Reliable Data-Center Scale Computations

Pramod Bhatotia[†] Alexander Wieder[†] Rodrigo Rodrigues[†] Flavio Junqueira[‡] Benjamin Reed[‡]
[†]Max Planck Institute for Software Systems (MPI-SWS) [‡]Yahoo! Research

ABSTRACT

Neither of the two broad classes of fault models considered by traditional fault tolerance techniques – crash and Byzantine faults – suit the environment of systems that run in today's data centers. On the one hand, assuming Byzantine faults is considered overkill due to the assumption of a worst-case adversarial behavior, and the use of other techniques to guard against malicious attacks. On the other hand, the crash fault model is insufficient since it does not capture non-crash faults that may result from a variety of unexpected conditions that are commonplace in this setting. In this paper, we present the case for a more practical approach at handling non-crash (but non-adversarial) faults in data-center scale computations. In this context, we discuss how such problem can be tackled for an important class of data-center scale systems: systems for large-scale processing of data, with a particular focus on the Pig programming framework. Such an approach not only covers a significant fraction of the processing jobs that run in today's data centers, but is potentially applicable to a broader class of applications.

General Terms

Design, Performance, Reliability

Categories and Subject Descriptors

D.4.5 [**Reliability**]: Fault-tolerance; D.4.7 [**Organization and Design**]: Distributed systems

Keywords

Fault detection, Byzantine faults, non-adversarial faults, data center, data processing, Pig

1. INTRODUCTION

In the past decade, users have been offered a number of services that run in the data centers of Internet companies like Google, Yahoo!, or Facebook. The large infrastructure that supports these services is inevitably designed to assume an environment where faults are the norm, and not the exception, not only due to the use of commodity hardware to support these infrastructures, but also because of the sheer scale at which these services operate.

When handling faults in this setting, machines becoming (temporarily or permanently) unreachable is not the only type of faulty behavior that needs to be handled. Other fault modes that cause machines to behave incorrectly, which were often considered too unlikely to require guarding against, become part of the list of concerns at this scale.

Anecdotal reports point to the fact that the designers of these services resort to *ad hoc* mechanisms to prevent some forms of *non-crash faults*. For instance, some systems protect specific components against faults that cause state corruption, e.g., by deploying additional checksums to guard that state [1]. At the same time, designers have stayed away from using more systematic approaches like Byzantine fault tolerance [5], since they are considered costly in terms of performance and administrative overhead [2, 17]. Further, much of this overhead goes to protect from malicious behavior such as compromised clients, even though security infrastructure components such as firewalls or security scans already offer such protection.

But the problem with these non-systematic methods for handling this class of non-crash faults is that they only protect against faulty conditions that the developer provisioned for, and when not all erroneous conditions are captured, the consequences can be dire [1]. Furthermore, this approach imposes a burden on the developers of the infrastructure or the applications who have to manually implement a series of semantic checks.

In this paper, we present the case for a more practical and systematic approach for handling non-crash (but non adversarial) faults in data-center scale computations, and we take some initial steps towards a concrete solution in the context of a specific class of applications. The paper is divided in three main parts. Section 2 argues that a different approach for handling faults is necessary for any system that runs in the environment of a data center. In Section 3, we narrow the scope of the systems we consider to those that perform large-scale data processing. This enables us to highlight more precisely a series of requirements and challenges that this approach has to meet, and additionally some opportunities that can be leveraged to meet these challenges. Finally, in Section 4 we sketch the design of a solution in the context of the Pig [14] programming framework for large-scale data processing. Restricting ourselves to this framework makes it simpler to tackle this problem but also brings two important advantages. First, Pig is an important substrate that is used extensively to run many data-center scale processing jobs. For example, within Yahoo!, approximately 60% of the ad-hoc jobs and 40% of production jobs leverage the Pig platform [13]. This means that a large class of important computations can transparently benefit from our approach. Second, this is a starting point from which we intend to generalize to develop a broad class of fault tolerance techniques.

2. FAULTS IN LARGE-SCALE SYSTEMS

In this section, we discuss the fault patterns in data-center scale systems, and some of the current methods for handling them.

2.1 Instances of Faults

The methods employed to handle faults in the software infrastructure running in data centers focus primarily on handling crash faults, i.e., the silent failure of a data center host that causes it to stop operating.

However, not all faults that occur in data center computations cause machines or software to fail silently. The following are examples of faults that have been reported in data-center scale systems that cause faulty components to behave in various other ways.

Storage System Faults: File and storage systems are prone to faults due to causes that range from the gradual decay of storage media to bugs in disk firmware or software layers above it. While disks already provision against some of these problems, such as ECCs protecting individual disk blocks, some faults can go undetected. For instance, a recent study of storage reliability in a production system [4] reports a surprisingly large prevalence of undetected disk errors (i.e., instances when reading from the disk returns a value that does not correspond to the latest value that was written). In particular, for some drive models, 4% of the drives exhibit this type of errors over the 17 months of this study.

Network Faults: Network channels are notoriously noisy, and networking protocols guard against message corruption by adding redundancy to individual packets. However, there is a chance that the channel noise induces errors in both the original packet and the redundancy, in such a way that an error goes undetected. While checksums are dimensioned to make this occurrence unlikely, there is evidence that data-center services take this possibility into account due to their scale [10].

Memory Corruption: Soft errors or physical defects can also result in the corruption of bits in DRAM, or other parts of the data path. Although some faults can be detected by using ECC-protected DRAM chips, a recent study of detected DRAM errors at Google points to a surprisingly large prevalence of these errors [16]. In particular, this study points to the fact that memory errors detected by ECC are on the order $25,000$ to $70,000$ errors per billion device hours per Mbit and that more than 8% of DRAMs are affected by errors per year.

Software Bugs: The size and complexity of the software infrastructure running in today's data centers makes this infrastructure particularly prone to software bugs. Even if the application code is correct, a bug in the underlying infrastructure might cause that application to produce an incorrect output. For instance, many data-center applications rely upon MapReduce [7] and Pig [14] for deploying large-scale computations. The Apache software bug repository contains numerous instances of bugs in Pig that illustrate this problem. For example, Bug #PIG-1289, caused the implementation of the `Join` operator to return incorrect results for certain inputs, in which case even a correct Pig program might produce a wrong output.

2.2 Handling Faults

When it comes to handling various types of faults, crash faults, being the most common, are well provisioned for by the different components of the software infrastructure running in data centers. For instance, systems such as GFS, HDFS, or ZooKeeper use replication to tolerate crashes from a subset of servers. As another example, the MapReduce scheduler restarts tasks on another machine when it detects that they are no longer responding.

On the other hand, handling non-crash faults is done in a far less systematic way. Some components of the system are protected by additional checksums, *e.g.*, the file system or network transmission [10]. However, it is unclear how to ensure that all components are protected. A recent outage of Amazon's S3 storage service exemplifies this concern. The official description of the problem mentions that messages gossiped among servers had a single bit corrupted, and that the checksums they have in place happened not to protect that particular part of the state [1]. (Curiously, the response to the problem that was announced included deploying more checksums in more parts of the state.)

2.3 Alternative Methods for Non-Crash Faults

An alternative, more systematic way of handling non-crash faults is to employ Byzantine fault tolerance techniques [11, 5]. This class of fault tolerance methods makes no assumptions about the behavior of individual faulty components, which allows them to tolerate any deviation from their correct behavior. Many of these methods work provided there are enough correct components to outvote the faulty ones. Therefore this fault model encompasses not only the types of faults we mentioned above, but also situations such as malicious attacks where nodes may be under control of an adversary.

However, practitioners have often expressed their concerns about BFT techniques, which are often regarded as overkill, partly due to the fact that they need to handle malicious attacks, which are very unlikely in systems running within the security perimeter of the data center. In particular, BFT has been criticized for being too expensive [2], and having too many deployment hurdles, caused by requirements such as extra configuration steps or the need to overhaul the data center infrastructure [17].

Thus we advocate the need to find solutions that can handle the kind of arbitrary, non-crash faulty behavior that we exemplified, and yet are not concerned with the kind of adversarial behavior that BFT is designed to tolerate.

While in this work we do not propose generic solutions to this problem, the remainder of the paper will sketch an approach that works for an important category of systems.

3. CHALLENGES AND OPPORTUNITIES

In this section, we discuss challenges and opportunities that arise from handling the class of faults we motivated. To guide our discussion, we narrow the focus of our problem statement throughout the rest of the paper in two ways.

First, we focus on increasing the reliability of *large-scale data processing* systems. Processing large data sets using commodity hardware with the MapReduce [7] or Pig [14] programming frameworks is commonplace in current data centers. Consequently, such frameworks are part of the critical infrastructure of several Web companies.

The second way in which we focus our discussion is by restricting it to how to provide mechanisms for *fault detection*, since fault detection is the central aspect of any fault handling scheme. For instance, techniques for masking faults, rollback and recovery, or testing all have in common that they often require a fault detection scheme in place in order to trigger a subsequent fault handling mechanism.

3.1 Overhead

The first challenge we list is that a fault detection scheme should incur a modest *overhead*, and this observation is particularly stringent in the case of large-scale data processing computations, which typically require on the order of several machine-days to execute. In this context, the most straightforward way to perform fault detection by replicating the work, is not an attractive proposition.

Some specificities of the types of faults we are trying to detect and the restricted scope of the computations we are considering offer opportunities for savings. For example, the use of techniques such as cryptography to guard against adversarial behavior in BFT protocols is an unnecessary source of processing overhead and administrative overhead to manage keys.

Another opportunity stems from the fact that many of the large-scale data processing jobs that take place in today's data centers make use of programming frameworks that offer languages such as DryadLINQ [19] or Pig Latin [14] with a pre-defined set of constructs for expressing the kind of data processing that can take place. As will become apparent in the next section, this enables us to avoid performing full replication, and instead focus on semantic checks where the redundancy is of the form of a simpler computation over the inputs and the outputs.

3.2 Flexible Resource Utilization

It is not trivial to allocate resources for fault detection. Resource availability in a data center can vary dramatically according to the characteristics of the computation that is executed (*e.g.*, due to the use of synchronization points) and of other computations that share the computing environment.

Thus, in case there are not enough resources available for running a redundant computation that is required for fault detection (or also in the case time-critical tasks where delays are unacceptable) the system should enable decoupling fault detection from the main computation, and running detection in the background. Also, the fault detection system should support an *opportunistic* resource utilization. That is, such a system should be able to use a limited resource budget (*e.g.*, spare resources at low load) to check as much of the computation as possible.

Thus a crucial requirement is that the fault detection system must be able to trade cost for coverage. This is akin to the fault detection schemes that are used when transmitting a message through a noisy channel. In this case, the system can add a small amount of redundancy, e.g., in the form of a checksum or an error-correcting code, and this redundancy can be parameterized to trade cost (how many bits are there in the checksum) for coverage (the maximum number of bit flips that are detectable). An open research question is how to generalize this concept to complex computations whose components can suffer from the types of faults we exemplified before.

```
A = LOAD 'file1' AS (x,y,z);
D = LOAD 'file2' AS (t,u,v);
B = FILTER A by y>0;
C = FILTER B by z<1;
E = JOIN C BY x, D by u;
```

Figure 1: Example Pig program

Here, the fact that we are dealing with non-adversarial faults provides an opportunity to simplify the techniques for trading cost for coverage. E.g., going back to the network transmission example, an adversary could flip the right bits both in the message and in the checksum such that the fault goes undetected. However, since the bits that are corrupted are not chosen by an adversary, one can compute (and bound) the probability of undetected corruption.

3.3 Completeness

A system for fault detection should also be as complete as possible, *i.e.*, have a low false negative rate. While this challenge seems rather obvious, it differs from the approaches that are currently in use – as we discussed, in current systems the developers add checks and checksums to detect faults they anticipate could occur. This approach bears the risk of not detecting potential faults and also puts a heavy burden on the developers. In contrast, an ideal system for fault detection should be complete in the sense that it identifies accurately faults in a computation, even if their cause was not considered by the developer. Furthermore, it should detect the various types of faults we have described before, including situations like deterministic software bugs where traditional replication solutions will fail since all replicas will fail simultaneously.

For this challenge the existence of the previously mentioned programming frameworks also offers an opportunity for introducing some diversity into the system that would be otherwise difficult to achieve. In the next section we discuss how this can be achieved.

3.4 Transparency

The final challenge we point out is that an ideal fault detection system is *transparent* in the sense that it does not require modifications to the application code nor the use of a new programming framework. Again, the next section will demonstrate how the existence of programming frameworks brings an opportunity to automatically deploy the appropriate checks for existing computations that use them.

4. HogPen: TIDY PIG COMPUTATIONS

In this section we sketch the design of HogPen, a fault de-

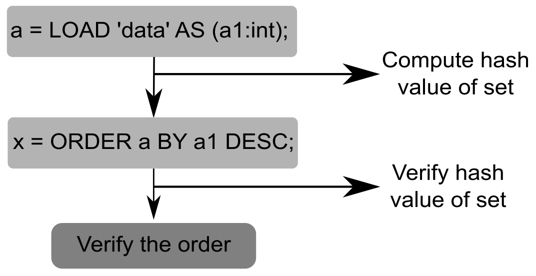

Figure 2: An example of a semantic check for `Order By`

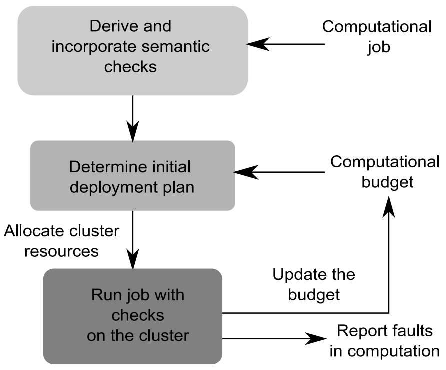
Derive and incorporate semantic checks
Computational job
Determine initial deployment plan
Computational budget
Allocate cluster resources
Run job with checks on the cluster
Update the budget
Report faults in computation

Figure 3: HogPen overview

tection system for the Pig large-scale data processing framework [14].

4.1 Background

Pig is a framework for performing large-scale distributed computations in a cluster environment. It provides a high-level language called Pig Latin [14] in which such computations can be expressed, and the infrastructure to deploy these programs on a cluster.

In a nutshell, Pig Latin enables programmers to express data analysis programs in an imperative language that is embedded with a set of queries akin to SQL or relational algebra. For example, the code in Figure 1, adapted from [8], shows how operators such as `join`, or `filter` are embedded in an imperative program.

Such a program then goes through a series of compilation stages that produce a job that can be executed in a Hadoop MapReduce cluster. Hadoop MapReduce is a framework for parallel processing of large data sets where programmers only specify the implementations of two functions, `map` and `reduce`. The framework then partitions the input data and each part is processed by the map tasks in parallel. Then the output of the map stage is fed into the reduce tasks, partitioned by a series of keys that are output by the map tasks. The Hadoop framework implements the scheduling, monitoring and re-execution of these tasks.

Therefore the compilation of a Pig program produces a series of implementations of the `map` and `reduce` functions, which are then chained together to produce the final computation. The MapReduce jobs are then submitted to Hadoop to be executed.

4.2 Design Overview

The approach taken in HogPen leverages the fact that Pig Latin has a limited set of constructs for large-scale processing, and these have well defined semantics based on relational algebra. This implies that by defining a set of custom semantic checks for each one of these constructs, with the characteristics that we listed in the previous section, we are able to deploy them transparently for detecting faults in the execution of any Pig program.

For example, the execution of an `Order By` operator can be checked as shown in Figure 2. The output of `Order By` needs to be verified for two properties: (i) the output set must be a permutation of the input set, and (ii) the output must be sorted. The semantic check for the `Order By` operator verifies these two properties: (i) the check computes the hash value for the input set and compares it with the hash value of the output set, and (ii) the check also verifies that the output set is sorted. Furthermore, these verifications can be done in linear time. Another example is `Join` operator, which can be checked by comparing the set of tuples in the input tables with the corresponding parts of the output table.

Given a set of parameterizable checks for each operator that is supported by the language, these can be integrated in a complete system. Figure 3 depicts an overview of the HogPen architecture. The system takes the following input: (1) a Pig computation to be executed in a cluster, and (2) a "computational budget" of resources that can be used for checking the computation. Given these inputs, the system automatically derives a set of program specific checks and incorporates them in the original computation. Furthermore, the system deploys these checks alongside the computation automatically. The checks will be executed in an opportunistic manner to maximize coverage with the available computational budget. At run-time, the system reports faults, if any, detected by the deployed checks. These can trigger automatic recovery actions like re-executing a subset of the computation, or present an error report to the user. Finally, the computational budget for running these checks might vary according to the difference between the estimated cost for the computation and its actual cost. Thus the system should enable adapting the level of the checks to changes in the available computation resources.

This high-level description of the system hides many of the challenges in implementing such a system. In the remainder of the section, we describe a few of the main technical hurdles that underlie the implementation of this approach.

4.3 Deploying Semantic Checks

The simplest way to deploy these semantic checks would be to use a source-to-source compilation where each operator would generate a corresponding check that we run alongside. Furthermore, we might think of aggregating multiple operators and devising a single, more efficient check for a set of consecutive operations.

However, Pig performs several optimizations akin to traditional database query optimizations that could draw these checks ineffective. These optimizations result in reordering, deletion, and simplification of Pig operators. This is illustrated by Figure 4, which shows different compilation stages for the example Pig program from Figure 1. In the logical plan, the `Join` operator is applied on two tables. Before executing the `Join` operation, one of the input sets is fil-

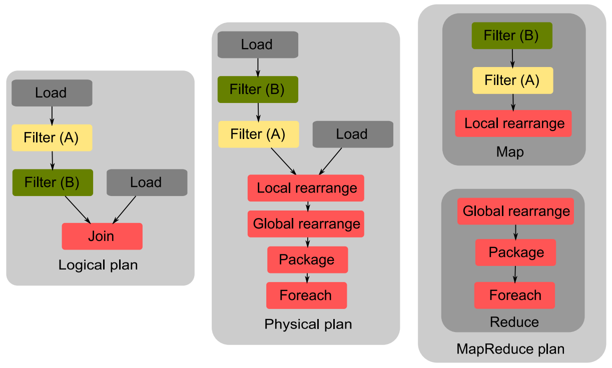

Figure 4: Compilation stages in Pig

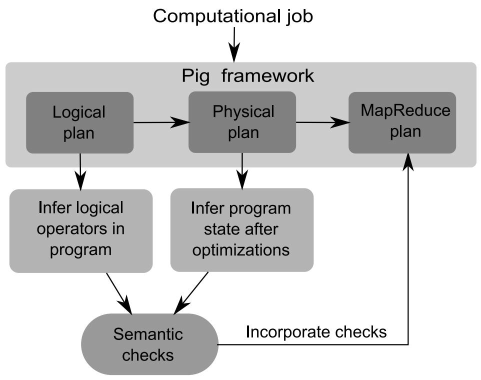

Figure 5: Deploying semantic checks in Pig

tered twice using the `Filter` operator. To build the physical plan, the optimizer swaps the `Filter` operators in the data flow graph to minimize the number of tuples to be processed. The logical operator `Join` is then broken down into a sequence of the physical operators `Local rearrange`, `Global rearrange`, `Package` and `Foreach`. Finally, parts of the physical plan are aggregated into MapReduce computations that are executed on a Hadoop cluster. This illustrates the challenge when associating checks with the logical plan: care must be taken that the subsequent reorganization of the computation does not break their semantics.

We consequently propose to incorporate the semantic checks after the optimization stage completes in order to avoid such interference. To enable this, we have to keep track of how the high-level operators are compiled such that we can insert the operator-specific checks after the optimization. These checks can be inserted into the resulting MapReduce plan as illustrated in Figure 5.

4.4 Opportunistic Checking

As mentioned in the previous section, we intend to enable trading cost for coverage to reduce the cost of checking a computation when the computational budget for redundancy is low. To achieve this flexibility, we intend to incorporate parameterized checks as a function of the available computational budget. These parameterized checks provide a knob that trades cost of checking for the corresponding coverage, resorting to techniques such as sampling. The knob could be used at run-time to perform or skip certain checks based on the available computing resources. For the example shown in Figure 1, the system might be able to check only a subset of the data or the operators based on the priority. The system should also be able to adapt to a varying resource availability at runtime by estimating how cost and coverage vary according to this knob, and finding the best possible coverage for a given cost budget.

While estimating cost is relatively straightforward, an open research question is how to gauge the coverage of a certain check.

5. RELATED WORK

In this section we cover prior work related to handling non-standard failure modes.

Fault models. The most common way to model faulty behavior that does not lead to a silent crash is using the Byzantine fault model that was proposed by Lamport et al. [11]. There have been several proposals for handling Byzantine faults, e.g. by using replication protocols to tolerate a subset of Byzantine replicas [5], or by detecting their occurrence using an auditing approach that checks node behavior against that of a reference implementation [9]. Our work points to the direction of an intermediate fault model, much like the NR-arbitrary model of Abraham *et al.* [3] that captures non-responsive processors and arbitrary corruption of state. However, we need to consider other sources of non-crash faults, which are not captured by this simple model. Clement also mentions the need for a relaxed Byzantine fault model to design a more practical and cost effective fault tolerance system [6]. In his note, he raises attention to the fact that the design choices for such a fault model raise important questions in terms of the trade-off between consistency and safety semantics, liveness guarantees, and restrictions on the model.

Software bugs. Many techniques were proposed to handle software bugs [18], such as finding these bugs using invariant checking, comparing to a reference implementation, or model checking. While software bugs are a source of non-crash faults that we are concerned with, they are not the only one, and finding such problems requires going beyond this set of techniques.

Spread-spectrum computations. Our notion of adding a small, parameterizable amount of redundancy to a computation is related to the work by Murray and Hand [12]. They propose computation dispersal algorithms that add redundancy to some restricted types of computations, hence tolerating crash faults of a subset of the nodes performing each part. This idea could also be adapted to detect faulty computations, albeit restricted to the set of tasks that are suited to this approach.

Arithmetic checks. Schiffel et al. proposed a fault detection mechanism targeted at a much lower level to guard against hardware faults [15]. They develop checks for arithmetic operators based on AN-encoding, and extend a compiler to insert those checks in the application. While the two approaches are similar in spirit, such a low-level approach could impose a high overhead, and can only guard against a more limited set of faults.

6. CONCLUSIONS

We present a preliminary approach to automatically deploy semantic runtime checks for Pig programs to detect arbitrary, non-malicious faults. In contrast to approaches based on replication, such as BFT systems, we can leverage the semantics of Pig to generate language-specific checks that can be significantly less expensive than replication. Additionally, we can take advantage of a non-adversarial setting to minimize costs in comparison to a BFT system, or to trade cost for coverage.

Our proposal is of practical significance given the prevalence of data processing frameworks like Pig/Hadoop in production settings of large Web companies, and the business value of the computations they support. But at the same time our proposal opens a variety of interesting research questions that we intend to tackle in the future. In particular, we left open the issue of defining a precise fault model, and such definition might have interesting implications for the design of the system. In particular, when we enable trading cost for coverage, the notion of coverage (i.e., what fraction of faults can we detect if we only check a subset of the computation) is intertwined with the definition of a realistic and precise fault model. Another related direction that we intend to explore is the possibility of choosing which parts of the computation to check based on the impact that a fault might have on the final computation result. Finally, generalizing this approach and applying it to other components of the infrastructure that comprises the data center back-ends of Internet companies is an interesting avenue of future work.

7. REFERENCES

[1] Amazon S3 Availability Event. `http://status.aws.amazon.com/s3-20080720.html`.

[2] Things that scale. `http://www.thingsthatscale.com/2008/09/large-scale-distributed-systems-and-middleware-ladis/`.

[3] Abraham, I., Chockler, G., Keidar, I., and Malkhi, D. Byzantine disk paxos: optimal resilience with byzantine shared memory. In *Distributed Computing* (2004), ACM Press, pp. 387–408.

[4] Bairavasundaram, L. N., Goodson, G. R., Schroeder, B., Arpaci-dusseau, A. C., and Arpaci-dusseau, R. H. An analysis of data corruption in the storage stack. In *Proceedings of the 6th USENIX Symposium on File and Storage Technologies (FAST)* (2008).

[5] Castro, M., and Liskov, B. Practical byzantine fault tolerance. In *OSDI '99: Proceedings of the third symposium on Operating systems design and implementation* (1999), pp. 173–186.

[6] Clement, A. Distributed computing in sosp and osdi. *SIGACT News 39*, 2 (2008), 84–91.

[7] Dean, J., and Ghemawat, S. Mapreduce: simplified data processing on large clusters. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation* (2004), pp. 10–10.

[8] Gates, A. F., Natkovich, O., Chopra, S., Kamath, P., Narayanamurthy, S. M., Olston, C., Reed, B., Srinivasan, S., and Srivastava, U. Building a high-level dataflow system on top of map-reduce: the pig experience. *Proc. VLDB Endow. 2*, 2 (2009), 1414–1425.

[9] Haeberlen, A., Kuznetsov, P., and Druschel, P. PeerReview: Practical accountability for distributed systems. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)* (Oct 2007).

[10] Isard, M. Autopilot: automatic data center management. *SIGOPS Oper. Syst. Rev. 41*, 2 (2007), 60–67.

[11] Lamport, L., Shostak, R., and Pease, M. The byzantine generals problem. *ACM Trans. Program. Lang. Syst. 4*, 3 (1982), 382–401.

[12] Murray, D. G., and Hand, S. Spread-spectrum computation. In *HotDep '08: Proceedings of the Fourth Workshop on Hot Topics in Syetms Dependability* (2008).

[13] Olston, C. Pig: Web-scale data processing. `http://infolab.stanford.edu/~olston/pig.pdf`.

[14] Olston, C., Reed, B., Srivastava, U., Kumar, R., and Tomkins, A. Pig latin: a not-so-foreign language for data processing. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data* (2008), pp. 1099–1110.

[15] Schiffel, U., Süsskraut, M., and Fetzer, C. An-encoding compiler: Building safety-critical systems with commodity hardware. In *The 28th International Conference on Computer Safety, Reliability and Security (SafeComp 2009)* (2009).

[16] Schroeder, B., Pinheiro, E., and Weber, W.-D. Dram errors in the wild: a large-scale field study. In *SIGMETRICS '09: Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems* (2009), pp. 193–204.

[17] Song, Y. J., Junqueira, F., and Reed, B. BFT for the skeptics. In *BFTW3: Affiliated with DISC* (2009).

[18] Wenchao, Z. Fault management in distributed systems. Tech. Rep. MS-CIS-10-03, University of Pennsylvania Department of Computer and Information Science, 2010.

[19] Yu, Y., Isard, M., Fetterly, D., Budiu, M., Úlfar Erlingsson, Kumar, P., and Currey, G. J. Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI '08: 8th USENIX Symposium on Operating Systems Design and Implementation* (2008).