Fast Low-Overhead Logging Extending Time

Anusha Dandamudi

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2021-117
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2021/EECS-2021-117.html

May 14, 2021

Copyright © 2021, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Fast Low Overhead Recovery Extending Time

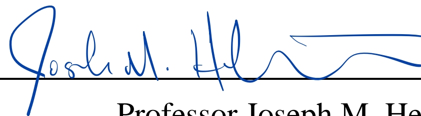
by Anusha Dandamudi

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

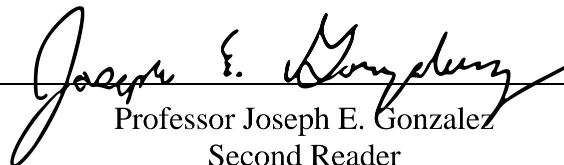


Professor Joseph M. Hellerstein
Research Advisor

MAY 9, 2021

(Date)

* * * * * * *



Professor Joseph E. Gonzalez
Second Reader

5/12/2021

(Date)

Fast Low-Overhead Logging Extending Time

Copyright 2021
by
Anusha Dandamudi

Acknowledgments

To my advisor, Prof. Joseph Hellerstein, thank you for providing me with the opportunity to participate in exciting research. I truly appreciate your invaluable feedback and the time you took each week to discuss my progress.

To my graduate mentor, Rolando Garcia, thank you for your continuous guidance, support, and feedback. I am very grateful for the time you took to assist me throughout my research journey.

Abstract

Fast Low-Overhead Logging Extending Time

by

Anusha Dandamudi

Master of Science in Electrical Engineering and Computer Science

University of California, Berkeley

Professor Joseph M. Hellerstein

ML model development has become increasingly challenging due to the rapid increase in the scale and complexity of models and model workloads, making standard logging and debugging practices insufficient. Due to the impracticality of exhaustive logging during model training as well as the infeasibility of a complete re-execution of model training for debugging purposes, we turn to hindsight logging. Hindsight logging is an efficient post-hoc logging method that periodically collects checkpoints during model training and performs parallelized checkpoint resume for context recovery during debugging. With hindsight logging, model developers can defer logging overhead and postpone considering which execution data they may need for analysis until model training completes.

However, model developers and machine learning researchers and practitioners are faced with a continually evolving landscape of datasets, processing pipelines, analysis frameworks, and models to explore and investigate. In this paper, we introduce FlorET (Fast Low-Overhead Recovery Extending Time), a system for empowering model developers to perform hands-free analysis and debugging over multiple model training sessions. The key features that facilitate this advancement consist of (i) automatic version control, (ii) robust log statement propagation, and (iii) parallelized inter-version replay. We evaluate the efficacy of log statement propagation and code alignment techniques for temporal hindsight logging and then conduct system overhead analysis.

Contents

1 Introduction **1**

2 Background **4**
2.1 Motivating Scenario 4
2.2 Related Work 5

3 System Overview **7**

4 System Architecture **10**
4.1 Version Control Engine 10
4.2 Log Statement Propagation Engine 11
4.3 Parallelized Replay 13

5 Results **14**
5.1 Synthetic and Ecological Benchmarks 14
5.2 Log Statement Propagation Accuracy 15
5.3 System Overhead 16
5.3.1 Versioning Overhead 16
5.3.2 Log Statement Propagation Overhead 18
5.3.3 Parallel Replay Overhead 19

6 Future Work **20**
6.1 Optimizations and Improvements 20
6.2 Usability-Enabling Features 21
6.2.1 IDE Integration for Version Navigation 21
6.3 Efficiency-Enabling Features 21
6.3.1 Query-Cost Highlighting 21
6.3.2 Search and Sampling 22
6.4 Delta Pruning 23

7 Conclusion **25**

Bibliography **26**

Chapter 1

Introduction

ML model development has become increasingly challenging due to the rapid increase in the scale and complexity of models and model workloads [3]. Model developers track and visualize time series data, often with the help of tools like WandB [14] and Tensorboard [9], in order to identify core ML issues evident during model training. However, standard logging and debugging practices are not sufficient for model development. The challenges that accompany model development and debugging often exceed those of regular software development in both complexity and obscurity, especially because of the incapability of modularization in deep learning and the increased dependence of model training on data. As such, the "possible error surface," specifically when working with deep models, is substantially larger [10]. Hence, model developers, when encountered with bugs, turn to analyzing data such as training metrics and gradient magnitudes for debugging their model pipelines.

Recovering execution data of model training is an indispensable part of model analysis and debugging. Model developers often forgo heavy logging during training since moving execution data from GPU to DRAM results in significant overhead. Instead, they attempt to hand-select which execution data to log, but there are many situations where they neglect a key piece of information necessary for pinpointing an issue that could arise in the future. Due to this potential for high miss rates, model developers are usually forced to rerun model training from scratch to obtain the critical missing information.

Such cyclic debugging leads to prohibitive latency as model training runtimes can be lengthy. Also, the cost of re-running model training in its entirety for solely debugging purposes is often problematic in terms of compute. Hence, a purely logical approach, where the required execution data is recomputed, is impractical. Model developers must therefore turn to tracking key time series data such as training metrics, tensor histograms, and images/overlays [3]. However, a purely physical approach, where all execution data is stored and then recovered from disk, is also infeasible because such a logging mechanism would be too prohibitive in terms of latency and storage. Additionally, since model developers cannot be expected to predict in advance which execution data they will need to debug a future problem, we need a

mechanism that allows querying any execution data. A hybrid between a physical approach, which takes partial checkpoints during training with minimal logging overhead, and a logical approach, which uses these checkpoints to considerably speed up re-execution of training, is ideal. This approach to execution data recovery is called hindsight logging [3].

Hindsight logging is a method that enables model developers to defer logging overhead and postpone considering which execution data they may need for analysis until they begin debugging. By periodically saving checkpoints at sparse intervals during training, we can keep logging overhead to a minimum, while also enabling efficient future context recovery via checkpoint resume. This way, selective logging can be performed post-hoc and model developers can query any execution data after model training completes without performing exhaustive logging during training. Queries can be answered by extracting relevant data directly from checkpoints if available or resuming execution from a checkpoint until all relevant data becomes available.

Nonetheless, model developers and machine learning researchers and practitioners are faced with a continually evolving landscape of datasets, processing pipelines, analysis frameworks, and models to explore and investigate [12]. As a result, they regularly update, expand, and evolve their ML codebases throughout the lifecycle of their analyses. Rather than strictly following a predefined experimental procedure, the speed of development in machine learning necessitates a dynamic, investigative, and evolving approach to research and development. A low-effort, high-reward mechanism for answering queries through comparison and/or visualization of the effects of code changes through the project version tree would provide the insights necessary to correct and improve code, producing high-accuracy, generalizable models more efficiently.

FlorET (Fast Low-Overhead Recovery Extending Time) is a system that allows machine learning developers to query any state post-hoc across multiple versions of code with minimal effort from their side. It enables model developers to query any execution data without a complete program re-execution, with the additional capability of querying over a history of model training runs. If the model developer has some insight into what the problem is and what in the execution data to investigate, then debugging and anomaly detection with fast low-overhead recovery over a single version would suffice. However, if this is not evident to the model developer, then FlorET can guide hindsight logging via cross-comparison between the execution data from the current model training run and historical ones.

From their standpoint, model developers simply run model training with the FlorET harness and insert a log statement in their most recent code version to indicate their query and to recover execution data from a history of code versions. This minimizes technical debt as the model developer can avoid the laborious and more error-prone process of manually inserting the relevant log statement in the sequence of prior versions as well as individually performing checkpoint resume on that sequence of prior versions.

In this paper, we discuss a key scenario that motivated FlorET as well as describe important related work. We then introduce FlorET's architecture, and examine its key components, including the version control engine, the log statement propagation engine, and the inter-version parallel replay. Next, we test the effectiveness of our log statement propagation techniques on both synthetic and ecological benchmarks, evaluating it on its correctness and robustness. Lastly, we assess the overhead accumulated by each of the key components of FlorET.

Chapter 2

Background

Because model development can fail either silently or due to a series of accumulated changes, the model developer may need to perform root-cause analysis over a series of model training runs to identify the training run the bug was initially introduced. We present a motivating scenario where a model developer, finding a decrease in accuracy and normal gradient magnitudes, wishes to investigate the bounding box outputs of their object detection model over the history of code changes.

2.1 Motivating Scenario

Suppose a model developer wishes to build a new computer vision model for the task of object detection. After constructing the model and specifying the optimizer and loss function as well as performing data augmentation, the model developer trains their model. Once training completes, the model developer, equipped with checkpoints of training metrics, recognizes that the model accuracy prematurely plateaued. Suspecting that they set their learning rate too high, they retrain using an exponential learning rate scheduler, but still see no improvement. After a couple more iterations of adjusting the learning rate and other hyperparameters in addition to increasing the width and depth of the model, the model developer sees a drop in accuracy.

They decide to discontinue ad-hoc adjustments and turn to a more methodical approach to perform root-cause analysis. They begin by examining the gradient magnitudes via checkpoint resume on the most recent code version to discern if they have encountered the vanishing gradients problem. Finding the gradient magnitudes to be normal, they decide to inspect the bounding boxes instead. However, model developers typically avoid logging bounding boxes until all other training metrics, tensor histograms, etc. are checked and found to be insufficient in identifying the problem since logging bounding boxes for every image during each epoch is a high-overhead activity. Having neglected logging bounding boxes, the model developer retrieves them from the most recent training run via checkpoint resume. They find

that the bounding box of an object, say a dog, captures only a fraction of it, say its ear. The model developer now asks, "Has this been happening in all my code versions or is this just a consequence of my most recent change?" Specifically, they wish to determine if the most recent change, an accumulation of changes, or both caused the bounding box discrepancy.

To visualize the bounding boxes from the previous model training runs, the model developer can pose their query by simply inserting a post-hoc log statement in only their most recent code version, leaving FlorET to automate this log statement's propagation to previous versions and run checkpoint resume for each. This circumvents the need to manually add log statements to all prior versions as well as perform checkpoint resume separately for each by the model developer. The methodology used to achieve this is described in more detail in Chapter 4.

In addition, FlorET can assist the model developer in searching over historical execution data to determine whether the detected anomaly is present in all `n` code versions, in `k` of the `n` code versions, or just the latest code version. This information is useful when trying to understand the reasons why code changes in the project version tree resulted in an unsuccessful training run. Being able to identify the most recent working version in addition to determining whether the same underlying problem caused subsequent versions to fail is especially valuable during exploratory model development and debugging.

2.2 Related Work

FlorET is an extension of prior work by Garcia et al. called Fast Low-Overhead Recovery (Flor) [3]. Flor enables post-hoc logging of a single version of training code to efficiently recover execution data after-the-fact, using a two-phase mechanism called hindsight logging enabled by the following steps:

1. **Record phase**. The record phase is characterized by the efficient capture and materialization of periodic checkpoints during training. Specifically, the side-effects of each training loop are checkpointed to enable efficient replay. An adaptive periodic checkpointing mechanism is used to control record overhead to remain within user-specified thresholds. Background materialization, in which a separate, short-lived child process handles the serialization of the checkpoint and materialization to disk, enables the parent process to continue with model training and prevents significant latency overhead.

2. **Replay phase**. The replay phase enables efficient context recovery of key times series data such as training metrics, tensor histograms, and images/overlays [3]. The checkpoints captured during the record phase eliminate inter-iteration dependencies making training replay fully parallelizable across epochs. This is because context recovery from any checkpoint to its subsequent one can now be run independently of

the others. Thus, model developers can quickly replay their code from the beginning with no additional code changes, and Flor will intelligently skip recomputation of some loops by loading the relevant checkpoints from disk.

Flor's automatic code instrumentation, necessary for methodical hindsight logging, decreases the technical debt that model developers incur and also reduces the error surface by eliminating developer-induced manual errors.

Chapter 3

System Overview

FlorET extends Flor to a new dimension – time. FlorET facilitates query serving and analytics over the code changes across project versions and provides model developers with the insight necessary to help answer their questions about their model training sessions. After surveying the literature and talking to data scientists and experts in the field, we derive the following requirements for the FlorET system.

- **Automatic version control**. Version control is necessary in order to replay historical versions and support inter-version queries posed by the model developer during debugging. We provide version control of the code and checkpoints collected during the record phase. We rely on Git for code versioning and S3 for storing the versioned checkpoint files. The code version is linked to its corresponding checkpoint files and we describe the mechanism used to link them together in Chapter 4.1. Furthermore, due to the continuously evolving datasets, we need to provide data versioning in addition to code-data coupling. The mechanism used to do this is also described in more detail in Chapter 4.1. Without this advanced version control, replay over the history of code versions would not be possible.

- **Log statement propagation**. To provide increased usability and productivity to model developers performing exploratory model development, we provide a mechanism for them to insert a log statement in a single version of code and backpropagate this log statement to any number of prior versions. Since code can change drastically across multiple versions, backpropagating a log statement becomes particularly challenging. FlorET's insertion of the log statement into past versions requires identifying the blocks in past versions corresponding to the block in the current version the log statement was manually inserted into. This relies on robust code alignment techniques, which we describe in detail in Chapter 4.2. Such an automated log propagation technique notably reduces tedium as well as human propensity for error when compared to its manual counterpart.

- **Record and parallelized inter-version replay**. FlorET uses Flor internally during initial execution of model training to periodically collect and materialize checkpoints. When the model developer queries time series data across many versions, we can parallelize replay of each past version, either by recovering historical data or by augmenting existing log files. Note that there are multiple versions of parallelization at play here: inter-version parallel replay by FlorET, as well as intra-version parallel replay by Flor. Both levels of parallelization significantly speed up the multi-version replay, thus enabling efficient hindsight logging across time. In view of the recent explosion in the popularity of cloud computing [2], accelerated by new cloud programming frameworks [4, 8, 11], parallel processing powered by cloud computing has become increasingly accessible. Hence, our multi-version replay can be executed at increasingly parallel rates offline in the cloud.

We now demonstrate the objective driving FlorET's development and the advantages it provides to Flor. With just Flor, inter-version querying across `n` training sessions requires numerous operations to be performed by the model developer. They have to first checkout each of the `n` versions of code, checkpoints, and data corresponding to each training session. Next, they have to insert `n` log statements, one for each code version, and last, they have to replay model training using Flor `n` times, one for each training session. Thus, a single inter-version query over `n` versions requires `3n` developer-initiated steps. In contrast, FlorET performs the same task but with a single insertion of the log statement and a single call to replay execution, invariant to the number of versions involved. Though the number of operations executed by the system remain the same regardless of whether it be with Flor or FlorET, FlorET ultimately minimizes the number of steps the model developer must perform to accomplish the same task from `3n` to just `2`. It is important to note that this process of historical replay is query driven and does not mandate replay across all prior versions, but solely those relevant to answering queries posed by the model developer. We leave the optimizations to reduce the space of what we replay for future work. Figure 3.1 illustrates FlorET's workflow with a small fragment of code.

With the above fundamental requirements in place, we can support additional features to increase usability, minimize technical debt, and increase system efficiency. Some potential features and optimizations are introduced below, a subset of which are described in more detail in Chapter 6.3.

- **Query-cost highlighting**. We can provide query cost highlighting to steer the model developer to faster queries and better equip them with the knowledge to make alternative, lower-overhead queries when possible.

- **Search and sampling**. We can use a search or sampling method to more quickly and efficiently locate regions in training that the model developer could potentially be interested in. We can also more efficiently detect anomalies in key training execution data by performing search and sampling over their values.

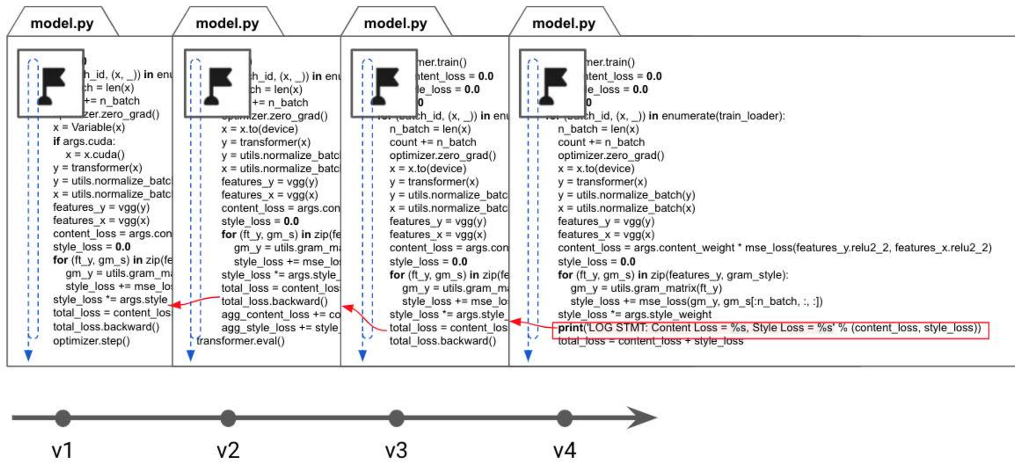


Figure 3.1: Flowchart of FlorET. The red box and arrows demonstrate log statement insertion and propagation, respectively. The flag icon indicates retrieval of code, data, and checkpoints. The dashed blue lines indicate Flor context recovery via replay.

- **Adaptive checkpointing**. We can use adaptive checkpointing to stay within developer-specified resource constraints, while taking checkpoints in a way to enable finer-grained replay and hence increased parallelization. Just as Flor ensures overhead limitations are met for each execution, FlorET can ensure overhead does not exceed constraints for a group of executions. The overhead incurred does not have to be homogeneous across these executions. Instead, aggressive logging in some versions can be offset by minimal logging in others.

- **Multi-stage execution**. We can use multi-stage execution when answering queries i.e. we try to answer queries via static analysis and if unsuccessful, we move to single-epoch replay, a higher cost alternative. But if both approaches are unsuccessful in answering the query, we turn to hindsight logging. Hence, with multi-stage execution, we can further limit the overhead from hindsight logging, with no reduction in applicability and usability.

Chapter 4

System Architecture

4.1 Version Control Engine

The version control engine provides versioning of the code, data, and checkpoints collected during the record phase of model training. Version control is necessary in order to replay historical versions and support inter-version queries posed by the model developer for debugging purposes.

For the model developer to be able to run queries over a history of code versions, we begin by ensuring that their code is versioned. This is sufficient to enable inter-version queries, but we will need additional mechanisms to make the query performance acceptable.

As a start, we need a mechanism to associate each code version with the related Flor checkpoints produced during its model training session. When the model developer queries execution data at a later time, they can be sure that re-execution will be accelerated with the checkpoints produced during the initial execution of the same session. Abiding by the Flor API, the model developer first runs model training with Flor on their current code. As the model trains, periodic checkpoints are saved to its designated location within the Flor hidden directory. Once the model training session completes, FlorET must commit and also link the code with the generated checkpoints. We rely on Git to help with code versioning. Given the precedent that model developers do not like systems committing to their repo and tarnishing their commit history, we create a separate Flor branch and operate on it instead. We commit the code to that branch once model training completes. We then spool the checkpoints saved locally in the Flor hidden directory to S3. We associate the checkpoints with the code that produced them by using the code's Git hash as the name of the parent directory on S3 under which the checkpoint file is stored.

FlorET requires a mechanism for versioning data to guarantee no discrepancy in data exists between execution and re-execution. To this end, FlorET requires that, during execution,

the model developer provide an S3 location of the data they will use for a particular training session. In case the model developer does not have the data stored on S3, they can instead provide the POSIX filepath to the data and let the background spooler, responsible for transferring the checkpoints to S3, also handle moving the data to S3. The data used for training is linked to the code by storing this S3 path to the data in the directory having the code's Git hash as its name. Storing the checkpoints file and the path to the data under this directory ensures that the same checkpoints and data can be automatically retrieved during FlorET's re-execution.

Because S3 buckets are by default mutable, correct re-execution is contingent on the training data on S3 not being altered between execution and re-executions. But if the model developer modifies the training data after execution, FlorET, having saved the hash of the original training data, can detect these changes before attempting re-execution. The model developer is thus not restricted from overwriting their training data, but with the caveat that future re-executions may fail.

4.2 Log Statement Propagation Engine

A key component of the FlorET system is the log statement propagation engine, which allows model developers to query across multiple versions of code in one shot. Given a model and its version history, when a developer inserts a log statement in the latest (or any) version, we aim to insert an equivalent log statement in a subset of the code's prior versions. However, as projects or models evolve, the internal code structure may change too drastically, rendering certain log statements in one version without an equivalent in another. In such cases, a notification to the model developer is issued and the system exits.

We achieve resilient log statement propagation by using a variant of a two-step GumTree diffing algorithm [1] proposed by Falleri et al. for fine-grained source code differencing. The first step, the top-down phase, finds all isomorphic subtrees between two code files, sorts them in decreasing height, and creates mappings, called anchor mappings, between the nodes of the matched subtrees. The second step, the bottom-up phase, creates mappings, called container mappings, between nodes if their descendants have a large number of common anchor mappings. Any unmatched descendants of two nodes that match are linked via recovery mappings. The aggregate of these mappings are used for code alignment.

We then perform a post-processing step to increase resilience to code refactoring and variable renaming by taking advantage of the syntactic structure of the code. We use this matching algorithm to find the block in past versions that most closely resembles the one containing the log statement in the current version. We then adjust the log statement by taking the most common renames over the entire file and applying them to any identifiers present. The GumTree-based propagation algorithm, outlined in Figure 4.1, is resilient to formatting or

```
def propagate(source, node, target):
  # Creation of anchor, container, and
  # recovery mappings between source and
  # target files
  mappings = GumTree(source, target)

  if node in mappings:
    exit('Already in target!')

  block, index = insert_loc(node, mappings)

  new = deepcopy(node)

  # Ensures robustness to renaming of
  # variables in log stmts across versions
  renames = common_renames(mapping)

  block.insert(index, new)
```

```
def insert_loc(node, mappings):
  parent = node.parent
  pos = parent.children.index(node)

  # Retrieving code block and file
  # location for inserting log stmt
  for prev in parent.children[:pos]:
     if prev in mappings:
       # Match found in target file
       ref = mappings[prev]
       block = ref.parent
       index = 1 + block.index(ref)
       return block, index

  if parent in mappings:
      return mappings[parent], 0

  exit('Unable to map context!')
```

Figure 4.1: Pseudocode for the GumTree-based log statement propagation technique.

documentation changes as it operates on the syntax tree. In addition, it is capable of divergence detection i.e. if a block was deleted or merged, it is likely the architecture has changed making propagation invalid, so GumTree is unlikely to find a spurious code block match in such cases.

We investigated a number of other approaches with varying runtimes and complexities before choosing the GumTree approach. We describe below these alternative approaches and highlight their shortcomings.

We initially began investigating ways to achieve log statement propagation that were Git reliant because FlorET relies on Git to handle version control of code, data, and checkpoints. Due to FlorET's versioning, each training session run with Flor involves tracking the histories of the code, data, and checkpoints. As such, the log statement the model developer later inserts on top of the most recent code commit can be cherry-picked and applied to prior code versions. Though straightforward and having low-overhead, this approach has a key drawback: cherry-picking a log statement from the current version to prior versions is prone to merge conflicts, exacerbated in cases where version-to-version diffs are adjacent to the location where the log statement is to be inserted. As with regular merge conflicts, the model developer could resolve them by hand, but this, to a large extent, nullifies our goal of making inter-version querying and debugging hands-free.

Due to similarities between our log statement propagation process and that of the creation and propagation of software patches, our next approach consisted of generating a "log patch" by applying the Myers diff algorithm [5] on the code. We then attempted to apply this patch to the previous code versions using the linux "patch" command, which takes into consideration the line number of the log statement as well as the context from adjacent lines.

However, similar to the Git reliant approach above, this approach often failed when lines directly before or after the log statement were modified between code versions.

A more robust solution involved the use of Google's best-effort patching algorithm, diff-match-patch (DMP) [7], which achieves code alignment by using a character error heuristic for identifying relevant code regions to apply the patch on. Since the diff-match-patch utility is language agnostic, changes in indentation between code versions introduced syntactic errors after applying the patch: the log statement propagated to prior versions is also inserted in a language and syntax agnostic manner. Therefore, we require an additional post-processing step to ensure syntactic correctness of the program. For most cases, this resulted in a simple and efficient solution, but the GumTree-variant propagation method still proved to be more resilient as shown in Chapter 5.2.

4.3 Parallelized Replay

Once we have propagated the developer-inserted log statement to prior versions, we must replay every version to answer the query and assist with analysis over execution data. Given the code, data, and checkpoints used for the initial execution of a particular model training run, re-execution of each version is independent. Hence, using Flor, we can replay each model training run on a separate core and with no overlapping overhead. This results in a linear scale-up, limited only by the number of cores the model developer has available.

Chapter 5

Results

In this section, we begin by detailing the methodology to evaluate the accuracy of our log statement propagation techniques, followed by an analysis of their robustness. We then assess the overhead of FlorET in the context of versioning, log statement propagation, and parallelized replay. Specifically, we will address the following in our evaluation:

1. What was the methodology used for testing the log statement propagation accuracy?
2. How do the log statement propagation algorithms compare with each other? What is their overall effectiveness?
3. What is the additional overhead incurred when using FlorET over Flor?

5.1 Synthetic and Ecological Benchmarks

Because the approach to use log statement propagation to enable analysis across time is a novel one, no benchmarks exist for testing accuracy or performance of this. We took a three-tiered approach when designing a set of baselines, consisting of:

1. **A diverse set of hand-constructed toy examples**. These include atomic programs that test a variety of program diffs including single-line insertions, single-line deletions, appending log statements to newly added conditionals, and variable renaming.
2. **A group of randomly generated variations of Pytorch programs**. Starting with a base Python program, we introduce an annotation over the program that allows three stochastic operations: (i) renaming variables, (ii) adding irrelevant statements (noise), (iii) randomly permuting lines. We then randomly generate multiple versions of the base program by applying these three operations in sequence.
3. **A set of popular codebases obtained through a Github scrape**. These include popular Pytorch examples from GitHub, looking across a number of different versions across time, particularly a few smaller repos [13] and a large industrial codebase [6].

The first two are part of the synthetic benchmark suite, while the latter is ecological.

We use synthetic benchmarks to provide the ability to experiment with contrived, challenging examples to conduct a more thorough investigation. On the other hand, ecological benchmarks have been used to focus on domain-specific examples. Because of the sheer variability in general Python programs, enabling log statement propagation efficiently is complex and perhaps infeasible in general. But by specializing on machine learning code, we hope to leverage the relatively simple control structures, like long-running training loops. Focusing on model training narrows the scope considerably, allowing us to measure the more relevant use cases.

5.2 Log Statement Propagation Accuracy

To evaluate the efficacy of log statement propagation, we test both the GumTree based technique and Myers-algorithm-based diff-match-patch technique on the synthetic and ecological benchmarks described above. The auto-generated benchmarks as well as both the small and large ecological benchmarks proved to be well-suited for log statement propagation over prior versions of code in the machine learning domain. As shown in Figure 5.1, the GumTree algorithm outperforms the Myers diff algorithm on the toy and auto-generated benchmarks primarily due to its robustness to code refactoring, specifically major logic changes, ambiguity from deletions, and variable renaming. In the ecological benchmark for large files, the Myers diff algorithm also falls behind its counterpart. This difference in the accuracy between the two was not due to the incorrect placement of the log statement in the file by the Myers diff algorithm, but due to it indenting the log statement incorrectly.

Figure 5.2 below summarizes key differences between the two algorithms on a couple of toy examples. The first example demonstrates the challenge that arises when there are considerable changes in the control flow from one version to the next. The expectation when propagating the log statement from version 2 to version 1 is that it is inserted after `i` is incremented, as shown in the column `V1-Ground truth`. The AST-based GumTree algorithm is able to infer this and insert the log statement correctly, while the Myers-diff-based algorithm is not able to.

The second example illustrates another challenge during log statement propagation i.e. ambiguity from deletions. To unambiguously propagate the log statement from version 2 to version 1, we need to know which of the two increments of `i` was deleted in version 1 to get version 2. Both the algorithms propagated the log statement but we cannot determine which propagation the model developer actually intended. This may not be a situation that a model developer frequently faces, but an important one to consider.

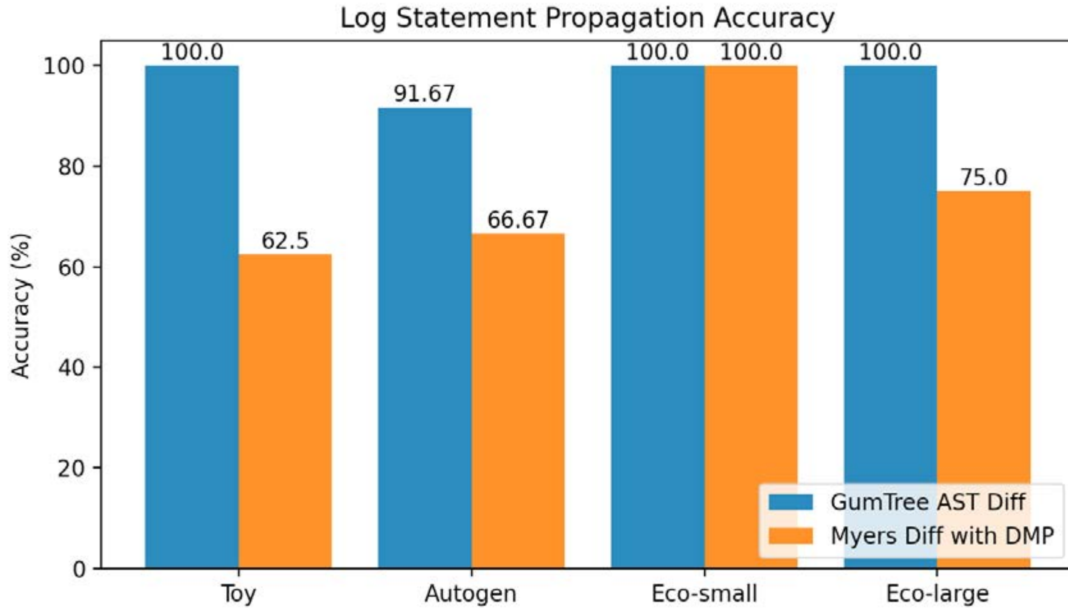


Figure 5.1: Histogram of accuracies of two log statement propagation techniques on the synthetic and ecological benchmarks. The ecological benchmarks are separated into eco-small and eco-large depending on the size of the file.

5.3 System Overhead

Overall, the FlorET system incurs minimal additional storage, compute, and latency overhead. We discuss below the specifics of the overhead incurred for performing versioning, log statement propagation, and parallel multiversion replay.

5.3.1 Versioning Overhead

FlorET achieves code, checkpoint, and data versioning with minimal running-time overhead. Code versioning is already fairly efficient as Git only snapshots modified files. Noting that time and storage are the factors contributing to the overhead, we conclude that both are negligible compared to the time and storage resources consumed by model training and hindsight logging with Flor.

FlorET does not incur any additional overhead from checkpoint versioning because collecting, materializing, and spooling checkpoints is already handled by Flor. Additionally, Flor's checkpoint spooling to S3 avoids long-term local storage overhead. Moreover, spooling to S3 is a non-blocking operation, so the model developer can avoid latency overhead by allocating a background process for this task.

V1	V1-Ground truth	V1-Myers	V1-GumTree	V2-Log stmt

i=0
if i == 0:
   i = i + 1
else:
   i = i - 1
	i=0
	if i == 0:
	   i = i + 1
	   print(i)
	else:
	   i = i - 1
		i=0
		if i == 0:
		   print(i)
		   i = i + 1
		else:
		   i = i - 1
			i = 0
			if i == 0:
			   i = i + 1
			   print(i)
			else:
			   i = i - 1
				i=0
				j=0
				a = "Hello"
				if i == 0:
				   if j == 0:
				      i = i + 1
				      j = j + 1
				      print(i)
				   else:
				      j = j - 1
				elif i == 1:
				      i = i + j
				else:
				      i = i - 1

i=0
if i == 0:
   i = i + 1
   i = i + 1
else:
   i = i - 1
	i=0
	if i == 0:
	   print(i)
	   i = i + 1
	   i = i + 1
	else:
	   i = i - 1
	
	OR
	
	i=0
	if i == 0:
	   i = i + 1
	   print(i)
	   i = i + 1
	else:
	   i = i - 1
		i=0
		if i == 0:
		   i = i + 1
		   print(i)
		   i = i + 1
		else:
		   i = i - 1
			i = 0
			if i == 0:
			   print(i)
			   i = i + 1
			   i = i + 1
			else:
			   i = i - 1
				i=0
				if i == 0:
				   print(i)
				   i = i + 1
				else:
				   i = i - 1

Figure 5.2: Two example code fragments illustrating the performance of the GumTree and Myers-diff based log statement propagation techniques over two code versions (`V1` and `V2`). The print statements in red represents log statements. The print statements in column `V2-Log stmt` are developer-inserted, while the ones in columns `V1-Myers` and `V1-GumTree` are algorithm-inserted.

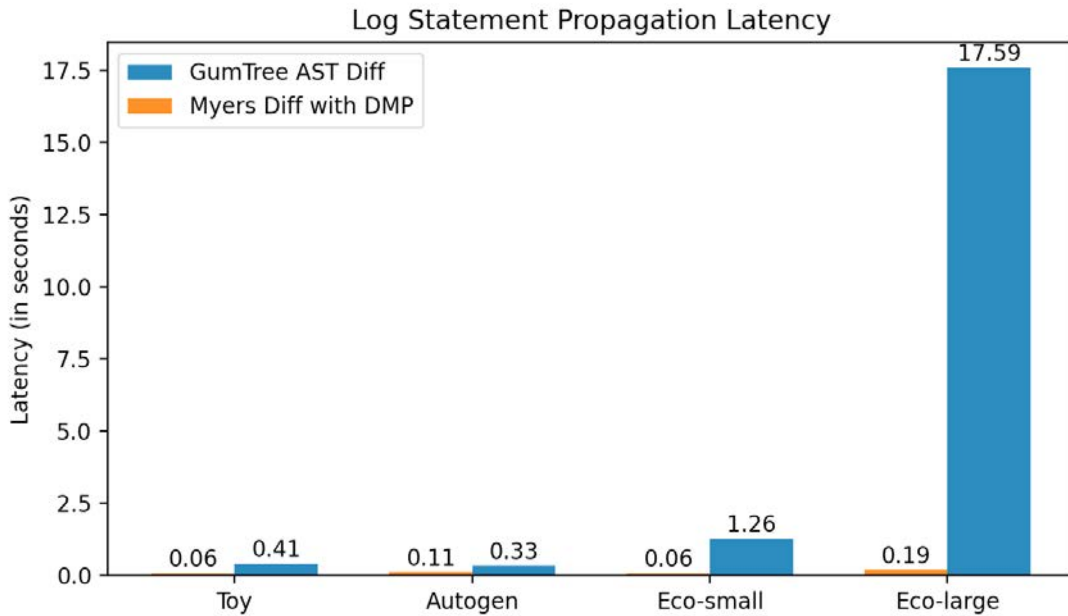


Figure 5.3: Latency incurred by performing log statement propagation on a single file to one previous version for the synthetic and ecological benchmarks. We performed log statement propagation for each of the categories for a total of five times and averaged the latencies. The latency of log statement propagation to a sequence of previous versions scales linearly with the number of versions.

FlorET handles data versioning with minimal time and storage overhead as well. Even though data versioning is performed once for each model training run, the dataset is not replicated and saved in its entirety each time. Only the S3 file location of the data is saved.

5.3.2 Log Statement Propagation Overhead

The algorithms used to achieve log statement propagation vary in both complexity and overhead. As briefly mentioned in Chapter 4.2, the Myers-diff-based diff-match-patch technique achieves quicker completion times than the AST-diff based GumTree algorithm when performing code alignment. However, the GumTree algorithm typically proves to be more robust to larger code refactors. Figure 5.3 shows the average completion times of both algorithms on a number of synthetic and ecological examples.

The toy and auto-generated examples we used have ∼10 and ∼100 lines of code in each file, respectively, while the ecological small and ecological large have ∼100-250 and 1200+ lines of code. As the file size increases, the task of performing code alignment and log statement propagation becomes increasingly time-consuming. However, the main training loop is where model developers usually insert their post-hoc log statements and the file size containing this

code is approximately the size of the examples in the ecological small benchmark. So model developers would incur a ∼1.26 second delay for every version they need to propagate the log statement to. If log statement propagation must be performed across large files, we can decompose the file in a way that the new decomposed file containing the log statement is smaller, thus limiting the latency overhead incurred.

It is important to note that if the model developer has to do log statement propagation across versions by hand, the time overhead would significantly exceed that of FlorET's automatic log statement propagation. Also, extending FlorET's capability by adding non-lazy code alignment, described in more detail in Chapter 6.1, could further reduce the latency of its log statement propagation engine.

5.3.3 Parallel Replay Overhead

FlorET's inter-version replay phase involves automatically running Flor's replay across multiple versions. The time overhead incurred here is no different than the model developer running Flor on the multiple versions independently. However, considering the enhanced capability of FlorET, which exploits the fact that the re-execution of each model training session is independent of the prior and subsequent sessions, it can achieve speedup, linear in the number of available cores, simply by allocating a core to each version that needs to be replayed.

Chapter 6

Future Work

We have thoroughly investigated and tested various approaches for semantic code alignment when developing FlorET. However, there still exists potential for further optimizations and additional features to improve the overall efficiency and usability of the system. In this section, we begin by describing optimizations that can be made to the current FlorET system. We follow by detailing the features for increasing the usability and efficiency such as query-cost highlighting and search and sampling, as previously mentioned in Chapter 3. Lastly, we discuss our plan to extend our work to support delta pruning for iterative model development.

6.1 Optimizations and Improvements

One important thing to note is that GumTree operates on many code versions. As such, increasing the level of parallelism as well as decreasing the memory overhead can make the system less prohibitive when working over these many versions.

We hope to incorporate the following changes in the future to facilitate the goals above:

1. **Parallelize the GumTree algorithm**. By parallelizing the GumTree algorithm, we can further limit the latency the model developer faces during log statement propagation. As mentioned before, while the GumTree approach of log statement propagation is more robust than others, it does incur perceivable latency overhead when compared to simpler, more naive approaches. Hence, decreasing its completion time can make the tool more plausible for a large number of use cases.

2. **Non-lazy code alignment**. A large fraction of the latency when using FlorET comes from performing code alignment over a series of code versions. To reduce latency at query time, we can perform code alignment between the current version and its predecessor once the execution of model training completes. This operation can be made non-blocking via the use of a background process. With this modification, we

hypothesize that the typical model developer will not perceive any latency at execution time or query time in most cases.

In addition, since our log statement propagation relies only on the end syntactic structure of the code before attempting to propagate semantic changes, we may encounter a number of edge cases. In particular, heavy use of metaprogramming, pervasive changes that affect a high percentage of lines, or subtle changes that affect control or data flow can affect the insertion or correctness of log statements. To increase robustness in the face of complex code versions and data lineages, we plan to investigate potential modifications to our code alignment algorithm with a goal to enable hindsight logging extending time for any ML-specific Python code. Furthermore, considering that the manual counterpart of automated log statement propagation is error-prone, we believe the benefits of automation outweigh the costs.

6.2 Usability-Enabling Features

FlorET provides inter-version querying and debugging capabilities with minimal effort from the model developer. We can further increase the usability of FlorET by prioritizing its ease of operation. We plan on developing a complementary extension to facilitate the querying and debugging process by the model developer.

6.2.1 IDE Integration for Version Navigation

To support a model developer in propagating log statements over a history of changes and verifying their placement, we plan to integrate the automatic log statement propagation facility into the model developer's IDE. We envision this integration to allow the model developer to more easily inspect and dynamically adjust the placement of log statements, to exclude code versions from code alignment and propagation, as well as to export logs produced by FlorET's replay for manually inspecting and visualizing. Figure 6.1 illustrates the proposed design.

6.3 Efficiency-Enabling Features

Another important goal of FlorET has been to enable debugging in an efficient manner. We plan to continue working on this goal by adding new efficiency-centric features, which we discuss below.

6.3.1 Query-Cost Highlighting

With the current FlorET system, we can efficiently query any execution data across a history of code versions. However, FlorET provides no insight to the model developer about the cost

```
540            if self.tpu and self.data_parallel_world_size > 1:
541                import torch_xla.core.xla_model as xm
542                gradients = xm._fetch_gradients(self.optimizer.optimizer)
543                xm.all_reduce('sum', gradients, scale=1.0 / self.data_parallel_world_size)
544
545            with torch.autograd.profiler.record_function("multiply-grads"):
546                # multiply gradients by (# GPUs / sample_size) since DDP
547                # already normalizes by the number of GPUs. Thus we get
548                # (sum_of_gradients / sample_size).
549                if not self.args.use_bmuf:
550                    self.optimizer.multiply_grads(self.data_parallel_world_size / sample_size)
551                elif sample_size > 0:  # BMUF needs to check sample size
552                    num = self.data_parallel_world_size if self._sync_stats() else 1
553                    self.optimizer.multiply_grads(num / sample_size)
554
555            with torch.autograd.profiler.record_function("clip-grads"):
556                # clip grads
557                grad_norm = self.clip_grad_norm(self.args.clip_norm)
558
559            # check that grad norms are consistent across workers
560            if (
561                not self.args.use_bmuf
562                and self.args.distributed_wrapper != 'SlowMo'
563                and not self.tpu
564            ):
565                self._check_grad_norms(grad_norm)
566
567            with torch.autograd.profiler.record_function("optimizer"):
568                # take an optimization step
569                self.optimizer.step()
570                print("LOG STMT: Model Parameters = %s" % (str(self.model.parameters)))
571
572        except FloatingPointError:
573            # re-run the forward and backward pass with hooks attached to print
574            # out where it fails
575            with NanDetector(self.model):
576                self.task.train_step(
577                    sample, self.model, self.criterion, self.optimizer, self.get_num_updates(),
578                    ignore_grad=False
579                )
580            raise
581        except OverflowError as e:
582            overflow = True
583            logger.info("NOTE: overflow detected, " + str(e))
584            grad_norm = torch.tensor(0.).cuda()
585            self.zero_grad()
586        except RuntimeError as e:
587            if "out of memory" in str(e):
588                self._log_oom(e)
589                logger.error("OOM during optimization, irrecoverable")
```

Controls
Jump to Next Change | Export Results | Propagate Changes

Nov 29 (12:52pm)
v1 v2 v3 v4

Figure 6.1: Proposed design of the IDE integration. The model developer can select any version (bottom blue bar) and inspect the placement of the log statements, dynamically adjust the placement of the log statement by dragging the newly inserted statement (in green) to the desired location, and export the query results (top right toolbar) as a table or as a CSV file.

of each potential query. If the model developer is able to estimate the cost of querying some specific execution data, they would then be able to make a more informed decision when selecting the appropriate query for any given problem. In other words, the model developer could select low cost queries over expensive alternatives. An example of this would be querying for the gradient magnitudes once per epoch instead of querying for tensors such as the weights, gradients, or activations in every update step. By color-coding execution data to indicate the cost of querying it, we can guide the model developer to lower cost queries, as shown in Figure 6.2. We plan to utilize Flor's capability of calculating replay latency factored on the position of the hindsight logging statements in the model training code to enable FlorET's query-cost estimation.

6.3.2 Search and Sampling

There may be times when a model developer may not require full replay of model training to gain the insight necessary to debug a problem. In such cases, search and sampling methods can be a much more efficient alternative than replaying the entirety of code from checkpoints for a number of model training sessions. FlorET currently does not support such an alternative, but considering how Flor enables parallel replay via checkpoints, extending FlorET to

```
for epoch in range(settings.EPOCH):
        net.train()
        for batch_index, (images, labels) in enumerate(cifar100_training_loader):
                clr_scheduler.step()
                images = Variable(images)
                labels = Variable(labels)

                optimizer.zero_grad()
                outputs = net(images)
                loss.backward()
                optimizer.step()
                print("LOG STMT: Gradients = %s" % net.parameters().gradients)
        print("LOG STMT: Gradient magnitudes = %s" % net.parameters().gradient_magnitudes)
        loss, acc = eval_training(epoch)

print('Test set: Average loss: {:.4f}, Accuracy: {:.4f}'.format(loss, acc))
```

Figure 6.2: Query-cost highlighting. The pseudocode above illustrates that the cost of querying the gradient magnitudes once per epoch (in green) is significantly lower than querying for the gradients every update step (in red).

support search and sampling optimizations over these checkpoints is a viable option. In other words, we can search and sample over the checkpoints to determine which, if any, epochs and/or versions need to be re-executed for anomaly detection because (i) replay execution from one checkpoint to the next is independent of all other segments of model training and (ii) replay execution of one version is independent of all others. Performing binary search over checkpoints and checkpoint sampling serve to help narrow down problematic portions of model training with greater speed. We plan to support this capability for enabling more efficient detection of anomalies in key execution data.

6.4 Delta Pruning

FlorET currently enables model developers to perform post-hoc logging across a number of code versions. However, once model developers have resolved the issue after debugging via FlorET, making a new change, and retraining the model, they have no way of selecting the minimum viable set of changes that achieve approximately identical results. Model developers cannot discard the code changes that are irrelevant to model accuracy because FlorET currently does not support delta pruning, where the code changes that do not positively contribute to expected results are dropped while only the promising ones are kept.

Being able to identify a subset of changes in the model training code becomes particularly useful in giving the model developer insight into the fundamentals of model training in addition to providing certain intuition into when a similar problem reoccurs in the future. The value of adding delta pruning to FlorET lies in the fact that no tools providing delta

pruning currently exist for us to use. A couple of approaches we hope to explore to support delta pruning include changeset analysis and changeset quality estimation via deep learning methods.

Chapter 7

Conclusion

Logging and debugging practices for ML model development need to be different than those used for normal software development. With model training runtimes being lengthy and model training failing silently and requiring far more than syntactic correctness to succeed, standard logging and debugging practices prove insufficient in most cases. Saving checkpoints during model training and achieving context recovery via checkpoint resume proved to be a more viable approach. This approach, called hindsight logging, is not only well-suited for model debugging, but also enables post-hoc data restoration efficiently via the record-replay phases. However, when debugging complex problems that could span multiple versions in the project version tree, hindsight logging – restricted to a single code version – can be limiting. Performing hindsight logging for each version separately does not scale well, so we facilitated the process of multiversion hindsight logging.

In this paper, we introduced the FlorET system for empowering model developers to perform hands-free analysis and debugging, by enabling efficient record-replay of model training over the evolution of projects. The key features that facilitate this advancement consist of (i) automatic version control, (ii) robust log statement propagation, and (iii) parallelized inter-version replay. We evaluated the efficacy of the log statement propagation and code alignment techniques requisite for temporal hindsight logging. We analyzed the overhead incurred by the version control engine, the log statement propagation engine, and the inter-version parallelized replay and found the performance overhead to be acceptable.

Bibliography

[1] Falleri et al. "Fine-grained and Accurate Source Code Differencing". In: *Proceedings of the International Conference on Automated Software Engineering* (2014), pp. 313–324.

[2] Hellerstein et al. "Serverless Computing: One Step Forward, Two Steps Back". In: *Proceedings in CIDR* (2019).

[3] Rolando Garcia et al. "Hindsight Logging for Model Training". In: *Proceedings of the VLDB Endowment* (2020), pp. 682–693.

[4] Amazon. *AWS*. URL: https://aws.amazon.com/.

[5] Nicholas Butler. *Myer's Diff Algorithm: The basic greedy algorithm*. URL: http://simplygenius.net/Article/DiffTutorial1.

[6] Facebook. *Fairseq*. URL: https://github.com/pytorch/fairseq.

[7] Google. *Diff-Match-Patch*. URL: https://github.com/google/diff-match-patch.

[8] Google. *Google Cloud Platform*. URL: https://cloud.google.com/.

[9] Google. *Tensorboard*. URL: tensorflow.org/tensorboard.

[10] Andrej Karpathy. *A Recipe for Training Neural Networks*. URL: http://karpathy.github.io/2019/04/25/recipe/.

[11] Microsoft. *Azure*. URL: https://azure.microsoft.com/en-us/.

[12] Joseph Misiti. *Awesome Machine Learning*. URL: https://github.com/josephmisiti/awesome-machine-learning.

[13] *Pytorch Examples*. URL: https://github.com/pytorch/examples.

[14] *Weight Biases*. URL: wandb.com.