CheckHint: User-assisted Static Checkpointing for ML

Cameron Wong, Shirley Zhang

ABSTRACT

We present CheckHint, a proof-of-concept lightweight Python library that (1) automatically determines optimal checkpoint placement from *checkpoint hints* provided by the programmer and (2) transforms an annotated function to an optimally fault-tolerant one by inserting code to save and restore program state at the determined locations.

1 Introduction

Modern machine learning jobs require processing unfathomable amounts of data, with training times taking on the order of days to weeks. Over such a long period, interruptions are almost inevitable – from hardware failure, network faults, user error, and so on. To avoid needing to restart days-long training jobs from scratch in such cases, most machine learning frameworks support *checkpointing*, in which the current training state is saved to persistent storage at specific intervals, allowing the job to be resumed from that point if an interruption is encountered.

The act of balancing checkpointing stalls with expected recovery time is quite nuanced, and is most typically done in an ad-hoc, bespoke manner. While there is a wide body of work in improving this state of affairs, very little of it has made its way to modern machine learning frameworks.

We throw our hat into the ring with a system of *checkpointing annotations*. Programmers can demarcate potential recovery points with a lightweight code annotation, which are then selectively inserted to the running program according to some configurable parameters. We intend to focus primarily on static analyses, rather than just-in-time approaches more reliant on run-time profiling.

1.1 Related work

The increasing size and popularity of machine learning models underscore the need for better fault tolerance protocols and automatic checkpointing. In response, there has been a surge of recent research focusing on checkpointing for distributed systems and large model training. Some works focus on developing checkpointing systems for specific applications. For example, Eisenman et. al developed Check-N-Run [2] , a checkpointing system specifically for training deep learning recommendation models that takes advantage of differential checkpointing and quantization techniques. Check-N-Run was implemented at Facebook and reduced both the necessary write bandwidth and capacity on real-world instances. In contrast, PhoeniQ focuses on fault tolerance for analytical parallel database systems by taking continuous checkpoints of every operator pipeline, which allows the database to recover after partial failures [1]. Earlier work by Salama et. al. also proposes a fault-tolerance strategy for mid-query failures in parallel data engines, with the particular aim of addressing heterogeneity [8]. For example, their work aims to handle both mixed workloads which have both short running, interactive queries as well as long running batch queries. In contrast, we develop a more general model for checkpointing that is not tied to any specific application.

One particular area of interest for researchers has been checkpointing strategies for deep neural network training. Checkfreq presents an automatic, fine-grained checkpointing framework that computes the frequency of checkpoints at an iteration level and determines whether to store each checkpoint on the CPU or GPU [6]. Furthermore, Checkfreq attempts to parallelize checkpointing with computation in order to reduce overhead by using two-step checkpointing. We take inspiration from Checkfreq's model by also asking the user for a permissible checkpointing overhead, but differ from Checkfreq as we consider adding checkpoints whenever the developer indicates a checkpointing annotation, which may be at a finer granularity than the iteration level. DeepFreeze also focuses on asynchronous checkpointing by using a multi-level approach that involves both lightweight and heavy persistence strategies in order to distribute the checkpointing load over multiple processes [7]. Swift considers the same setting as Checkfreq but takes a different approach by attempting to resolve the inconsistencies of the model state instead of saving a copy of the model state in memory [12]. While Checkfreq and Swift aim to recover from one failure at a time, Oobleck is instead designed to be able to tolerate up to f simultaneous failures by creating at least $f+1$ equivalent pipeline replicas for large DNN models [3]. Gemini identifies the bandwidth costs of using remote storage as a problem, and addresses this by instead checkpointing to CPU memory of the host machines [11]. This requires optimizing both the availability of checkpoints in CPU memory and balancing checkpointing traffic with train-

ing traffic.

Several other works take other approaches to fault tolerance that are not based on checkpointing. Wang et. al. propose lineage stash, a decentralized causal logging technique with less overhead during recovery, as an alternative to checkpointing [10]. In lineage stash, the most recent lineage is forwarded along with the task so that a worker rerunning a failed task has full information about which upstream tasks need to be recomputed. Hoplite advocates for a redesign of the communication layer when a task-based distributed framework is used [14]. In particular, Hoplite uses asynchronous fine-grained pipelining in its data transfer schedule, and dynamically adapts the schedule when a failure is detected.

Finally, there are a few works that tackle the notion of automatic checkpointing at compile time. Phoebe, a learning-based checkpoint optimizes developed at Microsoft, takes in constraints and an objective function at compile-time and outputs an optimal set of checkpoints [13]. The main technique used in Phoebe is determining the optimal cut in the execution graph using integer programming. Kraft et. al present a programming model, data-parallel actors, specifically for distributed query serving systems where the query serving system is viewed as a collection of stateful actors [4]. More broadly, Tardieu et. al. tackle fault tolerance in cloud computing by proposing a fault-tolerant programming model based on retry orchestration and tail calls [9]. For interactive applications, Mogk et. al. present new high-level programming abstractions based on the functional reactive programming model [5]. Our work builds on the idea of checkpointing at compile time, but we present a new model which depends on checkpointing annotations.

2 Model

Our model interpolates between the models in Checkfreq [6] and Phoebe [13]. Checkfreq focuses solely on DNN training, and its main algorithm determines how frequent checkpoints should be saved at an iteration level. We take inspiration from Checkfreq's notion of a 'permissible checkpointing overhead', although we ask for a slightly different user input than Checkfreq does. Our model will be more complicated than Checkfreq's in that we will consider saving checkpoints at a task-level granularity and in more general applications than DNNs. On the other hand, Phoebe is very general and is meant for determining when checkpoints should be saved in task graphs which may have complicated dependencies. To do this, Phoebe presents both a computationally expensive ILP algorithm and a heuristic algorithm which includes brute-force search. We will not consider task graphs with dependencies, which allows us to consider simpler algorithms not based on linear programming. Like Phoebe, we assume that there exist accurate predictions for inputs such as the runtime of each task, based on historical data.

While our model is general enough to be used outside of the context of machine learning, it was designed with machine learning procedures in mind. To that end, we will assume that a program is composed of discrete tasks (which we expect will be repeated) where we have accurate predictions for the runtime of each task. This may not be true for general programs, but is a reasonable model for machine learning pipelines where a large amount of data needs to be repeatedly processed in the same way. For example, in DNN training, we could think of each iteration as a task or each of [data augmentation, forward pass, backward pass, weight update] as a task [6]. As another example, in Monte Carlo Tree Search, we might think of the tasks as [selection, expansion, simulation, backpropagation]. It may even make sense to consider multiple regimes for the same type of task, as runtimes and complexities for a task may vary drastically over the course over the algorithm. In Monte Carlo Tree Search, for instance, we expect non-trivial simulation runtime at the beginning of training, but not near the end. Therefore, we could think of the tasks as [selection, expansion, sim-beginning, sim-mid, sim-end, backpropagation], where each of the simulation tasks has a different runtime. Because machine learning algorithms often follow similar patterns, a user may already have a good idea of when it makes sense to checkpoint. In this project, we will assume that the user has reasonable intuition for when to checkpoint and save the entire state of the program whenever checkpointing occurs. We will also assume a uniform failure rate (i.e. failure causes are external to the program) and that storage is not a concern. We acknowledge that each of these assumptions is significant and it would be ideal if such assumptions could be relaxed in future work, but feel that our model and implementation is interesting nonetheless.

In our model, we will assume that there are $x_1, ..., x_k$ separate training tasks that can be repeated and combined in any order. For example, if there were three tasks x_1, x_2, x_3, a valid task order could be $[x_1, x_3, x_1, x_3, x_2, x_1, x_2]$. Each task x has an associated integral time to completion $t(x)$. We are given the task order and task lengths as inputs to our program. Furthermore, we take as input *checkpointing hints* by the developer, which are only allowed to be inserted as the end of a task. For example, the previous task order could be augmented by checkpointing hints h as follows: $[x_1, x_3, h, x_1, x_3, h, x_2, h, x_1, x_2]$. Note that we could create a new type of task $x_{k+1} = [x_1, x_3]$ which has completion time $t(k+1) = t(x_1) + t(x_3)$. We can preprocess every sequence of adjacent tasks that do not have a checkpointing hint within them in this way so that every two tasks are separated by a checkpointing hint (Fig. 1). Therefore, we will assume WLOG from now on that the input task order has a checkpointing hint after every task, and that the length of the task order is n. It will be convenient to refer to a task by its index in the task order, and we will refer to the jth task in the task order by y_j and its time to completion as t_j.

We assume that checkpointing has a fixed time cost t_c and that we are given a permissible checkpointing overhead $o \in [0, 1]$, where o represents how much *extra* time we are

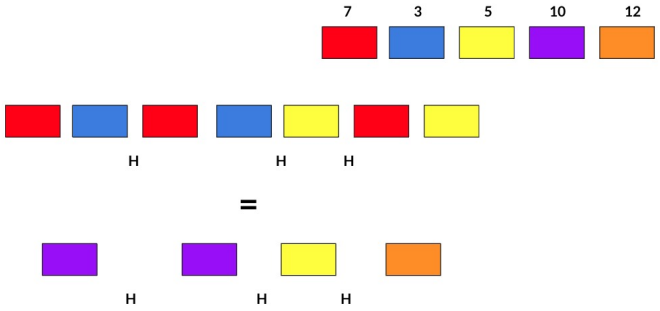

Figure 1: An example of preprocessing to ensure that every two tasks are separated by a checkpointing hint.

allowed to devote to checkpointing (as a percentage of training time). We can then calculate the maximum number of checkpoints that we can insert as:

$$\ell = \left\lceil \frac{o \cdot \sum_{j=1}^{n} t_j}{t_c} \right\rceil$$

We will assume that the failure rate is uniform over time. Specifically, we will assume that at each unit of time, the probability of failure is p. Our goal is then to minimize the total expected recovery time given that we can convert only ℓ of the checkpointing hints to real checkpoints. At each checkpoint, we save the entire state of the program. We call the set of checkpoints that achieves this an *optimal checkpointing schedule*.

We observe that adding ℓ checkpoints will naturally divide the tasks into $m = \ell + 1$ blocks. Each block ends at a checkpoint, except for the last block, which ends at the end of the program. Therefore, an equivalent framing of the problem is to find the optimal placement of block ends, where the last block end must always be placed at the end of the program. In order to find the optimal checkpointing schedule, our algorithm will use a dynamic programming approach which optimizes over placement of the block ends. Note that although the placement of the last block end is fixed, its existence is important in our algorithm as we still need to take into account the expected recovery time of the final block.

2.1 Recovery time per block

We define the total recovery time of a program as the total time the program spends repeating actions that it had already completed due to a failure. In other words, the total recovery time of a program is the total time to completion of the program (with failures) subtracted by the total completion time of a program assuming there were no failures. Note that minimizing the total expected recovery time is the same as minimizing the total expected completion time, as the total completion time of a program assuming no failures is a constant.

We start by computing the expected total recovery time for a block b which consists of the tasks with indices from j' to j, inclusive. The total length of the block is then $t(b) = \sum_{i=j'}^{j} t_i$. We will denote the probability of failure in block b as $p(b)$, and we observe that $p(b) = 1 - (1-p)^{t(b)}$. We also observe that as the failure rate is uniform, the expected recovery time in b after one failure is exactly $\frac{1}{2}t(b)$. We can therefore calculate the expected total recovery time for block b to be:

$$\begin{aligned}
&E[\text{total recovery time in } b] = \\
&\quad = \sum_{k=1}^{\infty} E[\text{total recovery time in } b \mid k \text{ failures}] \cdot \Pr[k \text{ failures}] \\
&\quad = \sum_{k=1}^{\infty} k \cdot E[\text{recovery time in } b \text{ after one failure}] \cdot \Pr[k \text{ failures}] \\
&\quad = \frac{1}{2}t(b) \cdot \sum_{k=1}^{\infty} k \cdot \Pr[k \text{ failures}] \\
&\quad = \frac{1}{2}t(b) \cdot \frac{p(b)}{1-p(b)}
\end{aligned}$$

The first equality is by the law of total expectation. The second equality is because the recovery time in b after each failure is is independent of how many failures have already occurred. The third equality is because the expected recovery time in b after one failure has no dependence on k. The fourth equality is by the expectation of a geometric random variable. We end up with a closed-form cost function that we can minimize over in the recurrence.

2.2 Recurrence

Let $\mathcal{S}$ be the set of all possible checkpointing schedules. In the dynamic programming recurrence, β indexes over block ends and j indexes over tasks. $OPT(\beta, j)$ then represents the optimal way to place β block ends among the the first j tasks, where 'optimal' means 'minimum expected recovery time'. We let $\text{cost}(j', j)$ be the expected recovery time for block b if block b contains all tasks between j' and j, inclusive, and observe that we derived a closed-form expression for the cost in the previous section. Recall that the last block end must be placed directly after the jth task. Therefore, when there is only one block end and j tasks, our placement of the block end is forced and we have

$$OPT(1, j) = \text{cost}(1, j)$$

We initialize our dynamic program with $OPT(1, j)$ for all $j \leq n$. We then build off of the initialization using the following dynamic programming recurrence:

$$OPT(\beta, j) = \min_{j' \in [\beta-1, j-1]} (OPT(\beta - 1, j') + \text{cost}(j' + 1, j))$$

Intuitively, the recurrence places the last block end after the jth task (as is required) and then iterates over all possible positions where the second-to-last block end could have been placed. For each such position, the recurrence computes the total cost as the cost of the best sub-schedule added to the cost of the final block. The recurrence saves the minimum

```
import checkhint
from checkhint import _checkpoint

@checkhint.schedule_checkpoints(
    allowance=0.1,
    # ...
    )
def train(cost_fn, gradient_fn):
  # ...
  for epoch in range(0, 5):
      _iterations: 5
      for batch in range(0, 50):
          _iterations: 50
          params = o.fmin_cg(
            cost_fn,
            params,
            fprime=gradient_fn)
          _checkpoint()
      _checkpoint()
```

Figure 2: An example training loop, with checkpointing hints inserted

total cost over all positions as $OPT(\beta, j)$. We can save the optimal schedule for each β, j as well using additional space.

Recall that n is the length of the task list and m is the number of available block ends. Using the recurrence, we can compute $OPT(m, n)$ and determine its associated schedule S. We then return the first $m - 1$ block ends as our optimal checkpointing schedule.

3 Design and Implementation

CheckHint is implemented as a code transformer acting on Python source code, making use of Python's built-in `ast` module. We proceed as follows:

- Split code into a task tree mirroring that of the input syntax tree.
- Unroll loops and join if-branches to create the flat task list expected by the model.
- Compute the optimal checkpointing schedule.
- Perform liveness analysis to determine the set of variables that must be written to persistent storage
- Re-roll loops and insert filesystem code to rebuild an equivalent Python program with checkpoints inserted.

3.1 Lightweight Syntax Extension

Potential checkpoint locations are marked by calling the special function `_checkpoint` with no arguments. Additionally, each loop includes a magic variable `_iterations` that is "type" annotated with a constant upper bound on the number of iterations. This (ab)use of Python's type annotation syntax (which allows types to include arbitrary Python expressions) allows us to mark loop bounds with no runtime overhead whatsoever.

```
@dataclass
class Basic(Task):
    body: list[ast.stmt]

@dataclass
class While(Task):
    test: ast.expr
    body: list[Task]
    num_iterations: int
```

Figure 3: Representing tasks as meta-syntax (snippet)

If the function `train` is simply called as-is, both `_checkpoint` and the variable annotation are a no-ops. However, calling the method `checkhint.expand` on the annotated `train` will engage the rest of the analysis pipeline and eventually return as a string a functionally-equivalent Python program with inferred checkpoints. This string can then be run with Python's builtin `exec` to register the new function immediately, or saved to a file to be run separately.

3.2 Meta-representation

CheckHint's analyses are entirely syntax-directed. Tasks are first constructed by grouping statements around calls to `_checkpoint`, then laid out in a structure resembling that of the original code with extra annotations. Fig. 3 shows the most basic Task (a body of statements with no checkpoints) and a `while` loop, split into sub-tasks and annotated with the number of iterations.

Note that `Basic` tasks do not necessarily straight-line code, only code that cannot possibly cross a checkpointing hint. This means that *all* loops must be annotated with an iteration bound to compute task execution times. Loop structures are only split into subtrees such as `While` if their bodies contain checkpoints.

3.3 Task reconstruction

Because this representation closely mimics that of the input code, reconstructing the structure of a given task is simple – for each task structure, compose its constituent `ast` nodes with the equivalent super-node, possibly recursively.

For *resumable* fault-tolerance, however, more work is needed to allow the application to cleanly resume from the start of a task. CheckHint lifts all tasks (including sub-tasks) to top-level functions and replaces the relevant sections with calls to those functions accordingly.

3.4 Loop re-rolling

Because loops are unrolled before being passed to OPT, the resulting checkpoint schedule may not cleanly separate top-level task boundaries. Instead, CheckHint tracks the provenance of each code block and attempts to re-group equiva-

```
def _train_inter1(params, rows, cols):
  # ...
  for epoch in range(0, 50):
    # inserted checkpoint
    _save_checkpoint(
        params=params,
        epoch=epoch,
        _task=2)
    params = _train_inter2(params)
  return params

def _train_inter2(params):
  for batch in range(0, 5):
    # ...
  return params
```

Figure 4: Lifting task bodies to top-level functions

lent block structures with a common parent. For example, if a `while` loop with sub-tasks a and b expands to tasks $a_1, b_1, a_2, b_2, ...$, CheckHint will know that each a_n and b_n correspond to a and b from the loop. If there is a checkpoint inserted between a_n and b_n for all n, we can simply rebuild the while loop with body $a; \texttt{\_save\_checkpoint}; b$.

4 Evaluation

4.1 Quantitative

4.1.1 Experimental setup

All experiments were run on the FAS research cluster. We repeat each experiment five times, dropping the page cache and removing all intermediate artifacts between runs. All experiments are run single-threaded.

We measured performance on a straightforward DNN training on the stock MNIST dataset with checkpoint hints placed at the end of each iteration and each epoch. We recorded "happy-path" performance testing only; we were unable to complete performance testing when restoring from backup due to lack of time.

4.1.2 Accuracy implications

Unlike traditional model checkpointing schemes, we choose to reload the entire relevant program state from checkpoint rather than only specific model parameters. In principle, this avoids additional hazards of resetting random state (although our implementation does not correctly accomplish this), and suggests that recovery time should be proportional only to the time delta between backups plus IO and state restoration overhead.

4.1.3 Results and analysis

DNN training against the MNIST data completed in approximately 2 hours with no difference distinguishable from noise with checkpoint tolerance set to 5%, 10% or 15%. This is most likely due to faulty benchmarking; our heuristic vastly overestimates checkpoint stalls relative to individual numpy operations, and our analysis of numpy operations is also very inaccurate (we discuss this further in Sec. 5.1.2).

We expected computing the optimal checkpointing schedule to also have meaningful benchmarks, but all permutations of hints inserted to real training pipelines completed instantly.

4.2 Qualitative

4.2.1 Soundness

An initial test revealed several soundness bugs with our liveness analyzer. The rules surrounding liveness of Python variables is far subtler than we had predicted even when limited to top-level names, and we were forced to manually reject several constructs such as the use of global variables, early returns and exception handling in the interest of implementation time. However, all "standard" control flow functions necessary for training (particularly notably loops and conditionals) had their live-in set correctly inferred.

Experimentally, we tested soundness by randomly interrupting and resuming training jobs and ensuring that the trained model weights remained the same. We further found that, to ensure full correctness, we would need to save the state of any randomness used during training, which we had neglected to account for.

4.2.2 Impact on programming style

When converting a stock MNIST training model to use our framework, we observed a crucial flaw in our core methodology, which is that we had assumed that loops in the training data would be written using Python's literal `for` or `while` constructs. In practice, however, real models make heavy use of list comprehensions and the higher order `map` function. This means that our benchmarks were unable to make use of data parallelism, as we had to hand-rewrite many such constructs to use manual loops. Even then, we found even hand-unrolling loops iterating over matrix-level operations to be impossible.

5 Discussion

5.1 Methodological problems

We believe that our results represent neither a concrete improvement to the state of the art nor a refutation of our core ideas. Many of the big-picture problems can be traced back to the authors' overall inexperience with machine learning (in fact, the process of assembling a benchmarking pipeline for this project was one author's first experience training a model at all) and lack of time.

5.1.1 Checkpoint cost

In practice, the assumption that all checkpoints are equally expensive is not a realistic one. Even setting aside systemic hazards such as storage contention, different tasks will naturally require different amounts of data to be saved. A fully-

realized system could perform further analysis on the types and number of live-in variables to each task to determine how expensive checkpoints are relative to each other. This could even be combined with just-in-time profiling in which hints refer to points at which system characteristics may be sampled before checkpointing.

5.1.2 Determination of task time

In a similar vein, the assumption that task times could be estimated statically from examination of the high-level Python code was very faulty. Seemingly straight-line code with no loops or branches could hide arbitrary control flow behind library functions, the source for which may-or-may not be available. Even innocuous-looking expressions like `x *= 1-x` may actually be expensive matrix operations. In a fully-statically typed language where the type of the variable `x` is known but any remotely realistic machine learning workload is likely far too complex for this to be feasible.

5.2 Over-generality

CheckHint performs relatively simple static analyses over the surface-level Python program. Because Python is such a complex language, this loses us significant amounts of operational precision and only allows coarse-grained results. While it is in principle possible to develop a more precise analysis at this level of granularity with more time and engineering effort, most practical machine learning pipelines use domain-specific frameworks that expose a deeply-embedded domain-specific language which serves as a far richer target. Performing liveness and cost-bound analysis on, for example, a Tensorflow computation graph would give far more accurate estimates, as well as allowing checkpoint hints to be inserted into much more idiomatic training code.

5.3 Future Work

5.3.1 Hint elision

In theory, there is no reason that the programmer should need to place all checkpoint hints manually. Most machine learning workloads have natural checkpointing locations (at the end of every loop, before touching a fallible resource, etc) that could be inferred.

5.3.2 More sophisticated cost analysis

Currently, our analysis relies on unrolling loop bodies and fusing parallel conditional branches to obtain a linear timeline of tasks. This can cause major blowup in the time needed to compute an optimal checkpointing schedule as loop iteration counts increase. Heuristically, we found that, for iteration counts over 10,000, choosing to batch loops by 100 iterations greatly improved allocation time at no observable cost to the final result, but this is very ad-hoc and does not scale further.

Instead, loops should be analyzed as-is, determining the optimal batch size from stall tolerance and the cost of tasks within the loop body. This would require enriching our model to be aware of the loop structure and know how to allocate stall costs to the loop body.

5.3.3 Distributed infrastructure

One advantage of snapshotting the entire local state (rather than only the model training weights) is that CheckHint scales for free to distributed settings in which individual nodes going offline is not necessarily fatal to the overall process – simply reload the machine state from the checkpoint, reconnect to the system and apply any updates. It would only be a matter of engineering effort to apply CheckHint to the code running on a node to allow mostly-seamless restoration of single nodes.

6 References

[1] Y. Bessho, Y. Hayamizu, K. Goda, and M. Kitsuregawa. Dynamic fault tolerance for multi-node query processing. *IEICE TRANSACTIONS on Information and Systems*, 105(5):909–919, 2022.

[2] A. Eisenman, K. K. Matam, S. Ingram, D. Mudigere, R. Krishnamoorthi, K. Nair, M. Smelyanskiy, and M. Annavaram. {Check-N-Run}: A checkpointing system for training deep learning recommendation models. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 929–943, 2022.

[3] I. Jang, Z. Yang, Z. Zhang, X. Jin, and M. Chowdhury. Oobleck: Resilient distributed training of large models using pipeline templates. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 382–395, 2023.

[4] P. Kraft, F. Kazhamiaka, P. Bailis, and M. Zaharia. {Data-Parallel} actors: A programming model for scalable query serving systems. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 1059–1074, 2022.

[5] R. Mogk, J. Drechsler, G. Salvaneschi, and M. Mezini. A fault-tolerant programming model for distributed interactive applications. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–29, 2019.

[6] J. Mohan, A. Phanishayee, and V. Chidambaram. {CheckFreq}: Frequent,{Fine-Grained}{DNN} checkpointing. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 203–216, 2021.

[7] B. Nicolae, J. Li, J. M. Wozniak, G. Bosilca, M. Dorier, and F. Cappello. Deepfreeze: Towards scalable asynchronous checkpointing of deep learning models. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pages 172–181. IEEE, 2020.

[8] A. Salama, C. Binnig, T. Kraska, and E. Zamanian. Cost-based fault-tolerance for parallel data processing. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 285–297, 2015.

[9] O. Tardieu, D. Grove, G.-T. Bercea, P. Castro, J. Cwiklik, and E. Epstein. Reliable actors with retry orchestration. *Proceedings of the ACM on Programming Languages*, 7(PLDI):1293–1316, 2023.

[10] S. Wang, J. Liagouris, R. Nishihara, P. Moritz, U. Misra, A. Tumanov, and I. Stoica. Lineage stash: fault tolerance off the critical path. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 338–352, 2019.

[11] Z. Wang, Z. Jia, S. Zheng, Z. Zhang, X. Fu, T. E. Ng, and Y. Wang. Gemini: Fast failure recovery in distributed training with in-memory checkpoints. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 364–381, 2023.

[12] Y. Zhong, G. Sheng, J. Liu, J. Yuan, and C. Wu. Swift: Expedited failure recovery for large-scale dnn training. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, pages 447–449, 2023.

[13] Y. Zhu, M. Interlandi, A. Roy, K. Das, H. Patel, M. Bag, H. Sharma, and A. Jindal. Phoebe: a learning-based checkpoint optimizer. *arXiv preprint arXiv:2110.02313*, 2021.

[14] S. Zhuang, Z. Li, D. Zhuo, S. Wang, E. Liang, R. Nishihara, P. Moritz, and I. Stoica. Hoplite: efficient and fault-tolerant collective communication for task-based distributed systems. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, pages 641–656, 2021.