2019.

[140] Marc Snir, Steve Otto, Steven Huss-Lederman, David Walker, and Jack Dongarra. *MPI-The Complete Reference, Volume 1: The MPI Core*. MIT Press, Cambridge, MA, USA, 2nd. (revised) edition, 1998. ISBN 0262692155.

[141] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Symp. on Operating Systems Principles*, SOSP '11, pages 385–400, New York, NY, USA, 2011. ISBN 978-1-4503-0977-6. doi: http://doi.acm.org/10.1145/2043556.2043592. URL http://doi.acm.org/10.1145/2043556.2043592.

[142] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Jose M. Faleiro, Joseph E. Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. Cloudburst: Stateful functions-as-a-service, 2020.

[143] Patrick Stuedi, Animesh Trivedi, Jonas Pfefferle, Ana Klimovic, Adrian Schuepbach, and Bernard Metzler. Unification of temporary storage in the nodekernel architecture. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 767–782, Renton, WA, July 2019. USENIX Association. ISBN 978-1-939133-03-8. URL https://www.usenix.org/conference/atc19/presentation/stuedi.

[144] Pierre Sutra. On the correctness of egalitarian paxos. *Inf. Process. Lett.*, 156:105901, 2020. doi: 10.1016/j.ipl.2019.105901. URL https://doi.org/10.1016/j.ipl.2019.105901.

[145] Pierre Sutra, Etienne Riviere, Cristian Cotes, Marc Sánchez-Artigas, Pedro García-López, Emmanuel Bernard, William Burns, and Galder Zamarreno. CRESON: callable and replicated shared objects over nosql. In *37th IEEE International Conference on Distributed Computing Systems*, ICDCS'17, 2017. doi: 10.1109/ICDCS.2017.239.

[146] Ole Tange. GNU parallel: The command-line power tool. *login Usenix Mag.*, 36(1), 2011.

[147] [The Infinispan Team.

[148] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. pages 172–182, Copper Mountain, CO, USA, December 1995. ACM SIGOPS, ACM Press. http://www.acm.org/pubs/articles/proceedings/ops/224056/p172-terry/p172-terry.pdf.

[149] TPC Benchmark B. URL http://www.tpc.org/tpcb.

[150] John A. Trono. A new exercise in concurrency. *SIGCSE Bulletin*, 26(3):8–10, 1994. ISSN 0097-8418. doi: 10.1145/187387.187391.

[151] Kenton Varda. Protocol buffers: Google's data interchange format. Technical report, Google, 6 2008. URL http://google-opensource.blogspot.com/2008/07/protocol-buffers-googles-data.html.

[152] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'11*. ACM, 2011.

[153] Ao Wang, Jingyuan Zhang, Xiaolong Ma, Ali Anwar, Lukas Rupprecht, Dimitrios Skourtis, Vasily Tarasov, Feng Yan, and Yue Cheng. Infinicache: Exploiting ephemeral serverless functions to build a cost-effective memory cache. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 267–281, Santa Clara, CA, February 2020. USENIX Association. ISBN 978-1-939133-12-0. URL https://www.usenix.org/conference/fast20/presentation/wang-ao.

[154] Hao Wang, Di Niu, and Baochun Li. Distributed machine learning with a serverless architecture. In *IEEE Conference on Computer Communications, INFOCOM 2019*, 2019.

[155] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 133–146, Boston, MA, 2018. USENIX Association. ISBN 978-1-931971-44-7. URL https://www.usenix.org/conference/atc18/presentation/wang-liang.

[156] Jim Webber. A programmatic introduction to neo4j. In *Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity*, SPLASH '12, page 217–218, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450315630. doi: 10.1145/2384716.2384777. URL https://doi.org/10.1145/2384716.2384777.

[157] Chenggang Wu, Vikram Sreekanti, and Joseph M. Hellerstein. Autoscaling tiered cloud storage in anna. *Proc. VLDB Endow.*, 12(6):624–638, February 2019. ISSN 2150-8097. doi: 10.14778/3311880.3311881. URL https://doi.org/10.14778/3311880.3311881.

[158] Mingyu Wu, Ziming Zhao, Haoyu Li, Heting Li, Haibo Chen, Binyu Zang, and Haibing Guan. Espresso: Brewing java for more non-volatility with non-volatile memory. In *Proceedings of the conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS'18*. ACM, 2018.

[159] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. NOVA-Fortis: A fault-tolerant non-volatile main memory file system. In *Proceedings of the Symposium on Operating Systems Principles, SOSP'17*. ACM, 2017.

[160] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 15–28, San Jose, CA, 2012. USENIX. ISBN 978-931971-92-8. URL https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia.

[161] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. Apache spark: A unified engine for big data processing. *Communications of the ACM*, 59(11), 2016.