Efficient Fault Tolerance for Massively Parallel Dataflow Systems

Sergey Dudoladov
supervised by Prof. Volker Markl
Technische Universität Berlin
firstname.lastname@tu-berlin.de

Abstract

Dataflow systems provide fault tolerance by combining checkpointing and lineage but leave it up to a data scientist to decide on when and how to checkpoint. This leads to job plans that are inefficient during failure-free execution or recovery, e.g., if a data scientist forgets to checkpoint expensive operators that need to be re-executed after a failure. In this work, we aim to (1) increase efficiency of checkpointing transparently to the data scientist and (2) automate placement of checkpoints and other fault tolerance mechanism. First, we show how to reduce checkpoint size for machine learning algorithms using *qpoints*, a compressed representation of the algorithms' parameters. Qpoints enable the algorithms to run faster by spending less time on checkpointing. Second, we show how to place checkpoints optimally for a given cluster without user intervention using *smartpoints*, our framework for building fault tolerance optimizers. Smartpoints free data scientists from making tedious decisions about fault tolerance while retaining reasonable performance guarantees in case of failure.

1. INTRODUCTION

The interest in Big Data has advanced the development of dataflow systems for large scale analytics such as Apache Flink and Apache Spark. Users run these systems in either private or cloud-based clusters that are often virtualized. Such clusters tend to be failure prone: the commodity hardware used to build them exhibits high failure rates when used in large quantities [11].

Another source of failures for cloud-based clusters, regardless of their size, is preemption. Cloud providers offer certain instances of virtual machines with large discounts to utilize otherwise idle hardware [2, 15]. The downside is that the cloud can *preempt* (reclaim) such an instance at any moment when it needs the resources back. For an application, preemption looks like a failure: the preempted instance disappears, causing the loss of the application state kept in memory. In this paper, we use preemption as a running example; our research project applies to other failure models as well, for instance, classic fail-stop failures.

To handle preemption, cloud providers advise on employing *rollback recovery* [3]. This approach to fault tolerance is conceptually simple: a system periodically checkpoints the current result of computation to durable storage, and, in the case of preemption, restarts the computation from the most recent checkpoint.

This scheme, while widely used, is not cost effective when it comes to modern applications such as machine learning (ML). We identify two problems that make cloud clients waste money. First, checkpoints waste resources users pay for. Persisting the application state keeps preemptible instances busy performing non-productive work, which increases expenses users want to minimize. Given that the state of an ML algorithm may contain tens of gigabytes of data [9], this may cause considerable overhead both in terms of resources and money. Avoiding checkpoints is not an option either, because recomputing application state from scratch after each preemption delays the results. Even worse, an application without checkpoints may fail to terminate under high preemption rates.

The second, arguably more general, issue is that fault tolerance requires manual tuning. For example, Apache Spark offers eleven persistence options to tweak its lineage-based version of rollback recovery [1]; examples include checkpointing to memory of two cluster nodes or storing the dataset entirely on disk of a single machine. Making end users manually tune numerous fault tolerance options - such as the placement or type of checkpoints in Spark - is likely to be problematic because it requires understanding of the system's internals. In the case of Spark, a data scientist has to decide where a lineage chain becomes too expensive to recompute (e.g., because of too many CPU-intensive operators) and insert a checkpoint of a suitable type. Given that many users of data analytics systems are in fact non experts in distributed systems [12], such performance tweaking becomes ineffective [4], i.e., leads to job plans that take more time than necessary to (re)compute. This extra time combined with the working time of data scientists, who tweak fault tolerance, also increases expenses of cloud clients.

In this research project, we aim to make fault tolerance for dataflow systems more efficient and usable. We propose:

1. *Qpoints*, a technique for reducing checkpointing time by persisting the compressed state of ML algorithms.
2. *Smartpoints*, a framework for building fault tolerance optimizers that aims to automate decisions on when and how to checkpoint.

2. QPOINTS

In Section 2.1, we describe the robustness of machine learning algorithms against approximation of parameter values, a property vital for qpoints. In Section 2.2, we employ the property to reduce size of checkpoints for ML by compressing them into qpoints.

2.1 Approximation in Machine Learning

Many machine learning algorithms are robust against approximation, that is, they can tolerate partial loss of state or some inaccuracy in parameter values [7]. In this research project, we will

Proceedings of the VLDB 2016 PhD Workshop, September 9, 2016. New Delhi, India.

exploit the robustness to approximate parameter values of several practically important classes of algorithms such as deep neural networks and generalized linear models.

For execution speed and coding convenience, practitioners encode parameters such as weights in the generalized linear models as IEEE 754 double values, but the experimental evaluation of ML algorithms shows that parameter values tend to cluster near zero, so the full range of a double value is rarely used [13]. Given that large scale ML algorithms can contain billions of parameters [9], suboptimal parameter representation can significantly increase the memory footprint of an algorithm.

Consider the recent Adam project[1] for deep learning [9]. This system is capable of training a neural network with 36 billion connections between neurons where each connection is represented with a weight. Assuming an IEEE double representation of each weight, this amounts to $36 * 10^9 * 8$ bytes of data, so 288 gigabytes are needed to represent only the parameters of a single model.

Machine learning practitioners recognize the need to use less space per parameter [10, 13, 16]. Instead of the IEEE 754 floating point format, they propose to use the $Qn.m$ encoding (Q for quantization), which can significantly reduce the memory footprint with little loss in performance. This encoding represents a real number with $n + m + 1$ bits: n bits for the integral part, m bits for the fractional part and one bit for the sign [13]. For example, in [13] the authors use $Q2.13$ encoding in the training phase to halve the logistic regression memory consumption compared with the model that represents parameters as floats. This saving costs only about 0.01% extra logistic loss in the testing phase.

2.2 Qpoints: Checkpoints in Qn.m Encoding

The robustness of ML to approximation hints that in many cases checkpointing the exact in-memory state may not be needed: an algorithm neither uses most of this memory footprint during normal operation, nor does it need the full state to recover after preemption.

Since $Qn.m$ encoding can save noticeable amount of memory without sacrificing the end performance, we suggest to checkpoint this representation of ML models parameters instead of the standard representation with doubles. We define a *qpoint* as a checkpoint that employs $Qn.m$ encoding to compress and persist model parameters. For speed and convenience data scientists can still use doubles while developing algorithms; a system should automatically compress parameters while checkpointing.

Qpoints come with a cost: unlike checkpoints, qpoints are not universally applicable. For qpoints to be beneficial, an ML algorithm should (1) have large number of parameters of limited range, (2) be able to terminate from approximate state after recovery, and (3) lack additional consistency requirements.

The first condition means that algorithm parameters do not use the full range of IEEE 754 double data type. This condition holds for at least two classes of machine learning algorithms: generalized linear models and deep neural networks [10, 13, 16]. Generalized linear models are commonly used at scale for tasks like spam filtering or predicting ad click-through rates, e.g., [5, 14]. Deep neural networks are used for tasks such as visual object recognition where human experts struggle to extract meaningful features from the input data [9].

The second condition selects algorithms that are can reach the desired level of performance starting from approximate state. For both neural nets and linear models, low precision parameter representation – about a quarter of the bits of the IEEE double format – is sufficient for training and running models and has little effect on the prediction accuracy [13, 16]. The guarantees, however, differ: linear models and convex optimizers in general provably reach the unique optimum from any approximation, while neural networks may converge to different solutions due to their non-convex nature.

The third condition rules out algorithms that require approximate state to be consistent. An example of such an algorithm is PageRank. This algorithm can converge using approximate parameter weights, but it requires the weights to form a probability distribution, i.e., to sum up to one. Simply restoring the weights from a qpoint may be insufficient because, due to rounding issues, the sum of parameter values may deviate too far from one. Both neural nets and linear models have no such special consistency requirements.

Despite these restrictions, qpoints bring two significant gains. First, qpoints can reduce the checkpointing overhead transparently to a user without sacrificing fault tolerance. The standard way to decrease checkpointing cost is to adjust frequency of checkpoints [22]. This strategy requires manual tuning from a user, and proved problematic in real world deployments of ML algorithms (e.g., [18]). Qpoints can reduce the time spent on fault tolerance by saving less data per checkpoint: data scientists can run their algorithms with some default qpoint frequency and gain sufficient fault tolerance without paying the cost of full checkpoints.

Second, qpoints enable exploring a failure model that, to our knowledge, is not currently discussed within the database community, namely *a failure with a prior warning*. Such failures correspond to preemption in clouds that notify a client about the upcoming preemption event. For instance, Google Cloud issues such warning 30 seconds before the preemption [15]. After getting this notification, a client would naturally want to persist the current computation state. However, starting standard checkpointing at this moment may fail to meet the hard time limit because of the data volume to persist. Qpoints have more chances to meet the limit because they have less data to save. Qpoints also enable *progressive checkpointing*, that is, saving coarse approximation of parameters with as little data as possible and then gradually refining the approximation up until the preemption takes place.

3. SMARTPOINTS

In Section 3.1, we outline the randomized weighted majority, the meta-algorithm we use to develop smartpoints. In Section 3.2, we describe smartpoints, our framework for building fault tolerance optimizers.

3.1 Optimal Prediction from Expert Advice

Consider the following situation: one has a pool of experts who need to predict future events, for example, if the price of a single stock will go up or down next day. Naturally, one would like to select the best expert - the one who makes the least amount of mistakes - and follow their predictions. The problem is, it is not known beforehand which expert will perform best on a given sequence of future events, e.g., days. In such setting, the randomized weighted majority algorithm (RWM) enables to perform provably close to the best expert without any apriori knowledge.

The RWM assigns equal initial weights to all experts, each expert essentially being a function with the $\{0, 1\}$ range (say, 0 means the price will go down and 1 means it will go up). The algorithm then proceeds in a sequence of *trials*. At each trial, the algorithm chooses an expert at random, with probability proportional to the current weight of the expert, and follows the prediction of this expert. Once the true answer is revealed (e.g., the price went down), the algorithm punishes all experts who predicted wrongly by multiplying their weights by the penalty β, $0 \leq \beta < 1$; weights of the correct experts stay intact. By doing so, the algorithm decreases

[1]Adam is technically not a dataflow system. However, current projects such as SparkNet [19] actively port deep learning to dataflow systems. So these systems can be expected to run into the problem of deep learning memory requirements in the near future.

the probability of choosing a mistaken expert in the next trial. Intuitively, if an expert predicts wrongly, the algorithm trusts them less in the future. The RWM can guarantee the following property [6]:

Theorem 1. *On any sequence of trials, the expected number of mistakes X made by the Randomized Weighted Majority algorithm satisfies:*

$$X \leq x + \ln(y) + \mathcal{O}(\sqrt{x \ln(y)})$$

where x is the number of mistakes of the expert who performed best so far, and y is the total number of experts.

So, the expected number of mistakes of the algorithm is bounded by the number of mistakes of the expert from the pool who performs best on a given sequence of trials. Intuitively, *the RWM overall performance is close to that of the best expert in the pool.* The bound from Theorem 1 holds for the worst case of input data without any probabilistic assumptions about the input or experts [17].

3.2 Smartpoints: Fault Tolerance via Randomized Weighted Majority

Fault tolerance mechanisms span a large spectrum ranging from usual checkpoints [3] to lineage [26] to less standard ideas such as qpoints. With that variety, choosing among the mechanism becomes non-trivial even for expert users. Researches are well-aware of this problem: the recent work has shown that automatically choosing the most suitable mechanism (e.g., checkpoints) and its placement (for instance, checkpoint each third operator) is possible and does improve performance [23, 25]. We observe, however, that current approaches to fault tolerance optimization – such as the ones discussed in [20, 24, 25] – share common shortcomings.

First, they require significant implementation effort, often at the system runtime level. An example is the FTOpt optimizer [23], which requires a system to have a special acknowledgement protocol to track tuples' flow through the system; reimplementing this protocol would greatly complicate system design and increase development costs.

Second, current approaches tend to make assumptions that may be difficult to fulfill. For example, optimizers proposed in [21, 23] depend on accurate cost estimates that are hard to obtain in the presence of user-defined functions [4].

In this research project, we propose an approach to building fault tolerance optimizers with RWM that alleviates these problems. We define a *fault tolerance policy* to be a set of decisions on where to use which fault tolerance mechanism. Continuing our running example of preemption, a policy can be a heuristic that advises on checkpointing before the end of each hour. To simplify terminology, we will henceforth use the term checkpoint to denote any fault tolerance mechanism, for instance, qpoints. Smartpoints as a framework should be capable of incorporating such mechanisms.

Intuitively, in our approach fault tolerance policies become experts who periodically vote according to the rules of RWM if a system should checkpoint or not. For instance, the "checkpoint nothing" policy would always vote against checkpointing and rather rely on job re-execution to provide fault tolerance. The RWM ensures that the most suitable policy for a given environment will eventually retain most weight. For example, the "checkpoint nothing" policy should win under low preemption rates, because there rare preemption events do not justify the cost of checkpointing.

Thus *a dataflow system with RWM-based fault tolerance will eventually employ the fault tolerance policy most suitable for its particular cluster without any user involvement* at the cost of few initial mistakes. For brevity, we use the term *smartpoints* to refer to the idea of using RWM to select the best fault tolerance policy.

The Algorithm 1 describes smartpoints more formally. In this algorithm, fault tolerance policies effectively predict failures by their votes. That is, the decision to checkpoint can be rephrased as "the next operator will fail": if the next operator does not fail, we do not need to checkpoint. If one of the subsequent operators (e.g., the second next) fails, we ideally would like to checkpoint the immediate predecessor of the operator-to-fail to avoid re-executing any successful operators. Real failures become true labels used by RWM to penalize experts: if a policy voted to checkpoint, and a failure did not happen during the next operator, the algorithm reduces the weight of this policy. For the purposes of smartpoints, we define a RWM *trial* to consists of (a) an execution of a single operator, (b) a vote among checkpoint policies if the operator output should be checkpointed, and (c) observing if a failure happens during the execution of the next operator.

This algorithm ensures three properties. First, due to the properties of RWM (see Theorem 1), for a given pool of policies Algorithm 1 checkpoints in a way provably close to the policy optimal for a given cluster. In other words, assuming a well-designed pool of policies, smartpoints can automatically adapt to a wide range of cluster environments. Second, the algorithm makes very few assumptions: it does not require any specific knowledge about failure distributions or cluster size or previous history of a system. Instead, we propose to encode this domain-specific knowledge into fault tolerance policies unique for a particular system. Finally, the reuse of elements common in dataflow systems (checkpoints, blocking operators) combined with the simplicity of the algorithm itself greatly reduces the implementation effort compared to existing fault-tolerance optimizers. For example, nothing in the algorithm itself requires a special support from the runtime.

Algorithm 1 Smartpoints

```
1: for $each\ policy\ i$ do
2:     $w_i = 1$
3: for $each\ operator\ t$ do
4:     $Use\ policy\ i\ prediction\ with\ probability\ p_i = \frac{w_i}{\sum_j w_j}$
5:     for $each\ policy\ i$ do
6:         if $policy\ i\ made\ a\ mistake$ then
7:             $w_i = \beta * w_i$
```

4. RELATED WORK

Approximation in Machine Learning. The work of Bousquet and Bottou [7] provided the theoretical foundation for understanding this phenomenon; a series of recent papers [10, 13, 16] studied the effect of approximating parameter values with $Qn.m$ encoding on the prediction accuracy of deep neural networks and generalized linear models. We plan to piggyback on this approximation tolerance to decrease checkpoint size with qpoints.

Randomized Weighted Majority. Littlestone and Warmuth proposed the original idea and later summarized it in [17]; the follow up work [8] conducted extensive theoretical analysis and showed how to choose the penalty parameter β to minimize the expected number of mistakes of the algorithm. The paper by Blum [6] provides an overview of the key results in the area. We adopt these results to develop the framework of smartpoints, which should yield a family of fault tolerance optimizers capable of intelligently choosing the optimal checkpoint policy at runtime.

Fault tolerance optimization for dataflows. The FTOpt optimizer proposed in [23] employs geometric programming to reduce the overhead of checkpoints. Smartpoints differ from it in three ways. First, they do not require a cost model and cost estimates for operators. Second, they do not require dedicated support from the runtime (FTOpt requires an ack protocol to track tuples). And third, smartpoints do not restrict job plans in any way besides forming a directed acyclic graph of operators (FTOpt handles only job plans with aggregations at the top). The more recent optimizer from [21] probabilistically models the likelihood and impact of failures using yet another cost model. Unlike [21], which assumes the Poisson distribution of failures, smartpoints do note make any assumptions about failure rates. Rather, they adjust to the actual failure rate at runtime by selecting the checkpoint policy optimal for the rate.

5. RESEARCH PLAN

We intend to implement and evaluate qpoints and smartpoints during the years 2016-2017. With qpoints we have to address three key issues. First, current $Qn.m$ encoding schemes rely either on custom hardware [10, 16] or on algorithms hand-crafted to represent parameters with $Qn.m$ values [13]. To keep our approach general, we cannot assume such support and have to come up with a software encoder.

Second, to make qpoints handle failures with a prior warning, we have to meet hard real-time requirements of the warning. Given the limited number of IO operations per second, considerable memory footprint and the software $Qn.m$ encoder, qpointing in a timely manner requires finding a tradeoff between parameter memory usage and accuracy of the final model.

Third, we have to avoid numerical issues. The $Qn.m$ parameter representation needs (a) to have enough accuracy to represent small parameter values or updates commonly encountered in real world machine learning deployments and (b) to avoid introducing bias into parameter values.

With smartpoints, we need to solve two challenges, namely (a) we have to adjust Randomized Weighted Majority to handle the specific case of roll-back recovery and (b) we have to preserve RWM guarantees during this adjustment. With respect to modifications, three are absolutely necessary: designing the expert pool, adjusting the penalty rate β, and handling cases where persisting the data is compulsory, such as when a system cannot hold data in memory due to memory pressure.

Our baseline for both qpoints and smartpoints will be fault tolerance policies commonly hard-coded into modern dataflow systems such as "checkpoint everything" in Apache Hadoop. We will fix the placement and frequency of checkpoints, e.g., "checkpoint the last operator of each third iteration", and run jobs with such setting to get the baseline median (out of five identical runs) wall-clock job execution time. We will then strive to decrease this time with qpoints and smartpoints. With qpoints, the time should reduce because jobs will spend less time persisting qpoints than checkpoints due to the smaller qpoint size. With smartpoints, we expect reduction in the execution time because the majority of jobs will checkpoint optimally for the cluster they run in, while a fixed checkpoint policy is likely to mismatch certain environments. For instance, jobs under low preemption rates should on average complete faster because smartpoints will automatically select the "checkpoint nothing" policy and remove entire checkpoint overhead.

Acknowledgements. This work has been supported through the grant by the German Ministry for Education and Research as Berlin Big Data Center BBDC (ref. 01IS14013A).

6. REFERENCES

[1] RDD Persistence. `spark.apache.org/docs/latest/programming-guide.html#rdd-persistence`, 2015.

[2] Amazon Web Services. EC2 Spot Instance Termination Notices. `aws.amazon.com/blogs/aws/new-ec2-spot-instance-termination-notices/`, 2015.

[3] Amazon Web Services. Managing Interruption. `aws.amazon.com/ec2/spot/spot-tutorials/`, 2015.

[4] S. Babu. Towards Automatic Optimization of MapReduce Programs. SoCC '10, pages 137–142.

[5] M. Bilenko and M. Richardson. Predictive Client-side Profiles for Personalized Advertising. KDD '11, pages 413–421.

[6] A. Blum. On-line Algorithms in Machine Learning. In *Developments from a June 1996 Seminar on Online Algorithms: The State of the Art*, pages 306–325, 1998.

[7] O. Bousquet et al. The Tradeoffs of Large Scale Learning. In *Advances in Neural Information Processing Systems 20*, pages 161–168. 2008.

[8] N. Cesa-Bianchi et al. How to Use Expert Advice. STOC '93, pages 382–391, 1993.

[9] T. Chilimbi et al. Project Adam: Building an Efficient and Scalable Deep Learning Training System. OSDI '14, pages 571–582.

[10] M. Courbariaux et al. Low precision arithmetic for deep learning. *CoRR*, abs/1412.7024, 2014.

[11] J. Dean. Lessons from Building Large Distributed Systems. `www.cs.cornell.edu/projects/ladis2009/talks/dean-keynote-ladis2009.pdf`, 2009.

[12] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113, 2008.

[13] D. Golovin et al. Large-Scale Learning with Less RAM via Randomization. ICML 2013, page 10.

[14] J. Goodman et al. Spam and the Ongoing Battle for the Inbox. *Commun. ACM*, 50(2):24–33, Feb. 2007.

[15] Google Cloud Platform. Creating a Preemtible VM Instance. `https://cloud.google.com/compute/docs/instances/preemptible`, 2015.

[16] S. Gupta et al. Deep Learning with Limited Numerical Precision. *CoRR*, abs/1502.02551, 2015.

[17] N. Littlestone et al. The weighted majority algorithm. *Information and computation*, 108(2):212–261, 1994.

[18] Y. Low et al. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *Proc. VLDB Endow.*, 5(8):716–727, 2012.

[19] P. Moritz et al. SparkNet: Training Deep Networks in Spark. *arXiv:1511.06051*, 2016.

[20] M. Pundir et al. Zorro: Zero-Cost Reactive Failure Recovery in Distributed Graph Processing. SoCC '15, pages 195–208.

[21] A. Salama et al. Cost-based Fault-tolerance for Parallel Data Processing. SIGMOD '15, pages 285–297.

[22] S. Schelter et al. All Roads Lead to Rome: Optimistic Recovery for Distributed Iterative Data Processing. CIKM '13, pages 1919–1928.

[23] P. Upadhyaya et al. A Latency and Fault-Tolerance Optimizer for Online Parallel Query Plans. SIGMOD '11, pages 241–252.

[24] C. Xu et al. Efficient Fault-Tolerance for Iterative Graph Processing on Distributed Dataflow Systems. ICDE '16.

[25] S. Yi et al. Monetary Cost-Aware Checkpointing and Migration on Amazon Cloud Spot Instances. *IEEE Transactions on Services Computing*, 5(4):512–524, 2012.

[26] M. Zaharia et al. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. NSDI'12, pages 2–2.