Chrysaor: Fine-Grained, Fault-Tolerant Cloud-of-Clouds MapReduce

Pedro A. R. S. Costa*†, Fernando M. V. Ramos*†, Miguel Correia*§
Universidade de Lisboa, Portugal*, LaSIGE, Faculdade de Ciências, Portugal†, INESC-ID, Portugal§
palcosta@fc.ul.pt, fvramos@ciencias.ulisboa.pt, miguel.p.correia@tecnico.ulisboa.pt

***Abstract*—MapReduce is a framework for processing large data sets much used in the context of cloud computing. MapReduce implementations like Hadoop can tolerate crashes and file corruptions, but not arbitrary faults. Unfortunately, there is evidence that arbitrary faults do occur and can affect the correctness of MapReduce job executions. Furthermore, many outages of major cloud offerings have been reported, raising concerns about the dependence on a single cloud.**

In this paper we propose a novel execution system that allows to scale out MapReduce computations to a cloud-of-clouds, and tolerate arbitrary faults, malicious faults, and cloud outages. Our system, Chrysaor, is based on a fine-grained replication scheme that tolerates faults at the task level. Our solution has three important properties: it tolerates the above-mentioned classes of faults at reasonable cost; it requires minimal modifications to the users' applications; and it does not involve changes to the Hadoop source code. We performed an extensive evaluation of our system in Amazon EC2, showing that our fine-grained solution is efficient in terms of computation by recovering only faulty tasks. This is achieved without incurring a significant penalty for the baseline case (i.e., without faults) in most workloads.

I. Introduction

Since MapReduce was proposed in 2004 [1], it has been widely adopted in practice. The original implementation created by Google was proprietary, but a few open versions have been developed, among which the popular Hadoop MapReduce [2].

MapReduce is a paradigm that combines distributed and parallel computation with distributed data storage and retrieval. This programming model allows a user to analyze data that resides in a distributed file system using two types of functions: *map* and *reduce*. MapReduce offers means to handle data partitioning, task scheduling, distributed computation, and fault tolerance in a cluster of commodity servers, such as those available in common cloud computing services. Since its origin, it has been used for a variety of tasks, from page ranking [3] to climate research [4], genome analysis [5], [6], astrophysical problems [7], and high-energy physics simulation [8].

The last years have witnessed the requirements for data- and compute-intensive analysis to grow significantly, increasing the need to scale out computation across clouds. However, Hadoop MapReduce supports only execution in a single cloud (or datacenter). Using multiple clouds to store and compute data can bring many benefits. First, it increases resilience by avoiding single points of failure. A user can be made immune to any single cloud availability zone outage by spreading its services across providers. Second, it can improve performance – for example by taking advantage of data locality (bringing the data to a cloud closer to the user), or by leveraging from the computing and network diversity of multiple infrastructures. Third, it can improve security, for instance by exploring the interaction between public and private clouds. A tenant that needs to comply with privacy legislation may demand certain data to be stored in a specific location (e.g., in a private facility). Finally, it may help in reducing costs, by taking advantage of dynamic pricing plans from multiple cloud providers [9]. In this work we provide a solution for doing MapReduce computation on such multi-cloud – or *cloud-of-clouds* – environment, for *fault tolerance*.

Indeed, at scales of thousands of computers, switches, routers, power units and other components, failures are frequent. Therefore, both Google's and Hadoop MapReduce use two mechanisms to tolerate crash faults: they monitor the execution of tasks and reinitialize them in case they stop (thus tolerating crash faults); and they add checksums to files that contain data to detect file corruptions [2], [10]. However, these MapReduce implementations do not handle arbitrary or malicious (Byzantine) faults, nor cloud outages.

Unfortunately, accidental arbitrary faults that may affect the *correctness of the results* have been known to happen, *corrupting the processing* and leading to wrong values. A study on Google datacenters concluded that DRAM errors are more prevalent than previously believed, with more than 8% DIMMs affected by errors yearly, even if protected by error correcting codes (ECC) [11]. Also, a Microsoft study of 1 million consumer PCs showed that CPU and core chipset faults are frequent [12]. A recent study at Facebook provided evidence that more recent DRAM fabrication technologies lead to higher error rates [13]. This shows that the problem, far from being solved, may indeed become more frequent.

Second, *malicious attacks* perpetrated by cloud insiders or external hackers can also cause *corruption of the processing and of its results*. For example, a malicious insider in a cloud that hosts an epidemiological surveillance system can tamper the diagnosis of patients with tragic consequences. A recent report mentions malicious insiders as one of the top threats in cloud computing [14], and alarming instances of this problem have occurred in companies such as Google [15], [16].

Third, cloud outages may lead to the *unavailability of MapReduce instances and data loss*. Experience shows that these events are also frequent, with cases of unavailability of

minutes to days in services like Google Drive or Amazon EC2, to name just a few [17]. Several cases have been reported, including the disruption of one Amazon EC2 datacenter for almost five hours in 2015 [18], and the disruption of the Google Cloud Engine service for some periods in 2016, affecting customers in all regions [19]. Cloud outages can *interrupt the execution of MapReduce jobs*, and the original framework cannot deal with this type of fault as it is restricted to work in a single datacenter.

In this paper we propose a novel MapReduce runtime environment, *Chrysaor*, that scales out MapReduce computations to clouds-of-clouds in order to tolerate arbitrary faults, malicious faults, and cloud outages. Chrysaor is based on a *fine-grained replication* scheme that tolerates faults at the task level. This scheme allows recovering from faults by re-executing only the tasks that were affected. In previous work we have proposed a system, Medusa [20], which shared the goal of allowing MapReduce computations to tolerate the same type of faults. However, Medusa worked at the granularity of MapReduce jobs, resulting in a high cost for fault recovery. For a realistic workload composed of several map and reduce tasks, the single fault in a task would require the whole job to be recomputed in Medusa.

The challenge of achieving the form of fine-grained replication we target in Chrysaor – of tasks, not full jobs – is exponentiated by one of our requirements: not changing the Hadoop source code. Our goal is for our system to use unmodified Hadoop runtimes running in the clouds, including commercial offerings, such as Amazon Elastic MapReduce [21]. As a result, Chrysaor employs a more sophisticated approach that involves the creation of "logical jobs" to enable the re-execution of MapReduce tasks, in order to reduce the cost of recovery from a fault.

Tolerating cloud faults through replication may be considered expensive as one expects these faults to be rare. The reality contradicts this expectation: cloud outages are becoming very common [22], [23]. Moreover, Hadoop MapReduce is increasingly being used for critical applications such as medical research and finance, where any incorrectness or unavailability issues may be unacceptable. Cloud providers seem to share this concern: Amazon recently launched Cross-Region Replication to automatically replicate data in different geographical locations [24]. Motivated by these facts, we believe the cost of replication to be acceptable for such critical applications, in order to guarantee that rare faults with devastating consequences do not occur. Nevertheless, limiting the cost of replication does not cease to be an important goal in our design.

We have performed an extensive experimental evaluation of our system on Amazon EC2. The main conclusion is that Chrysaor is more efficient in terms of computation (number and size of replicas executed) by recovering only faulty tasks, instead of the whole job. This is achieved without incurring a significant penalty for the baseline case (i.e., without faults) in most workloads.

In summary, the main contribution of this work is Chrysaor[1], a system that leverages from several Hadoop MapReduce runtimes spread in different clouds to provide fault-tolerance against arbitrary and malicious faults, and cloud outages. Chrysaor fulfills three additional requirements. First, it requires only minimal modifications to the users' applications. Second, it is based on Hadoop but does not involve modifications to the Hadoop source code. Third, it achieves its goals at a reasonable cost, as our experimental evaluation shows. As a result, with Chrysaor users can outsource their critical computations while being assured that the result is trustworthy.

II. Hadoop MapReduce

This section briefly introduces MapReduce and its implementation in Hadoop, as background for our work. Hadoop is an Apache project that includes the Hadoop kernel, Hadoop MapReduce, and Hadoop Distributed File System (HDFS).

As the term MapReduce suggests, an execution of an application, commonly known as a *job*, consists in executing *map* and *reduce* functions. Many cases of computations that process large amounts of raw data can be solved using this model [1].

A job is executed in two phases, also called *map* and *reduce*. In each phase, the map or the reduce tasks process input data and produce an output. Each part of the job input (a split) is processed by the map function in a map task. Then, the output of the map tasks is partitioned, sorted and transferred to the reduce tasks in a phase called shuffle&sort. Finally, the reduce tasks run the reduce function and save the final output.

In Hadoop, MapReduce jobs are submitted to and managed by a central service called *resource manager* (previously called *JobTracker*). The role of the resource manager is to assign the execution of (map and reduce) tasks to *node managers* (previously called *TaskTrackers*). The framework is composed of the resource and node managers. The resource manager arbitrates the use of resources in the system, and the node manager is responsible for managing containers where tasks run. When a job is submitted, the resource manager reserves resource containers, monitors the nodes, and tracks the progress of the job.

HDFS is the main distributed storage used by MapReduce applications. By default the input files for a MapReduce job reside in HDFS and are divided into blocks that are replicated in a set of hosts for fault tolerance. By default, each map task saves its output in the local file system (to avoid the unnecessary burden of data replication) and the final job output is saved in HDFS (then replicated for fault tolerance).

Hadoop tolerates faults by *(i)* monitoring and restarting tasks when servers, node managers or the tasks themselves crash; and *(ii)* adding checksums to the files in HDFS to detect data corruption in disks. However, these mechanisms only work in a single cloud and deal only with crash faults.

[1]Medusa was a Greek mythology figure that had snakes in place of hair. Chrysaor was one of the sons of Medusa and his name meant "he who bears a golden sword".

III. Preliminaries

This section presents the system model and the problem Chrysaor aims to solve.

A. System model

The system is composed by a set of distributed *processes* (see Figure 1): the *client* that request the execution of job, the *proxy* (also called Chrysaor) that submits the job to the *resource manager*, the *resource manager* that governs the execution of jobs and tasks in a cloud, and a set of *node managers* that execute map and reduce tasks. We do not consider the components of HDFS in the model, as the algorithm is mostly orthogonal to that service. For simplicity we consider that each cloud contains exactly one MapReduce runtime.

We say that a process is *correct* if it follows the algorithm, otherwise we say it is *faulty*. We also use these two words to denominate a task (map or reduce) that, respectively, returns the result of applying the map/reduce function to the input (correct) or some other result (faulty). We assume that clients are always correct, because they are not part of the MapReduce execution. If clients were faulty, the job output would be necessarily incorrect. We also assume that the proxy is always correct because it runs at the client side, e.g., in the same host as the client or in a host under the same administration. Resource managers and node managers can fail arbitrarily: they can return wrong results (e.g., processing corruption, or malicious insider) or even stop executing (e.g., due to a cloud outage). A cloud is faulty if it becomes partitioned from the rest of the processes, it is compromised by a malicious attacker, or suffers an outage.

Our algorithm is configured with two parameters f and t. In distributed fault-tolerant algorithms f is usually the maximum number of faulty replicas, but in our case the meaning of f is different and weaker: f is the maximum number of faulty replicas that can return the same wrong output given the same input. t is the number of faulty clouds that the system tolerates before the service becomes unavailable. The rationale is that f is the maximum number of replicas that can be faulty and still allow the system to find out that the correct result is O. If the system selects the correct output by picking the output returned by $f + 1$ task replicas, it will never select O' because it is returned by at most f replicas. Similarly to the usual parameter f, our f has a probabilistic meaning (hard to quantify): it means that the probability of more than f faulty replicas of the same task returning the same output is negligible.

The other parameter, t, is the maximum number of clouds that may fail arbitrarily (including outages and malicious faults). We assume there are at least $2t + 1$ clouds, to ensure that there are always enough clouds to execute the job.

Chrysaor does not rely on assumptions about processing and communication delays per se. On the contrary, the original MapReduce makes assumptions about such times for termination (e.g., they assume that heartbeat messages from correct node managers do not take indefinitely to be received). Therefore, Chrysaor also makes these assumptions implicitly.

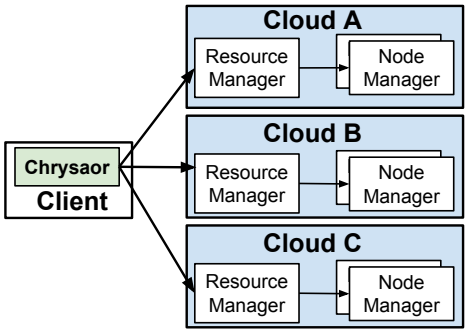


Fig. 1: MapReduce in a multi-cloud system.

We assume that processes are connected by reliable channels, so no messages are lost, duplicated or corrupted. In practice this is provided by TCP connections. We assume the existence of a hash function that is collision-resistant, i.e., it is unfeasible to find two inputs that produce the same output (e.g., SHA-256).

B. Problem formulation

We aim at tolerating *(i)* arbitrary and malicious faults, and *(ii)* cloud outages, when running MapReduce jobs in multi-cloud systems. To tolerate f faults, a basic approach is to create $2f + 1$ replicas of each task (i.e., the task running in each cloud), spread them in $2t + 1$ clouds, and compare the $2f + 1$ outputs of each task. If at least $f + 1$ outputs are identical, the tasks that produced them must be correct and this must be the correct output, due to the definition of f. The proxy can replicate all tasks, verify all outputs, and obtain the correct output of the job.

This basic approach is expensive in terms of computation. Even if there is no fault, each job is executed $2f + 1$ times. Therefore, our objective in this work is to design a proxy that ensures the MapReduce job running in multiple distributed clouds to tolerate cloud faults while *(i)* minimizing the amount of processing; *(ii)* ensuring efficient completion of the entire MapReduce job; *(iii)* and tolerate faults at the task level.

IV. Chrysaor

To tolerate arbitrary/malicious faults and cloud outages, tasks need to be replicated in a few clouds for ensuring the existence of $f + 1$ identical outputs, thus the correctness of the result. Let us first describe briefly how our previous system works [20]. Medusa has a proxy that works similarly to a middleware node in a multi-cloud environment. In that solution, a full job execution is comprised of two phases. The first phase runs a *vanilla MapReduce job* in each cloud, which holds a copy of the data to be processed. The second phase runs a *global MapReduce job* that aggregates the outputs from all clouds to generate the final job result. If insufficient identical results are obtained, additional vanilla MapReduce jobs are executed.

In Chrysaor, the proxy still works as a middleware node, but now it has the advantage of just relaunching failed tasks,

instead of having to relaunch a whole job in case of no majority.

A. Chrysaor overview and logical jobs

Our solution involves defining the concept of *logical job*. To perform fine-grained replication, more specifically to re-execute faulty tasks, we would like Hadoop MapReduce to provide an API to control the execution of a job (e.g., to pause it), but such an API is not available. There would be two ways to solve this limitation: *(i)* by modifying the Hadoop source code to provide such control; *(ii)* or to split a job into parts that from the point of view of Hadoop are still jobs (as Hadoop can only execute jobs). We opted for the latter solution.

Chrysaor executes jobs that are divided in two *logical jobs* that are launched and managed by the Hadoop framework in each cloud. From the Hadoop viewpoint, each logical job is a complete MapReduce job. From the Chrysaor viewpoint, one logical job corresponds to the map tasks of the Chrysaor job, and the other to the reduce tasks. These logical jobs are executed replicated in different clouds. In case a task fails, Chrysaor creates yet another logical job for that failed task and gives it to Hadoop for execution in one or more clouds.

The end of the Chrysaor map phase corresponds to the end of the first logical job, so by default Hadoop would write the output in HDFS. This has a penalty in terms of performance, so in Chrysaor this output is written in a RAM disk (a virtual disk in RAM memory). The second *logical job* will read the stored data and will execute the reduce tasks. Since it is not possible to run a job in Hadoop starting from the reduce tasks (another API limitation), we need to use *identity map tasks* that output the same input data that reduce tasks consume.

During the execution of map and reduce tasks, each task will generate one digest of the output that our application will use to validate the result. In case there are enough equal results, the proxy will consider that the task has ended successfully. In contrast, if there are not enough equal results, Chrysaor creates a new logical job to run just the faulty task(s).

B. Chrysaor operation

Chrysaor has the capability to deal with accidental and malicious faults in the map and reduce tasks. In this section we explain the operation of Chrysaor in steps: first without faults, second with arbitrary faults, and finally with malicious faults. We consider $f = 1$ and $t = 1$ in the examples.

a) Chrysaor without faults: Figure 2 depicts a successful job execution in Chrysaor without faulty tasks. The scenario contains two MapReduce runtimes in two clouds. It assumes that input data is replicated in both clouds before the execution begins.

When the client (not represented in the figure) requests Chrysaor to execute a job, $t + 1 = 2$ clouds (clouds A and B) are selected by the proxy (Chrysaor in the figure) to run the first logical job (step 1). Chrysaor executes $max(f, t) + 1$ replicas of each logical job, which in this case means 2 replicas, one per cloud as $f = t = 1$ (max returns the maximum of two numbers). During execution, each map task

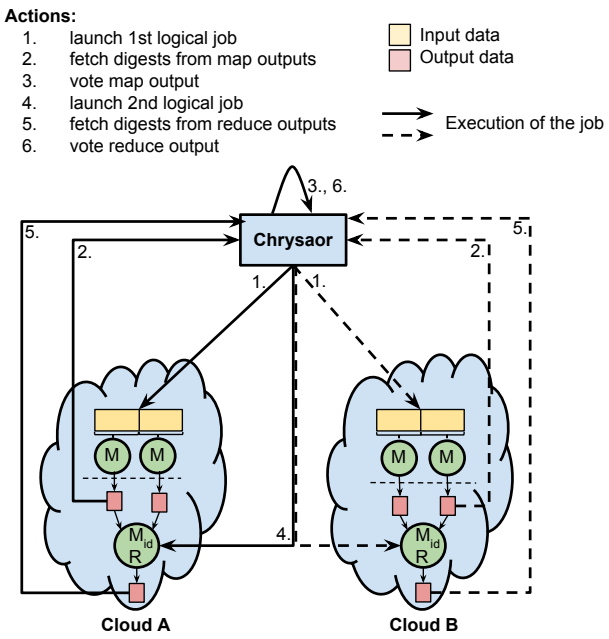


Fig. 2: Chrysaor executing a job in two clouds without faults

creates a digest of the map output. The digests are fetched (step 2) and compared by Chrysaor to check if all map task replicas produced the same results (step 3). This is the case (we are considering no faults), and so the second logical job is launched (step 4).

The second logical job will read the data that was stored previously, run the identity map tasks, and do the shuffle&sort phase before the reduce tasks start. The reduce tasks will produce the final output and the corresponding digest that will be fetched by our system (step 5). Again, Chrysaor will compare the results (step 6). As there are no faults, the results are the same and the job execution terminates successfully.

b) Chrysaor with arbitrary faults: This section explains the case when a map or reduce task returns a wrong output. The alternative case of a task not returning a result at all is not as interesting because it is handled autonomously by the Hadoop runtime: the resource manager of the cloud where the task is being executed simply re-executes the task in the already selected clouds.

Chrysaor detects that the result of a task is wrong when it observes that replicas of the task return different results, i.e., that there are no $max(f, t)+1$ equal results (2 in the example). Notice that it does not know which of the replicas is faulty, only that one of them is faulty as there is disagreement on the result. In that situation, Chrysaor creates a logical job with that task in both clouds and re-executes it.

Chrysaor cannot differentiate if a fault that affected the task was accidental or malicious. If it was accidental, the re-executions may be done in the same clouds: if the faults are intermittent, they will eventually no longer affect the same tasks; if they are permanent, the Hadoop runtime will eventually choose other node managers (and other nodes). If

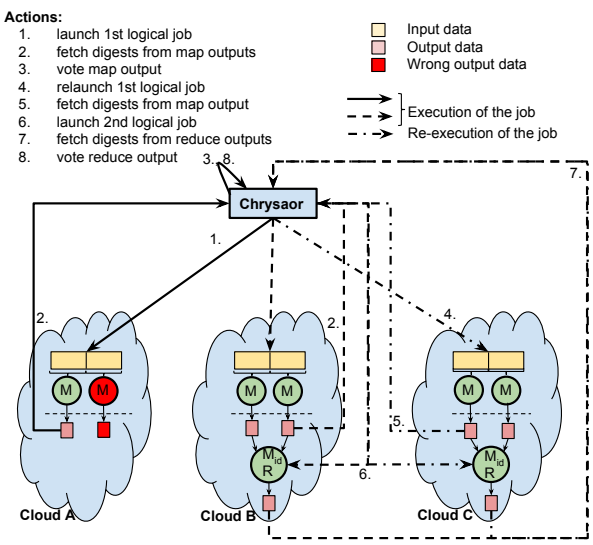


Fig. 3: Chrysaor executing a job in two clouds with a map task re-execution in another cloud due to a fault

the fault is intentional – malicious – it may affect a whole cloud so it is advisable to use another cloud.

Chrysaor allows configuring how to deal with such faults, but the key idea is that it can try a few times to re-execute the tasks in the same clouds, then pick additional clouds if no agreement is reached. Specifically, there is a threshold T_{faults} that is defined by the user. When T_{faults} occur, Chrysaor considers that there may be malicious faults and it picks an additional cloud to execute tasks (next section).

If a faulty result is detected at the end of the second logical job, it is necessary to relaunch the faulty reduce task. This involves re-running not only the reduce task but also the identity map tasks that precede it. The job ends successfully when all reduce task replicas produce the same result.

c) Chrysaor with malicious faults: When Chrysaor is dealing with malicious faults or cloud outages, it has to execute tasks in another cloud until it obtains $max(f,t)+1$ equal results. As already explained, Chrysaor is not able to detect that there is malicious behavior; it simply starts using a new cloud when the threshold T_{faults} is exceeded.

Figure 3 depicts the re-execution of a job when map tasks suffer a malicious fault and T_{faults} is exceeded (e.g., because $T_{faults}=0$). Something similar would happen if there was a cloud outage. Cloud C is going to be used to execute the extra replica. At the end of the first logical job (step 3), there is a task that did not return equal results forcing the system to re-launch the same job in cloud C (step 4). The validation of the digests in cloud C (steps 5, 6) will allow the system to obtain the correct result and show that A may be malicious as it did not provide the correct result. Again, at the end of the execution of the second logical job, Chrysaor reads the digests (step 7). If the execution has ended correctly, the solution as the capability to validate the results and find which cloud is compromised.

In case of a malicious fault at the end of the second logical job, i.e., in the reduce tasks, it is necessary to execute a new full job in the new cloud, and wait to validate the output result. This is the worst case scenario in terms of performance. If more clouds were necessary and were not available, Chrysaor would abort the execution and inform the client.

C. Chrysaor implementation

On the contrary of the rest of Section IV that explains Chrysaor at design level, here we explain its implementation.

The proxy (Chrysaor) is installed in the client machine. It interacts with the resource manager in each cloud using the RabbitMQ message broker [25].

Besides the proxy, Chrysaor provides a Java library that is installed in each resource manager. Although Hadoop supports map and reduce functions written in a few languages, the current implementation of Chrysaor supports only Java. Hadoop is not modified, which leads to each resource manager still being a single point of failure *in its cloud*. The framework is configured to store temporary outputs in RAM disk.

The server-side Java library has the goal of intercepting certain calls done by the executing jobs to the Hadoop API in order to execute Chrysaor server-side code. The calls are intercepted using AspectJ during the execution of the MapReduce job. AspectJ is an aspect-oriented extension to Java that allows, essentially, adding hooks that force calls to external methods in certain conditions, without requiring modifications to the original source code [26].

For an user to take advantage of Chrysaor, she has to do minor modifications to her application. The user does not need to modify the Java code of the map and reduce functions to use Chrysaor. The updates are related to the definition of the identity map function. The user can create his own identity map code, or use the template that is available in the Chrysaor library. The identity map code has to take as inputs a key and values with the same types (classes in Java) as those returned by the job's map function.

Chrysaor intercepts *write* calls that are made inside the map and reduce functions to update the digest for each key and value produced by the task [27]. When the task ends, it invokes the *cleanup* method. The *cleanup* call is intercepted by the Chrysaor server code to save the digest locally. The set of digests produced by the replicated tasks will be used to detect incorrect results.

The job is launched using the *run* method from the *JobClient* interface provided by the Hadoop API. *JobClient* provides facilities to submit jobs and track their progress.

V. Evaluation

The objective of our experimental evaluation is to answer the following questions: (1) How does the performance of Chrysaor compares with its nearest system, Medusa, in a baseline scenario without faults? (Section V-B.1); (2) What is the gain of the fine-grained fault-tolerance introduced in Chrysaor? (Sections V-B.2 and V-B.3); (3) How does the type of job affect Chrysaor's performance?

Section V-A describes the experimental setup as well as the configuration of our solution. Section V-B reports on the performance of Chrysaor, considering both the presence and absence of faults during job execution.

A. Experimental setup

To answer the aforementioned questions we evaluate our solution in a real-world scenario. We have configured a testbed in Amazon EC2 to run all experiments. This service provides a distributed networked infrastructure-as-a-service for computation and storage in the cloud. We considered three common applications provided by Hadoop's Gridmix benchmark [28]: *WordCount*, *WebdataScan*, and *Sort*. This choice aims to ensure application diversity in terms of communication and computation requirements. We have applied to all the applications the slight modifications required to run our system (cf. Section IV-C). We run each experiment 10 times, reporting in the figures the average results, and the 5th and 95th percentiles.

WordCount. The first application we evaluated is related to Web indexing. Running *WordCount* in a multi-cloud system can be considered as building the inverted indexes of a multi-site web search engine for each search site (i.e., cloud). Each cloud runs a local MapReduce job to parse the documents, and to build a search index that supports frequency-inverse document frequency style ranking functions (TF-IDF). This can be achieved by running WordCount as a vanilla MapReduce job in each cloud. To ensure the same search results can be retrieved as in a single-cloud search engine, the input data need to be replicated in each cloud.

WebdataScan. This application has the goal to extract value from big data, an increasingly important tool for decision-making. *WebdataScan* extracts a small amount of relevant data from a large data set, which is a common form of processing and data analysis in many systems. The map tasks keep just a small fraction of the data (0.2%) and the reduce tasks again return just a small part of their input (5%).

Sort. The last application we evaluated does sorting of big data, another typical use of MapReduce. *Sort* is an example of a benchmark that is computationally-intensive (rather than communication-bound). In this application the intermediate key/value pairs are processed in increasing key order. This ordering makes it easy to generate a sorted output file per partition, which is useful when the output file format needs to support efficient random access lookups by key, for instance.

To simulate a real-world scenario, we have set up three clouds located in different Amazon EC2 sites (Oregon, North Virginia, and California). Each cloud is composed of one *resource manager* (master) and four *node managers* (slaves). The hosts are general purpose instances that provide a balance of compute, memory, and network resources. Each instance contains a 2.3 GHz Intel Xeon E5-2686 processor for a total of 8 vCPUS per server. Each server has 12GB of memory, and 150GB of Elastic Block Store (EBS) space. The clouds are protected from the outside world using firewalls, so it is not possible to access them without proper credentials or specific configurations.

We compare the performance of Chrysaor and Medusa in terms of the time it takes to complete the entire job (makespan). Medusa is the only system to tolerate arbitrary and malicious faults, and cloud outages (at the job level). So we use it as a baseline.

Based on the observation that the faults we consider in this paper, despite potentially having devastating consequences, are rare, we consider $f = t = 1$ in our experiments.

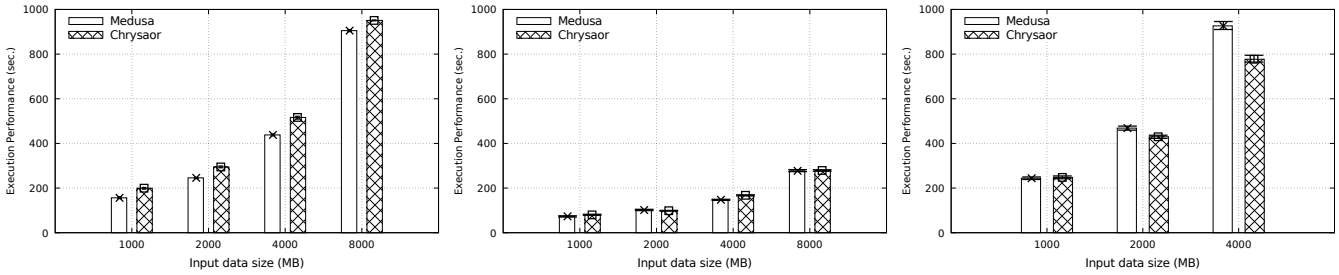
B. Experimental performance

a) Performance without faults: We start by analyzing the performance of Chrysaor without faults. In Figure 4a, we check the performance of the WordCount application with several sizes of input data, ranging from 1GB to 8GB. The choice of these input sizes is based on the fact that the MapReduce jobs that run in Microsoft or Yahoo! production clusters typically operate over input sizes under 14GB [29]. Overall, we see that Medusa got slightly better performance than Chrysaor. The reason is that in WordCount the map tasks produce a map output larger than the input data, so the identity map tasks will compute large data, with a considerable overhead. As result, Chrysaor is 5% to 27% slower than Medusa in the reported cases.

The WebdataScan application is mostly centered in the map side. As such, less time was spent in the second logical job. As consequence, a full execution in Chrysaor spends similar amount of time in the map and reduce tasks as in Medusa. In other words, the identity map tasks and the digests produced while the tasks run did not introduce visible delay as in the WordCount application. Thus, we can see in Figure 4b similar results between both solutions.

In the Sort application (see Figure 4c), Medusa got worse results in comparison to Chrysaor, due to the fact that generating digests after job execution in Medusa delays the entire execution (it involves invoking an HDFS command to access a file). In contrast, the digests are generated while the output is being produced in our new solution. Due to the large output that is produced, the characteristics of Chrysaor wins over Medusa. For instance, in the case of 4GB, Chrysaor was 16% faster. It was not possible to run a use case with 8GB of input data due to lack of memory. Anyway, the trend is clear in showing the advantage of Chrysaor for this type of application. Overall, generating a digest while the output is being produced is shown to be better than generating a digest after the reduce tasks finish. Importantly, this gain would not be made possible in Medusa. It is the new architecture introduced in Chrysaor of a fine-grained approach that allows this optimization.

In summary, the main cost of Chrysaor are the identity maps. As a result, for applications that require identity map tasks that handle large amounts of data, as in the WordCount example, the penalty introduced by our system is non-negligible. In contrast, applications such as WebdataScan that spend less time executing identity map tasks, do not suffer and behave similarly to the baseline case. Finally, the Sort



(a) WordCount **(b)** Webdatascan **(c)** Sort

Fig. 4: Detail of job execution without faults

application is an example of a class of MapReduce jobs where Chrysaor improves over Medusa even in the baseline case without faults. In particular, when the cost of generating digests is high, the advantage introduced by our architecture of enabling this cost to be amortized as the system runs results in an effective gain.

b) Performance with arbitrary faults: In this section, we want to understand the behavior of the system when arbitrary faults occur in the map and reduce tasks. Medusa and Chrysaor behave differently when they are dealing with arbitrary faults. The faults were injected using a configuration setup that tampers randomly the digests of the map or reduce tasks. In the case of Chrysaor we leveraged AspectJ to inject the faults.

Chrysaor is the only that responds immediately to a fault at the task level, so its behavior is different if the fault affects a map or a reduce task. When an arbitrary fault happens in a map task, the corrupted task will be relaunched in the same clouds. When it happens in a reduce task, it is necessary to relaunch again all the identity map tasks to re-execute the faulty task(s). When dealing with arbitrary faults, we consider the threshold T_{faults} to be greater than 1 in order to re-execute the faulty task in the same cloud. Medusa always re-executes the full job when there is a fault, so its performance is the same if the fault compromises a map or a reduce.

Figure 5 depicts the execution time in the case of accidental faults for the three applications. When a fault happens in the map side, *Chrysaor has always the best performance*, up to 56% better than Medusa. Dealing with faults at the task level brings this important benefit.

When we analyze the reduce tasks, we have different results depending on the application. In the case of WordCount (see Figure 5a), Chrysaor was slower when a fault happened in the reduce tasks, whereas in the case of WebdataScan (Figure 5b) we see that both systems reached similar results. In this application, re-executing our solution with the identity tasks took the same time as re-executing the full job. Finally, in the Sort application (Figure 5c), Chrysaor was always faster. Again, in the case of an arbitrary fault generating the digests whilst the output is being produced continues to be a better solution than generating the digest at the end of the job.

One important result from this analysis is the fact that Chrysaor was always faster when the faults affected map tasks. In most MapReduce jobs the number of map tasks (one per input slice) is much larger than the number of reduce tasks, which means that in the common case our solution will outperform Medusa in the presence of arbitrary faults.

c) Performance with malicious faults: When the system deals with a malicious fault, it re-executes the faulty tasks in a new cloud (as at least one cloud is deemed malicious). In this section, the system used $T_{faults} = 0$ in order to use immediately a new cloud without trying first re-execution in the same clouds.

There are different scenarios of execution when we test Chrysaor with malicious faults. In case of a malicious fault in the map side, the algorithm chooses to execute all the map tasks in a new cloud. It is only after this process that the system can check which cloud is compromised, and decide to continue with the correct ones. When a malicious fault is detected in the reduce tasks, a new cloud is chosen to execute the whole job before finding the compromised cloud.

The results are presented in Figure 6. As in the previous section, Chrysaor got the best results when dealing with faults that occurred in the map tasks. Again, our solution can be up to 60% better when compared with Medusa. As before, the results are not as positive when a fault happens in the reduce tasks. Observing Figure 6a, we see that Chrysaor was slower than Medusa. The reason is the need to execute a large identity map task in the new job that runs in the additional cloud.

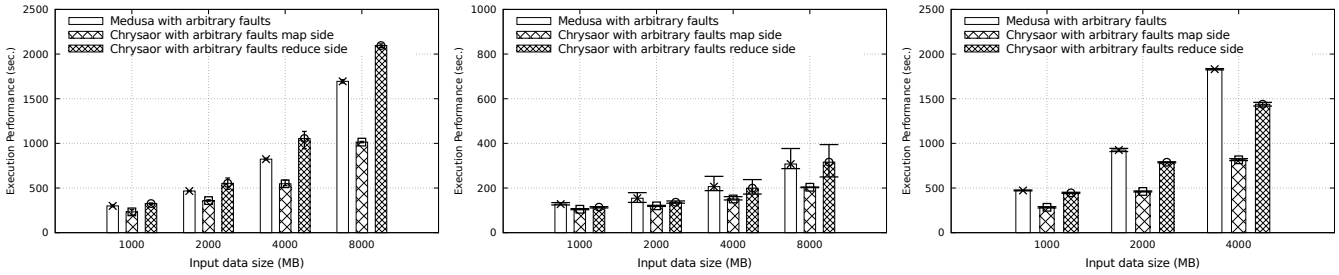
In the WebdataScan application (see Figure 6b), we confirm the similarity of results between Medusa and Chrysaor. This is due to the identity map tasks not being the most important component of the overall execution time.

Finally, in Figure 6c we see that the Sort application had better performance in Chrysaor than in Medusa for all cases. In this case, the Sort application was 2% to 27% faster in Chrysaor.

The conclusions to draw in the experiments with malicious faults is similar to those of the arbitrary case. Namely, Chrysaor is always the best solution when dealing with faults that occur at the map tasks. However, in case of faults in the reduce tasks, Chrysaor is only favorable for workloads that do not involve large execution in the identity step.

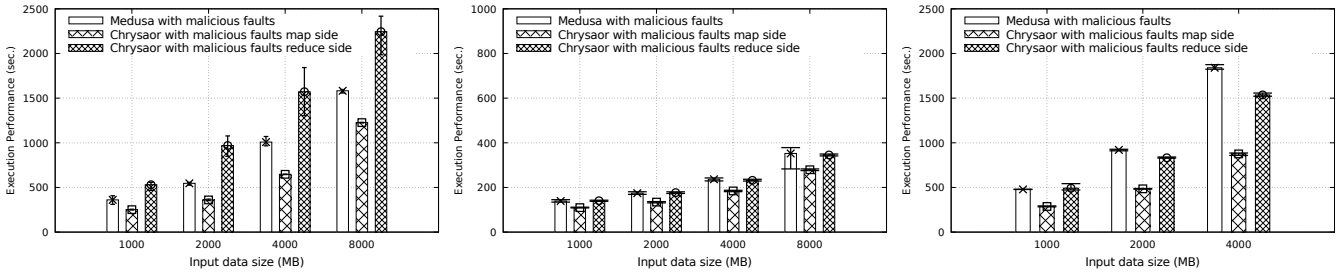
VI. Related Work

There has been much research on MapReduce since the original paper was published in 2004 [1]. The simplicity of the programming model and the effectiveness provided by many



(a) WordCount **(b)** WebdataScan **(c)** Sort

Fig. 5: Detail of job execution with arbitrary faults



(a) WordCount **(b)** WebdataScan **(c)** Sort

Fig. 6: Detail of job execution with malicious faults

implementations of MapReduce attracted a lot of enthusiasm among distributed computing communities. Since then, different lines of work have tried to improve specific features of the MapReduce framework to solve particular problems. For instance, there has been work to support MapReduce on opportunistic [30], dynamic [31], and heterogeneous environments [32], [33], or to perform more complex SQL queries [34] or to manage RDF graphs [35]. These works show how MapReduce simplifies distributed computation and adapts well to large real-world environments.

Alternatives to MapReduce have also appeared [36], [37] to try to enhance its characteristics to enable complex interactions, improve efficiency, and provide a higher level of abstraction. This is useful when we want to deal with more complex scenarios. One example is ClusterBFT, which is a powerful system for Byzantine fault-tolerant data-flow processing in clouds [38]. This system allows the creation of more complex direct acyclic graphs and the replication of subgraphs to reduce the overhead and improve utilization.

A few systems have been proposed that run MapReduce in more than one cloud, although none of them is targeted at tolerating arbitrary faults and/or cloud outages. G-MR is a Hadoop based framework that can run a sequence of MapReduce jobs on geo-distributed data across multiple datacenters [39]. G-MR determines an optimized path to perform a sequence of MapReduce jobs and uses Hadoop MapReduce clusters deployed in each data center. Once the output data is generated in one or more data centers, it is copied to a single destination where it will initiate the aggregate operation and return the final result. G-Hadoop is a MapReduce framework that aims to enable large-scale distributed computing across multiple clusters [40]. This framework replaces HDFS with the Gfarm file system, a network shared file system that can federate local disks of network-connected nodes from several clusters. Users can submit their MapReduce application to G-Hadoop, which executes map and reduce tasks across multiple clusters.

The most similar system to the one proposed in this paper is Medusa [20], our previous proposal that enables scaling out MapReduce computations to multiple clouds and to tolerate the same types of faults we consider here. Similar to Chrysaor, that framework did not require any modification to Hadoop as it simply replicated entire job executions, then compared the outcomes and re-executed the entire job in case a fault was detected. The main differentiating factor of Chrysaor is in handling faults at a more fine-grained level – at the task level. As such, it can react more quickly to faults and avoid the overhead of re-executing the entire job.

VII. Conclusion

We presented a runtime environment called Chrysaor that allows to scale out MapReduce computation to a cloud-of-clouds (or multi-cloud) environment. The motivation is twofold: to tolerate arbitrary and malicious faults that may corrupt the result of MapReduce jobs at the fine granularity of a task, and to tolerate cloud outages and other severe faults in clouds.

Our solution involved the development of a new abstraction – the "logical job" – to obviate the need to modify the Hadoop source code. As such, Chrysaor requires minimal

modifications to the users' applications and does not involve changes to Hadoop.

We compared Chrysaor with the closest alternative – Medusa – to understand the impact of our new architecture. The results from experiments in Amazon EC2 have shown that our fine-grained solution is always better in the most common fault case (a fault in a map task). In addition, despite the unavoidable penalty introduced by not changing Hadoop, our novel design allows performance improvements even in the baseline case for particular workloads.

REFERENCES

[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[2] T. White, *Hadoop: The Definitive Guide*, 1st ed. O'Reilly Media, Inc., 2009.

[3] J. Dean and S. Ghemawat, "MapReduce: A flexible data processing tool," *Communications of the ACM*, vol. 53, no. 1, pp. 72–77, Jan. 2010.

[4] D. Q. Duffy, J. L. Schnase, J. H. Thompson *et al.*, "Preliminary evaluation of MapReduce for high-performance climate data analysis," NASA Goddard Space Flight Center, Tech. Rep., 2012.

[5] A. McKenna, M. Hanna, E. Banks, A. Sivachenko *et al.*, "The genome analysis toolkit: A MapReduce framework for analyzing next-generation DNA sequencing data," *Genome Research*, vol. 20, no. 9, pp. 1297–1303, Sep. 2010.

[6] D. Decap, J. Reumers, C. Herzeel, P. Costanza, and J. Fostier, "Halvade: scalable sequence analysis with MapReduce," *Bioinformatics*, Mar. 2015.

[7] A. Vulpe and M. Frincu, "Exploring scalability in pattern finding in galactic structure using MapReduce," in *Proceedings of the 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, May 2016, pp. 582–587.

[8] G. Brumfiel, "High-energy physics: Down the petabyte highway," *Nature*, vol. 469, no. 7330, pp. 282–283, Jan. 2011.

[9] L. Zheng, C. Joe-Wong, C. W. Tan, M. Chiang, and X. Wang, "How to bid the cloud," in *Proceedings of the 2015 ACM Conference on Data Communication (SIGCOMM)*, 2015, pp. 71–84.

[10] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google file system," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, 2003, pp. 29–43.

[11] B. Schroeder, E. Pinheiro, and W.-D. Weber, "DRAM errors in the wild: A large-scale field study," in *Proceedings of the 11th International Joint Conference on Measurement and Modeling of Computer Systems*, 2009, pp. 193–204.

[12] E. B. Nightingale, J. R. Douceur, and V. Orgovan, "Cycles, cells and platters: An empirical analysisof hardware failures on a million consumer PCs," in *Proceedings of the 6th Conference on Computer Systems (EuroSys)*, 2011, pp. 343–356.

[13] J. Meza, Q. Wu, S. Kumar, and O. Mutlu, "Revisiting memory errors in large-scale production data centers: Analysis and modeling of new trends from the field." in *Proceedings of the IEEE/IFIP 45th International Conference on Dependable Systems and Networks*, 2015, pp. 415–426.

[14] Cloud Security Alliance, "The notorious nine: Cloud computing top threats in 2013," Feb. 2013.

[15] A. Chen, "GCreep: Google engineer stalked teens, spied on chats," http://gawker.com/5637234/gcreep-google-engineer-stalked-teens-spied-on-chats, Sep. 2010.

[16] M. Kandias, N. Virvilis, and D. Gritzalis, "The insider threat in cloud computing," in *Critical Information Infrastructure Security*, ser. LNCS. Springer Berlin Heidelberg, 2013, vol. 6983, pp. 93–103.

[17] CloudSquare, "Cloudsquare service status," https://cloudharmony.com/status-1year-of-storage-and-compute-group-by-regions-and-provider, November 2015, accessed: Nov. 2016.

[18] M. Smolaks, "AWS suffers a five-hour outage in the US," http://www.datacenterdynamics.com/content-tracks/colo-cloud/aws-suffers-a-five-hour-outage-in-the-us/94841.fullarticle.

[19] J. Bort, "Google apologizes for cloud outage that one person describes as a 'comedy of errors'," http://www.businessinsider.com/google-apologizes-for-cloud-outage-2016-4.

[20] P. A. R. S. Costa, X. Bai, F. M. V. Ramos, and M. Correia, "Medusa: An efficient cloud fault-tolerant mapreduce," in *Proceedings of the 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2016, pp. 443–452.

[21] Amazon Web Services Inc, "Amazon Elastic MapReduce," http://aws.amazon.com/elasticmapreduce/.

[22] C. Cerin et al., "Downtime statistics of current cloud solutions," Jun. 2013, the International Working Group on Cloud Computing Resiliency.

[23] G. Clarke, "Microsoft Azure was most FAIL-FILLED cloud of 2014," http://www.theregister.co.uk/2015/01/16/microsoft_worst_cloud_uptime_2014.

[24] Amazon Web Services Inc, "Amazon S3 introduces cross-region replication," https://aws.amazon.com/pt/about-aws/whats-new/2015/03/amazon-s3-introduces-cross-region-replication/, Mar 2015.

[25] A. Videla and J. Williams, *RabbitMQ in Action: Distributed Messaging for Everyone*. Manning Publications Company, 2012.

[26] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of AspectJ," in *Proceedings of the 15th European Conference on Object-Oriented Programming*, 2001, pp. 327–353.

[27] "Mapreduce tutorial," https://hadoop.apache.org/docs/stable/hadoop-mapreduce-client/hadoop-mapreduce-client-core/MapReduceTutorial.html.

[28] "MapReduce 0.22 Documentation – GridMix," https://hadoop.apache.org/docs/r1.2.1/gridmix.html.

[29] A. Rowstron, D. Narayanan, A. Donnelly, G. O'Shea, and A. Douglas, "Nobody ever got fired for using Hadoop on a cluster," in *Proceedings of the 1st International Workshop on Hot Topics in Cloud Data Processing*, 2012, pp. 2:1–2:5.

[30] H. Lin, X. Ma, J. Archuleta, W.-c. Feng, M. Gardner, and Z. Zhang, "MOON: MapReduce on opportunistic environments," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, 2010, pp. 95–106.

[31] F. Marozzo, D. Talia, and P. Trunfio, "P2P-MapReduce: Parallel data processing in dynamic cloud environments," *Journal of Computer and System Sciences*, vol. 78, no. 5, pp. 1382–1402, Sep. 2012.

[32] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving MapReduce performance in heterogeneous environments," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, 2008, pp. 29–42.

[33] D. Cheng, P. Lama, C. Jiang, and X. Zhou, "Towards energy efficiency in heterogeneous Hadoop clusters by adaptive task assignment," in *Proceedings of the IEEE 35th International Conference on Distributed Computing Systems*, 2015, pp. 359–368.

[34] L. Liu, J. Yin, and L. Gao, "Efficient social network data query processing on MapReduce," in *Proceedings of the 5th ACM Workshop on HotPlanet*, 2013, pp. 27–32.

[35] A. Cuzzocrea and R. Buyya, "MapReduce-based algorithms for managing big RDF graphs: State-of-the-art analysis, paradigms, and future directions," in *Proceedings of the 1st IEEE/ACM International Workshop on Distributed Big Data Management*, 2016.

[36] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2007, pp. 59–72.

[37] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand, "CIEL: A universal execution engine for distributed data-flow computing," in *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, 2011, pp. 113–126.

[38] J. J. Stephen and P. Eugster, "Assured cloud-based data analysis with ClusterBFT," in *Proceedings of the 14th ACM/IFIP/USENIX International Conference on Middleware*, ser. LNCS, vol. 8275. Springer, 2013, pp. 82–102.

[39] C. Jayalath, J. J. Stephen, and P. Eugster, "From the cloud to the atmosphere: Running MapReduce across data centers," *IEEE Transactions on Computers*, vol. 63, no. 1, pp. 74–87, 2014.

[40] L. Wang, J. Tao, R. Ranjan, H. Marten, A. Streit, J. Chen, and D. Chen, "G-Hadoop: MapReduce across distributed data centers for data-intensive computing," *Future Generation Computing Systems*, vol. 29, no. 3, pp. 739–750, Mar. 2013.