Low Overhead Materialization with Flor

Eric Liu

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2020-79
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2020/EECS-2020-79.html

May 28, 2020

Low Overhead Materialization with Flor

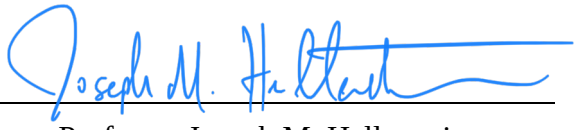
by Eric Liu

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 26, 2020

(Date)

* * * * * * *

Professor Joseph Gonzalez
Second Reader

May 27, 2020

(Date)

Low Overhead Materialization with Flor

Abstract

Low Overhead Materialization with Flor

by

Eric Liu

Master of Science in Electrical Engineering and Computer Science

University of California, Berkeley

Professor Joseph M. Hellerstein,

Research in AI is dominated by model experimentation, and training a single model can be extremely expensive. However, there is no efficient way of recovering information if something goes wrong or the model behaves differently from expected. Flor is a system that provides a record-replay approach to ML training, allowing developers the flexibility of retrieving the data they want after execution. It takes intermittent checkpoints during model training to help speed up and parallelize replay. However, achieving a record-replay system requires expensive checkpoints to be serialized during program execution, causing high overheads and making this solution less palatable.

Flor is able to achieve a low overhead, fast materialization process by using its multiprocessing materializer, a solution that takes advantage of parallelism to offload the burden of serialization to other processes. The multiprocessing materializer uses forking to both spawn processes and provide one-way interprocess communication (IPC), allowing Flor to quickly share and serialize expensive checkpoints. We also show that our multiprocessing materializer performs better than other popular methods of multiprocessing and IPC. Notably, this method achieves checkpointing at only an additional 1.74% runtime cost.

To my parents and brothers

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Modern machine learning workloads involve increasingly sophisticated models and pose challenges from a data management standpoint [11]. As models grow in size and complexity, training runtimes can exceed days or even weeks [2], while bugs remain hidden until developers realize the resulting model performs poorly in tests or during deployment. Because of the staggering cost of training models, typical debugging approaches are highly inefficient, leaving ML developers to approach debugging with logging. Logging frequently results in large sets of execution data, which must be gathered and analyzed with domain specific tools like TensorBoard [3], MLflow [14], and WandB [13]. Unfortunately, this style of debugging requires the developer to have a certain omniscience about their model; they must anticipate where bugs may occur, forcing them to insert necessary log statements in advance.

Furthermore, if the ML developer fails to anticipate even a minor bug, recovering the relevant data needed for debugging requires a complete execution. Since re-execution can be expensive in time and resources, developers will often try to make educated guesses as to why their code behaves differently than expected and attempt to fix their training code based on their hypotheses. Guessing incorrectly could render the debugging process even more inefficient than simply running their code with logging statements.

Since bugs should always be expected, we need new ML debugging techniques that are a better fit for runtime and cost constraints. As a result, data-driven debugging techniques should be driven by *hindsight logging*, a novel method of recovering any unanticipated runtime information. Efficient hindsight logging can enable developers to code freely without the burden of foresight at what needs to be logged while maintaining the ability to perform data-driven debugging later. However, this feature should not come at the cost of additional runtime. A naive technique for achieving hindsight logging is to record as much program state as possible during training; extracting logs from these traces allows for data on demand and has a comparatively short runtime. The drawback with this strategy is its unacceptably high cost of logging program state. In fact, this method incurred over a 1000× slowdown recording ResNet-18 training on the Cifar100 dataset. What is needed is a hindsight logging mechanism that:

1. Incurs minimal logging overhead at runtime.

2. Enables low-latency extraction of execution data during debugging or analysis.

To achieve efficient hindsight logging of Python-based ML training, we created a record-replay system called Flor. In this framework, model training corresponds to the record phase, and analysis corresponds with the replay phase. Flor automatically and transparently logs program state during the record phase, using program analysis to prune unnecessary state capture. When model developers want to analyze their code, they can add any log statements they want to the code and replay the code to retrieve the desired data. In most cases, Flor is able to quickly retrieve the desired data with partial replay: it loads state from the nearest captured checkpoint and executes the code to get values. Even in the worst case scenario, when complete re-execution is needed to retrieve values, Flor can substantially reduce the cost of replay through parallelism.

Within Flor, we face the significant technical challenge of reducing the overhead of checkpointing during the record phase. There are three key insights that enable Flor to achieve a low overhead, low latency record-replay system. First, Flor has conservative checkpointing policies as well as an adaptive checkpointing mechanism, which combine to provide a logging system that logs as little as possible. Second, Flor's multiprocessing materializer provides checkpointing at low costs, ensuring that the objects we do log are done so with minimal overhead. Third, autoparallelized re-execution of training, which is enabled by checkpointing, allows for greatly increased worst-case replay performance.

In this paper, we introduce the architecture of Flor as background, then focus on the multiprocessing materializer used in its record phase. We discuss the design of the materializer in detail and explain how it is able to achieve minimal overheads while consuming as few resources as possible. To evaluate the effectiveness of our materializer, we measure the runtime of Flor on a range of computer vision and NLP benchmarks of various scales. In total, we find that Flor captures all of the necessary checkpoints while accounting for just 1.74% of the total runtime during training.

Chapter 2

Background

Model training bugs are different from typical software bugs since they often fail without crashing. Models can easily conceal errors until after deployment, and the root cause for overfitting or underfitting models is unclear in most cases. Here, we present an example case where non-decreasing losses indicate failure, but fixing the problem involves deeper, root-cause analysis and many re-executions of the training process.

2.1 Motivating Example

In this example, our model developer, Alice, will attempt to implement *super-convergence* [10] after successfully training an off-the-shelf model on a standard vision benchmark. Super-convergence is a state-of-the-art technique that can reduce training time by an order of magnitude.

To get started, Alice has access to a number of off-the-shelf model training pipelines and she selects a pipeline that trains ResNet-18 on the CIFAR-100 dataset. She begins training

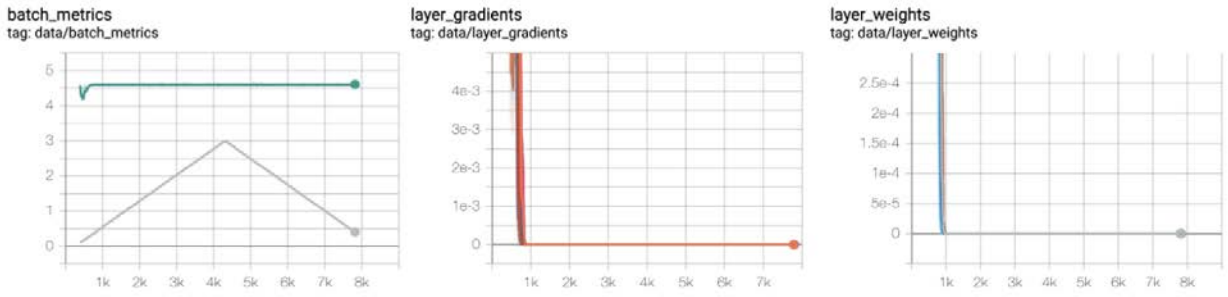


Figure 2.1: Plots of ResNet-18 batch metrics (loss and learning rate), layer gradients, and layer weights versus time steps

the model and reaches 80% test set accuracy in an hour. Now that she is confident that the pipeline works, Alice implements super-convergence and begins training the model again. In the first run with the modified pipeline, the model finishes training in six minutes without crashing, but has a test accuracy of around 50%.

Like most ML developers, Alice already tracks the loss, accuracy, and learning rate throughout training. In the leftmost plot of Figure 2.1, we can see the average loss and learning rate versus time steps. The learning rate over time matches her expectation for the schedule [10], so she concludes that the scheduler is modifying the learning rate properly. Alice also notices that the loss starts to decrease early on in training, but then increases shortly thereafter. She realizes that if the learning rate is greater than 0.2, the loss increases and stays constant for the remainder of training (Figure 2.1). Therefore, Alice knows that the pipeline behaves well for small learning rates, but is currently incompatible with super-convergence.

Alice starts by inspecting the gradients for each layer in her model (Figure 2.1). In this case, she adds TensorBoard statements to the model and performs a second training run. By doing this, she sees the gradients become equal to zero at the same time the loss becomes constant. Upon closer inspection, Alice notices that the gradients spike first before plummeting. If the magnitude of the weights are controlled by regularization, then the weights will be forced to a magnitude near zero given the high learning rate. To confirm the hypothesis, Alice runs the experiment a third time, logging the weights for every layer, and finds that the weights are quickly driven close to zero (Figure 2.1). Armed with this knowledge, Alice disables weight decay and retrains for a fourth time. This time, her model reaches a test accuracy of 70% after six minutes.

2.2 Constraints and Limitations

In the example, we show the benefits of being able to capture additional execution data after runtime. Alice ran her model training code a total of four times: once during her first execution, twice more to retrieve relevant debugging information, and once after she fixed the problem. If Alice had been able to log everything she needed, she could have avoided executing the second and third training runs. Alice's need for unanticipated information in the form of gradients and weights forced her to re-run the experiment twice and consume additional resources.

However, the difference between no logging and logging everything poses an issue for the model developer. Running models without additional logging incurs no overhead and is not a problem if there are no bugs in the code. However, if unforeseen issues do occur and necessary debugging data is missing, re-training at least one additional time is necessary. This lends credibility to the logging approach, but the overhead of logging can be many times that of a single training run. This cost can be mitigated if a smaller subset of items is logged to disk, but there will always be a risk of not collecting sufficient data for analysis.

An ideal solution would have low execution overhead and enable fast recovery of execution data.

To achieve this ideal scenario, we must abandon the route of logging all execution data proactively. Any overhead that we incur must be small and always accompanied by the promise of reduction in latency during analysis. This report focuses on the multiprocessing materializer, Flor's method of meeting this overhead requirement. The rest of the paper is sectioned as follows: Section 3 provides an overview of Flor; Section 4 presents the record architecture in detail, including alternative architectures that were considered; Section 5 contains our performance review; Section 6 is a discussion of the results; and Sections 7 and 8 contain related works and future work, respectively.

Chapter 3

System Overview

The objective of hindsight logging is to allow model developers the flexibility of choosing what they want to log *after* execution. This means that we must be able to quickly retrieve or recreate the values or objects that they want to probe. If these objects and values are logged to disk, we can simply read the value from storage; otherwise, they must be recomputed by rerunning the code. However, both of these methods have expensive drawbacks. Dumping full execution traces to disk is expensive in both runtime and storage footprint. Recomputing values requires re-execution of model training, which takes a long time and costs a substantial amount of money.

In this section, we give an overview of Flor's record and replay phases. Flor is able to avoid much of the runtime and storage overheads by using a hybrid strategy involving logging and re-execution. Because model training is inherently iterative, we can provide record-replay capabilities and break cross-iteration dependencies by capturing the side-effects of loop iterations. Algorithm 1 gives an abstract model of training, which is representative of a large number of deep learning workloads.

During execution, Flor selectively captures state at certain checkpoints, reducing the amount of logging we incur up front. Flor also parallelizes this logging process with its multiprocessing materializer to decrease logging cost. During analysis, Flor recovers the state from disk and continues execution from the checkpoint. This allows it to skip much of the recomputation and achieve order-of-magnitude speed-ups when compared to retraining from scratch. The speed-ups are further increased when multiple GPUs are available, as Flor can use checkpoints to autoparallelize re-execution proportional to the number of GPUs. In summary, Flor transparently uses state captured in the background during model training to gain speed-ups during analysis.

3.1 Record Phase

The record phase in Flor efficiently captures checkpoints during execution to enable future replay by tracking loops and checkpointing their side effects. It can automatically analyze

Algorithm 2: Abstract Model Training Example, after Flor transformations.

```
Result: Net fits Training Data
initialize();
for epoch ← 1 to N do
    if not flor.SKIP then
        for batch ∈ training_data do
            predictions = net(batch.X);
            avg_loss = loss(predictions, batch.y);
            avg_loss.backward();
            optimizer.step();
        end
    end
    flor.proc_side_effects(net, optimizer);
    evaluate(net, test_data);
end
```

Algorithm 1: Abstract Model Training Example, in PyTorch

```
Result: Net fits Training Data
initialize();
for epoch ← 1 to N do
    /* Epoch Start Checkpoint ************** */
    for batch ∈ training_data do
        predictions = net(batch.X);
        avg_loss = loss(predictions, batch.y);
        avg_loss.backward();
        optimizer.step();
    end
    /* Loop End Checkpoint ***************** */
    evaluate(net, test_data);
end
```

Figure 3.1: Code snippets showing before and after Flor transformation

loops to find the set of all values and objects modified during the loop; these objects become part of the Loop End Checkpoint. We show two code snippets in Figure 3.1, in which Algorithm 2 is a simplified version of how Flor would transform Algorithm 1.

Flor follows several policies that ensure it only stores the state that is necessary. Instead of taking Epoch Start Checkpoints, as annotated in Algorithm 1, Flor can reconstruct these checkpoints using previous epochs. Because Loop End Checkpoints are still logged, we can quickly load these checkpoints to restore the desired Epoch Start Checkpoint. The Loop Start Checkpoints become logical checkpoints that can be recomputed with the physical Loop End Checkpoints. Furthermore, Flor is not required to capture full program state at any checkpoint, since we only need to keep track of the states that are modified inside the loop. For example, Algorithm 1 only modifies the network and the optimizer, meaning that capturing the states of both is sufficient as a Loop End Checkpoint.

Additionally, Flor's automatic transformer eliminates the need for manual transformation of code, making the process less laborious and less error prone. To facilitate this, Flor performs best-effort static analysis to evaluate the side-effects of loops, since it can be done extremely quickly and allows us to avoid expensive runtime checks. In the event Flor fails to capture side-effects for the loop, Flor can easily fall back on a re-execution of the required loop.

Alongside these checkpointing decisions, Flor applies both adaptive checkpointing and the multiprocessing materializer as mechanisms for speeding up the checkpointing process. While the materialization cost of loop side-effects are typically negligible compared to the time it takes to train models, there are exceptions for extremely large models or short loop iterations. In the latter case, adaptive checkpointing reduces the worst case runtime by allowing Flor to ignore the Loop End Checkpoint if the cost of materialization dominates the cost of training. Even in this case, Flor will still take intermittent checkpoints to enable

parallel replay later down the line. The multiprocessing materializer is a technique that facilitates fast, parallel checkpointing (Section 4).

3.2 Replay Phase

When model developers want to retrieve objects with hindsight logging, they can simply add log statements into their code and re-execute with Flor. The checkpoints that Flor takes during its record phase break cross-iteration dependencies, enabling partial replay and parallel replay. These two mechanisms allow users to recover execution data without having to undergo the entire model training process.

Partial replay uses the checkpoints collected during the record phase to enable replay of specific epochs, reducing runtime by skipping execution of unnecessary epochs. To skip an epoch, Flor simply reloads the Loop End Checkpoint for that epoch. Using the example Algorithm 1 and its corresponding transformation in Algorithm 2, Flor's re-execution phase would involve reloading the checkpointed net and optimizer states. Flor will continue skipping loops unless the user has probed a value from the body of the loop. Partial replay is not always possible, but can help decrease the total amount of work that needs to be done.

Flor can further decrease replay times by using parallel replay, which leverages any additional available GPUs to simultaneously execute many epochs. The GPUs are not required to be on the same machine, since the checkpoints that Flor takes require no coordination to be replayed. Flor will automatically parallelize re-execution by splitting the work done by the outermost training loop into partitions. Each GPU gets assigned a Start Checkpoint and executes its share of work independently.

Chapter 4

Record Architecture

Now that Flor has been described at a high level, we address the multiprocessing materializer, a key mechanism that enables low overhead recording in Flor. Even though Flor avoids unnecessary checkpoints as much as possible, checkpointing can still be an expensive operation. The multiprocessing materializer helps mitigate the cost of checkpointing and prevents it from becoming a heavy burden on the machine learning process. This section provides a detailed description of the record architecture, with a focus on the development of the multiprocessing materializer.

4.1 Background and Challenges

When Flor saves a Loop End Checkpoint, it must materialize all the objects that have been modified during the loop. Materialization entails serializing the object into bytes and writing the bytes to disk. Before the development of the multiprocessing materializer, Flor used the cloudpickle module in Python to serialize objects. Using cloudpickle to immediately materialize objects results in large latency overheads because the ML workload is being processed by the same thread, forcing the training code to wait for logging to complete. However, since modern microprocessors all have multiple CPUs, leveraging other CPUs to perform materialization would decrease total overhead. Achieving this level of parallelism is made difficult by Python's Global Interpreter Lock (GIL), and different methods of avoiding the GIL come with different challenges. In this subsection, we discuss mechanisms for achieving parallel logging in Python as well as their drawbacks. We culminate with a description of the multiprocessing materializer, our mechanism that enables Flor to efficiently serialize and write objects to disk.

An immediately obvious solution for this problem of materialization would be to create a separate thread for serialization and I/O. Since threads exist within the same process and operate in the same memory space, we do not have to perform IPC to share objects. Threads are also ideal for isolating I/O bound tasks, allowing the main task to proceed without having to wait. However, we would only be able to gain speedups during I/O

because serialization is unable to be parallelized with this mechanism. The GIL only allows one thread per process to execute Python bytecode at any time; this protects Python objects from being modified simultaneously, as memory management in CPython is not thread safe. Additionally, we performed an experiment measuring the time to serialize and write a 1.1GB RoBERTa checkpoint that was taken during training on the RTE dataset. We found that serialization accounted for 77% of the total runtime. Because so much time is spent on serialization, the threads we spawn for materialization have limited capacity for parallelism and spend most of their time executing serially.

We cannot use Python threads if we want to avoid the limitations of the GIL. Instead, we have to utilize multiprocessing and create additional worker processes that serialize and write data to disk. Because each Python process has its own instance of the interpreter and the GIL, we no longer deal with threading issues and blocking parallelism. Each process also has its own memory space, making it impossible for one process to access objects belonging to another process. One solution is to share objects between processes using Python's shared memory module, but it is limited to specific data types and is relatively inconvenient to use. The shared memory module allows processes to access shared objects, which take the form of a byte array buffer or a shared memory list. The byte array must be declared in advance and be given a fixed size, making it difficult to share multiple objects of different sizes. Furthermore, because it can only store bytes, we still have to perform some serialization before inserting into the buffer. A shared memory list is easier to use, as it essentially allows a Python list to exist in shared memory. Unfortunately, like the byte array buffer, we are unable to expand the list or resize any of the elements inside the list. All in all, the inflexibility offered by Python's shared memory solutions makes it infeasible as a general purpose IPC mechanism.

A common way of communicating information between two Python processes is with queues from the *multiprocessing* module. These queues are built on top of two-way pipes and come in two flavors: Multiprocessing Queue and Simple Queue. Both are FIFO queues that send serialized objects through the pipe. The only difference between the two is that the Multiprocessing Queue uses multithreading, creating a new thread to handle queue insertions. Any processes that have access to the same instance of these queues can read objects from or write objects into it. To use these queues, we first spawn a process with Python's multiprocessing module that takes an instance of the desired queue as an argument. The sole purpose of this spawned process is to monitor for new objects, which it then pops from the queue, serializes with pickle, and writes to disk. When the ML process finishes, it waits for the worker process to finish as well before joining and terminating.

The issue with both the Multiprocessing Queue and the Simple Queue is that they are still dependent on serialization. Because they are built on top of Posix pipes, the queues require objects to be turned into byte streams before sent through. This means that the ML process still has to be suspended in order for the objects to be serialized. With Simple Queue, the main thread conducts serialization immediately, while with the Multiprocessing Queue the main thread will eventually sleep and let the serialization thread proceed. To compare the performance of the queues, we sent previously collected Flor checkpoints through the two

and recorded the runtime. The checkpoints were the same 1.1GB RoBERTA checkpoints that we used to test serialization overhead. We found that the Multiprocessing Queue took 1.377 seconds to send a checkpoint, while the Simple Queue took 1.48 seconds. Because of its slightly better performance, we use the Multiprocessing Queue when comparing to other methods. In the end, both queues slightly outperform cloudpickle, but fall short compared to the other alternatives we investigated.

Another method we explored is the Plasma object store, a data store developed at UC Berkeley and adopted by Apache. Plasma uses Apache Arrow's in-memory serialization format to represent Python objects, which allows these objects to be zero-copy shared once they have been inserted into the store. Its serialization process is also quite specialized; it performs exceptionally well on large numerical objects, but poorly on generic Python objects. The implementation of multiprocessing with Plasma is similar to that of the Multiprocessing Queue. Before we start the ML task, we need to create a new process that runs the Plasma object store. Next, in the ML code, we create a worker process that connects to Plasma, waits for new objects to be inserted, and writes the objects to disk.

Plasma is still suboptimal for several reasons. As previously noted, it is specialized to serialize numerical data very well. The reason for this specialization is the Arrow object format. Plasma converts objects into Arrow format, but falls back on pickle to serialize some of the objects that Arrow does not support [8]. However, we have observed that there are still objects that it can't serialize, including Pytorch tensors. In these cases, the user must register a custom serializer to handle these objects, which hampers usability and speed. Another downside of Plasma is its purpose as an object store instead of an IPC mechanism, which can create usability issues. Plasma must be instantiated and served on a separate process, and ML developers must determine the amount of memory to allocate to Plasma in advance of training.

Finally, we tested variants of forking as methods of both spawning processes and sharing memory. Using the fork function from the OS module invokes the fork syscall, which creates a child process that can be directed to serialize objects and write them to disk. The advantage of this method is that the operating system automatically copies the parent's memory space, effectively allowing us to perform one-way communication between processes without needing serialization. Furthermore, because forking is copy-on-write, it does not incur any immediate overhead; the only overhead we experience is a small, constant amount of additional runtime for instantiating a new process. In copy-on-write, memory pages belonging to the parent process are marked as read-only by the OS. If a process wants to modify an object stored on these pages, the OS automatically makes a copy. Ultimately, this method is the most appealing because it does not require serialization inside the parent process.

Looking at Figures 4.1 and 4.2, we can see why forking is such an appealing solution for checkpointing. Figure 4.1 represents the workflow of standard Python multiprocessing and IPC. We create one or more child processes that exist for the purpose of Disk I/O, and the main process uses some IPC mechanism to send data to these children. This is the type of architecture we employ when using the Multiprocessing Queue or Plasma. These methods are suboptimal because they entail parent-side serialization and impose a serialization burden

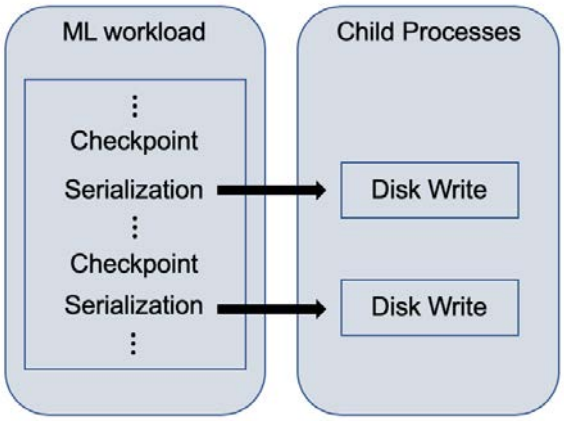


Figure 4.1: Architecture diagram for parent-side serialization

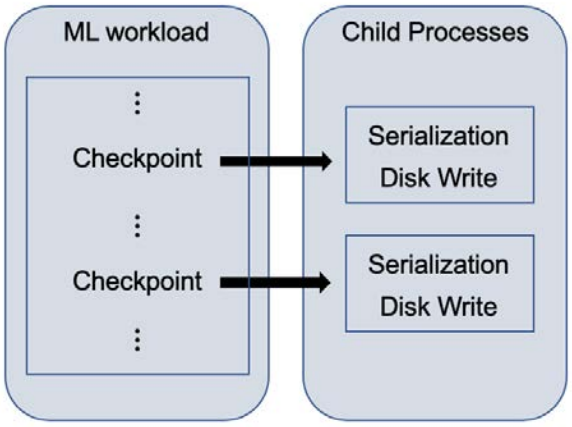


Figure 4.2: Architecture diagram for child-side serialization

on the same process handling the ML workload. On the other hand, forking allows us to utilize the architecture shown in Figure 4.2. In this case, we use fork to create child processes that handle serialization and disk I/O independently of the ML workload. Having child-side serialization allows the ML work to continue without interruption.

4.2 The Multiprocessing Materializer

Among the options we have discussed above, forking is the most attractive because its copy-on-write semantics enables IPC without serialization. Although this is a promising solution, there are still some drawbacks and challenges that need to be addressed to optimize its performance. This section describes the issues with fork and the modifications that were made to extract as much performance as possible from multiprocessing.

One side effect of forking that we observed early on is that child processes are not cleared from the OS process table unless the parent explicitly joins them. This leads to an interesting dilemma. If we don't clear children from the process table, the main machine learning process will be terminated when the user exceeds the OS process limit. However, calling join on every child process causes the main process to block and perform garbage collection on unused objects in memory, greatly increasing runtimes. We can solve this by ignoring SIGCHLD, the child termination signal. Python's signal module provides mechanisms for manipulating signal handlers, and we can simply set SIGCHLD as an ignored signal. Now, child processes that have finished executing no longer count as processes in the process table and are reaped by the kernel.

In addition to SIGCHLD, it is important to note that frequent forking can quickly cause thrashing. In early versions of the multiprocessing materializer, fork was invoked every time an object was to be logged. With some workloads, forking at this rate caused the accumulation of hundreds of additional processes that resulted in needless context switching, and a runtime that was even worse than Flor without multiprocessing.

The solution to this issue involves holding a buffer of checkpoints in memory. To prevent thrashing, Flor batches objects in a buffer so that every fork writes multiple objects instead of one. When the buffer reaches a certain size, Flor triggers a fork and the child process serializes all the batched objects. While the child process works on the objects, the main process simply resets the buffer to an empty state and resumes its task. Batching objects is a good way of reducing thrashing and lets us trade memory for speed. We have also determined in our experiments that a buffer size of 5000 objects is large enough to ensure that we don't thrash when forking.

One drawback to batching is that it involves keeping objects or pointers in a buffer for extended periods of time; these objects can be updated by the program and values can shift underneath us. If this happens, we would end up writing the newest version of objects to disk instead of the intermediate checkpoints we intended to capture. We avoid buffering live objects by performing a deep copy on objects every time they are inserted into the buffer. This solution ensures complete logging but comes at an additional runtime expense. Even

still, the cost of a deep copy is substantially lower than immediate serialization, so we still see overall performance gains.

Lastly, we must include special handling for GPU tensors. The most common objects we store are Pytorch states, which are often dictionaries containing large tensors. If these are GPU tensors, calling deep-copy on these dictionaries results in additional copies of tensors on GPU memory that cannot be accessed by children processes because of virtual addresses and memory protection. There is also the potential for GPU-out-of-memory errors, which can occur if we attempt to copy a tensor that is too big to fit in the remainder of GPU memory. To facilitate forking, Pytorch tensors that are stored on GPU must first be converted to CPU before the dictionary is deep copied. Fortunately, this conversion process automatically creates an isolated copy, which renders the deep copying process unnecessary. Thus, every state dictionary we capture must be walked through recursively to convert GPU tensors to CPU tensors and deep-copy every other object.

Chapter 5

Results

In this section, we evaluate the performance of the record architecture in Flor in the context of the multiprocessing materializer. We will address the following questions in our evaluation:

1. What is the total overhead of Flor?
2. How does the multiprocessing materializer compare against the alternative architectures?
3. How does buffer size affect Flor's record overhead?
4. Does thrashing make an impact on record overhead, and if so, how much?

5.1 Experimental Setup

In our experiments, we used a sample of eight different machine learning workloads that represented a broad range of model sizes, training times, and tasks. We selected models from different machine learning benchmarks to ensure Flor is capable of operating on a

Name	Benchmark	Task	Model	Dataset	Train or Tune	Epochs
RsNt	Classic CV	Image Classification	ResNet-152	Cifar100	Training	200
Cifr	Classic CV	Image Classification	Squeezenet	Cifar100	Training	200
ImgN	Classic CV	Image Classification	Squeezenet	ImageNet	Training	8
RTE	GLUE	RTE	RoBERTa	RTE	Fine-Tuning	200
CoLA	GLUE	Language Acceptability	RoBERTa	CoLA	Fine-Tuning	200
Wiki	GLUE	Language Modeling	RoBERTa	Wiki	Training	10
Jasp	MLPerf	Speech Recognition	Jasper	LibriSpeech	Training	4
RnnT	MLPerf	Language Translation	RNN w/ Attent.	WMT16	Training	8

Table 5.1: Summary of tested ML workloads

variety of ML workloads. Our models were selected from NLP tasks, computer vision tasks, the GLUE benchmark, and the MLPerf benchmark. We give a quick glance summary of all our workloads in Table 5.1.

All the experiments were implemented in Pytorch and ran on an AWS p3.8xlarge instance. The p3.8xlarge instance has 4 Tesla V100 GPUs, each with 16GB of RAM, 32 virtual CPUs, and 244GB of memory.

5.2 Record Overhead

Flor introduces minimal overhead during the record phase, as we show in Figure 5.1. This experiment tests the overall Flor overhead by running each ML workflow with and without Flor checkpointing. The green bar represents ML workloads with Flor enabled, while the blue bar represents ML workloads running without Flor. The average overhead for all these workloads is about 1.74%, which is a level that is virtually imperceptible to the user. This is largely due to Flor's conservative logging policies, which allow it to log a relatively small number of checkpoints. The checkpoints that are taken have their overheads mitigated by the multiprocessing materializer.

Certain workloads will have higher overheads than others. In particular, the overhead for CoLA, a fine-tuning workload, is higher than the other experiments. The Flor overhead for CoLA sits at 6%, which, while not horrible, is higher than the other experiments. This is likely because in fine-tuning workloads, most of the weights in the network are frozen and immutable, leaving only a few weights to be updated with a small set of training data. As such, fine-tuning is a very fast operation compared to training, but it still generates large model states. Flor performs relatively poorly in this case because the process spends very little time updating weights and a massive amount of time checkpointing model state.

We can also see a better breakdown of performance by looking at Figure 5.2. In this experiment, we compare the runtime of Flor with the multiprocessing materializer to Flor without. Flor without the multiprocessing materializer uses cloudpickle to serialize all objects, which gives us an indication of materializer performance without parallelism. The average overhead across all workloads using the cloudpickle materialization process is 4.76%, which is more than double the average overhead with multiprocessing, 1.74%. Utilizing multiprocessing yields measurable benefits, particularly for longer running experiments with larger or more checkpoints.

In total, we see very small overheads when using the multiprocessing materializer, ranging from about 0-2.5% in most cases. When we disable the multiprocessing materializer, we still get the benefit of Flor's conservative logging policies, but we see runtime overheads approaching 5%. Compared to the average cloudpickle overhead, the average multiprocessing materializer overhead was 73.5% lower.

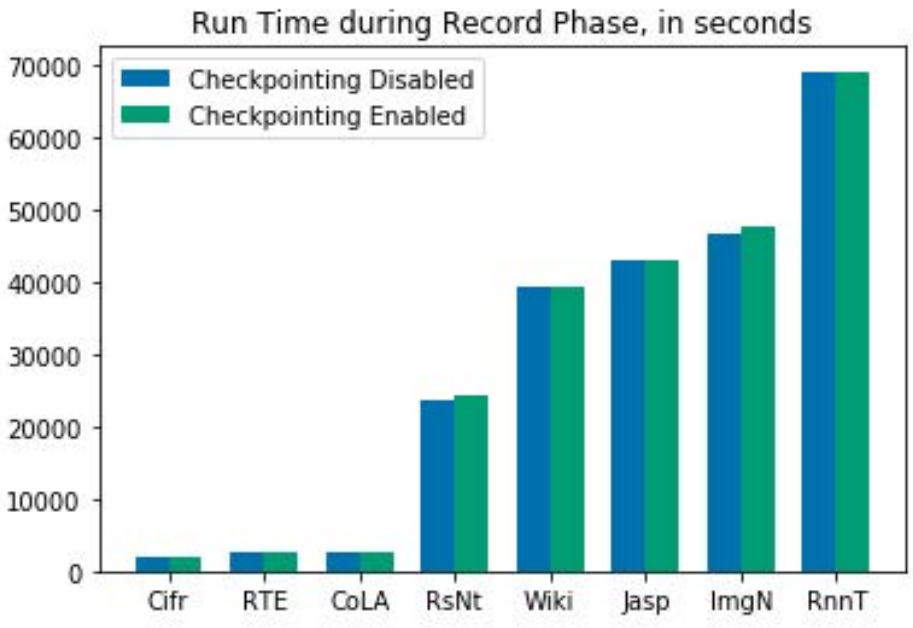


Figure 5.1: Comparison between ML experiments without Flor and ML experiments with Flor recording

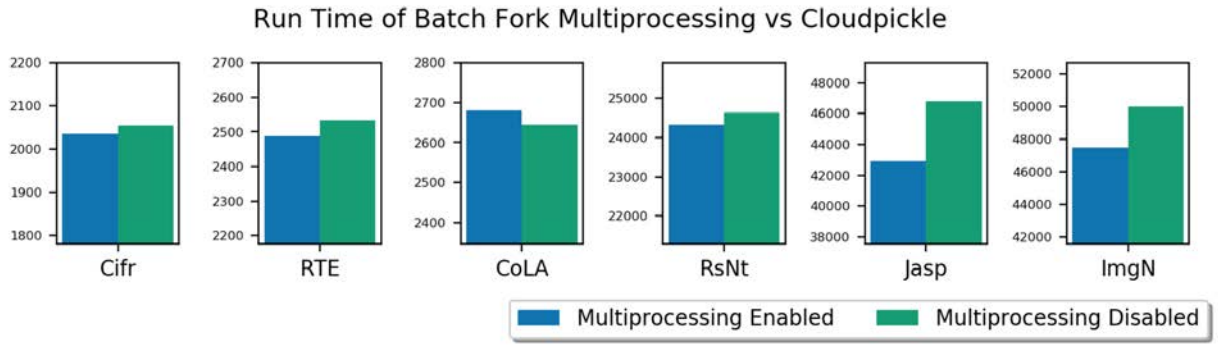


Figure 5.2: Flor runtimes during the record phase with the multiprocessing materializer enabled versus disabled

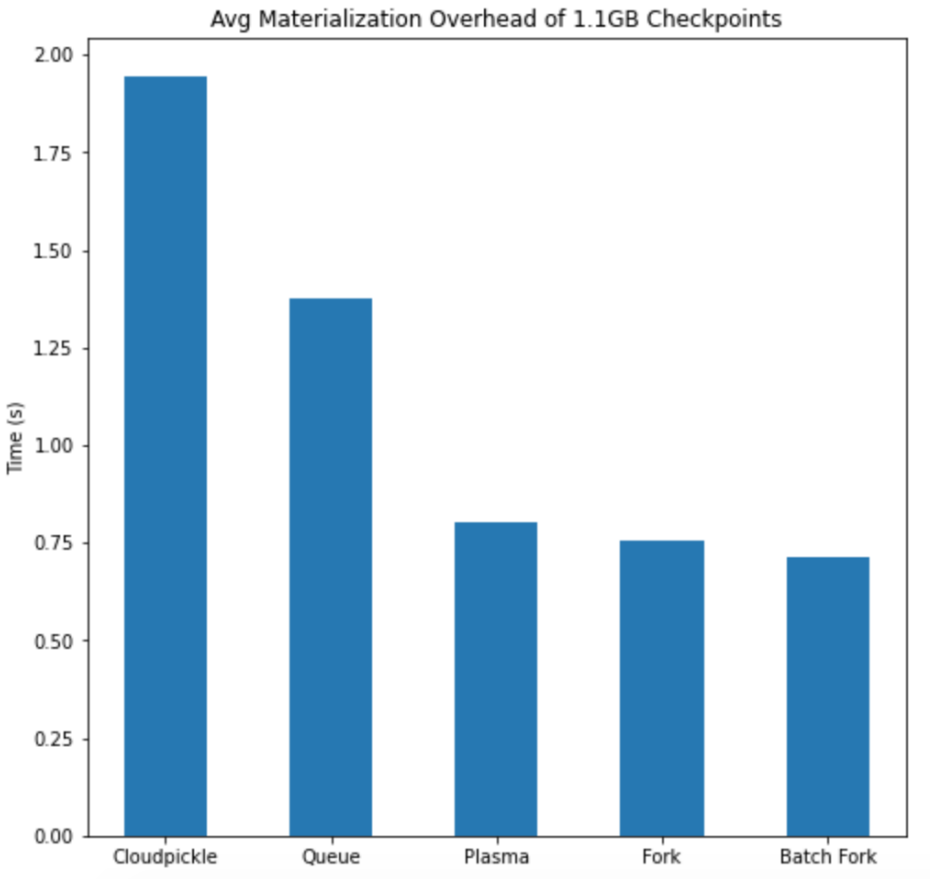


Figure 5.3: Serialization times of different multiprocessing architectures

5.3 Comparison to Alternate Architectures

Next, we compare our selected multiprocessing method, batch fork, and compare it against other potential mechanisms. In this experiment, we take a 1.1GB checkpoint from the RTE experiment, and use different methods to serialize it and write it back to disk. For each mechanism, we repeated this test ten times and reported the average time to serialize and write. Since cloudpickle does serialization and disk writes on the main thread, it has the longest runtime and serves as our baseline. The other methods being compared are all forms of multiprocessing that create workers for serializing and writing checkpoints to disk. Our batch fork experiment was conducted with a buffer of size 10, meaning we would only fork once in this experiment. To measure performance, we time how long the main thread takes to finish executing, ignoring any child processes and letting them run in the background.

As shown in Figure 5.3, cloudpickle clearly performed the worst, with an average serialization time of almost two seconds for one 1.1GB checkpoint. This makes sense, as cloudpickle

blocks the main process from serializing and writing objects to disk, preventing it from making progress. The Multiprocessing Queue fared slightly better, but was in the same range as cloudpickle. The queue performs badly in these scenarios because it still needs to pickle the object before it can be queued up. Although it finishes the main process sooner than cloudpickle, we still spend a significant amount of time performing serialization on the main thread.

The next few methods all fare a little better. Plasma, Fork, and Batch Fork all perform similarly in this experiment, with Batch Fork doing slightly better than the others. For all of these methods, the cost of IPC is small relative to the overall cost, while the cost of copying dominated and accounted for an average of almost 0.7 seconds of the total runtime. Recall that CPU copies of tensors are necessary for any IPC method, and this cost is present in every multiprocessing method we test. Given this number, we can approximate the time it takes for each method to perform their equivalent of IPC. Plasma is exceptionally fast at serializing and takes under 0.2 seconds to store arrays once they have been copied. Fork fares slightly better, as the cost of forking is extremely small on modern processors; we see that the cost to fork to create a child process is well under 0.1 seconds on average. Batch fork performs the best because we don't have to deal with fork overhead on every iteration. Since we are simply caching the copied states most of the time, we only pay for the cost of copying for the first 9 iterations. Only in the tenth iteration do we incur both the cost of copying and the cost of a fork. Interestingly, the cost of forking a batch of states is more expensive than the cost of forking every log statement, but the average cost per serialization is still lower.

While Plasma performs admirably and comes within striking distance of batch fork, it is a more brittle solution than forking. As we have previously mentioned, Plasma performs well on objects like numpy arrays, but performs worse on generic Python objects. We conducted an experiment that compared the costs of Plasma with batch fork on a 1GB list of integers, averaged over 10 iterations. Because the list is a Python object and not an array type object, the serialization and copying costs are substantially more expensive. Plasma takes 15.52 seconds to serialize the object, while batch fork takes 7.8 seconds to copy and fork. Although the runtimes are both higher than the experiment in Figure 5.3, Plasma takes twice as long as batch fork.

Additionally, Plasma's serialization does not support all Python objects and will throw serialization errors if it encounters an unrecognized object. In our experiment, Plasma threw serialization errors because it does not support serialization of Pytorch tensors. To make the experiment work, we had to translate Pytorch tensors to numpy arrays. Although the transformation between tensors and numpy arrays is not difficult or time consuming, it is just one of many unsupported objects that makes Plasma a less attractive choice than forking for IPC.

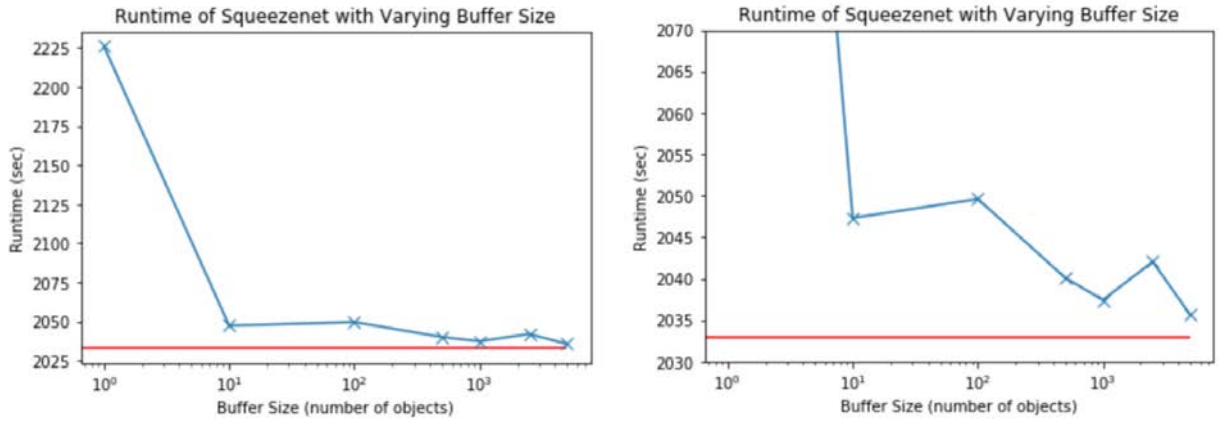


Figure 5.4: Comparison of buffer sizes based on Squeezenet trained on Cifar100 for 200 epochs

5.4 The Effects of Buffer Size

Next we perform a buffer size test in order to determine optimal buffer size and whether or not thrashing can cause problems with our batch fork implementation. In this experiment, we used the Cifr experiment, which trains Squeezenet on the Cifar100 dataset for 200 epochs. Our baseline, in red, reflects the runtime of the Cifr experiment when we don't use Flor. We repeat this experiment seven times, each time with a different buffer size. The buffer sizes we tested were 1, 10, 100, 500, 1000, 2500, and 5000. Since the Cifr experiment stores 3 states per epoch, we accumulate a total of 600 serialized objects on disk, totaling about 760MB of data.

Looking at the left plot of Figure 5.4, we see that forking on every object leads to a tremendously high runtime, while forking fewer times brings down the cost substantially. When the buffer size is 1, the Flor overhead balloons to 2226 seconds. The effects of thrashing are evident in this experiment, since we fork a total of 600 times and cause the CPU to perform needless context switching. This is a rather extreme case of thrashing, and the issue immediately starts to resolve once the buffer size is increased to just 10 objects.

With a size 10 buffer, the effects of thrashing are mitigated and the runtime decreases by over 100 seconds. The right plot of Figure 5.4 shows a zoomed in version of the left plot. As we can see, decreasing the buffer size further helps increase performance, but we quickly reach a point of diminishing returns. When the buffer size reaches 500, our time is just seconds away from the baseline. We can see that the overhead stays roughly the same when we increase the buffer size to 1000 or greater. We can achieve such a low overhead for two reasons. First, since the model size of squeezenet is quite small, the GPU to CPU tensor copy operation is quite fast, taking just milliseconds to complete. Second, our buffer allows us to reduce forking costs by forking very infrequently. In fact, for this experiment,

increasing the buffer size beyond 1000 does not have an effect because Flor would only fork once beyond this point.

Chapter 6

Discussion and Additional Concerns

The multiprocessing materializer was first being developed when Flor was naively recording as much execution state as possible. In these earlier versions, where overheads were as high as 1000× the execution cost, there were substantially more states to log and the serialization time was much higher than actual runtime. In these scenarios, the multiprocessing materializer was able to decrease runtime by over an order of magnitude, but that still wasn't quite enough. As a result, Flor's strategy shifted from a complete capture system to a record-replay system. While the new, lean record strategy Flor has taken on is accountable for much of the overhead reduction, the multiprocessing materializer is still required for good performance. Even though the effectiveness of multiprocessing is blunted when serializing fewer objects, we are still able to see a over 2x runtime improvement compared to cloudpickle in real-world benchmarks, reducing the overhead to below 2% of total runtime. This is especially important for long-running models that can take weeks to train; a percent decrease in overhead can save many hours or even days of additional runtime.

We also considered alternatives to multiprocessing in Python to achieve asynchronous logging, including using Cython or C for logging. These methods were not seriously considered since they would cost us flexibility: we would lose the ability to log arbitrary Python objects. They are, however, effective ways of subverting the GIL. Cython allows users to disable the GIL to increase performance, but this comes with a number of restrictions. The primary restriction of disabling GIL is that the code cannot access Python objects without first converting them to a ctype. Logging at C level is similarly restrictive, as we could parallelize at the cost of avoiding Python objects. To completely subvert the memory locks in C, we would have to modify the Python Interpreter itself, which poses implementation challenges and makes widespread adoption more difficult.

The materializer also has some important drawbacks to keep aware of. Copying objects and storing them in a buffer keeps them alive and consuming RAM. Buffering too many large objects can potentially cause users to run out of memory and crash the program. Mitigating this problem would involve calculating the amount of memory being used and forking when usage reaches a certain level, but doing runtime checks for this can be expensive and could inflate overhead. The alternative is measuring buffer capacity in bytes instead of by number

of objects, which is a more concrete measure of memory usage. Again, however, this would come at an expensive runtime cost, as we would have to estimate the size of every object added to the buffer and keep track of the total cost. Python's native function for getting size is not recursive, and a recursive solution would be the main driver of additional overhead.

In addition to RAM consumption, we note that ignoring SIGCHLD comes with some potential issues. While it is the easiest solution for avoiding the process limit, it can complicate the ML training task. Some ML training tasks already speed up training by performing distributed training, spawning multiple parallel processes to split the work. Ignoring SIGCHLD in the way we do can cause program exceptions because the program no longer receives status updates about their children. A potential solution to this issue is using the signal module to register a custom signal handler that would keep a list of the children processes spawned by Flor. Children not on this list can then be easily monitored and given standard handling.

Lastly, we have realized that forking performance is CPU dependent, and is particularly influenced by clock speed. This, in conjunction with GPU performance, has a major impact on the performance of Flor as a whole. For instance, having a better GPU paired with a slower processor would make it advantageous to simply re-compute values instead of spending the time to serialize them during execution. On the other hand, having a less powerful GPU paired with a higher clocked CPU would make it advantageous to serialize more and recompute less.

Chapter 7

Related Work

User checkpointing is a very common task in ML jobs, but it is often motivated by process saving, not debugging. A summary on checkpointing by the National University of Singapore [6] and a ML checkpointing tutorial for Pytorch, Tensorflow, and Keras [1] both note the importance of checkpointing in long running ML jobs and advocate for checkpointing the best model or at predetermined intervals. The motivation behind best-result checkpointing is to reduce the impact of external events, like hardware failures or OS interrupts. This style of infrequent checkpointing is unable to enable the same fast recovery that Flor checkpointing allows.

There are existing systems that are broadly similar to Flor and are motivated by debugging or analysis [3, 13, 4, 9, 12, 14], but none fill the same hindsight logging role Flor does. These similar systems that can be generally classified as logging/debugging systems or model management systems.

Loggers help gather information about machine learning models, whether they may be models themselves or metrics. Some notable loggers include Tensorboard and Sacred [3, 4]. Tensorboard is a very useful tool for analytics, but has limited utility in debugging because it requires additional training runs to log information. Sacred is a similar system that stores both metrics and sources of randomness in ML code, but similarly to tensorboard, has limited usefulness in debugging. Recently, there has also been an introduction to better debugging tools, notably Amazon's Sagemaker debugger [9]. This tool, however, simply forks the ML process and runs it in debug mode separately. Later, if you want to debug something in your model, you can inspect the trace or rerun the code in debug mode. This is a rather clumsy and resource intensive solution that still requires full re-executions for more information.

Other broadly related systems occupy the model management role, and target some of the stages in the ML lifecycle. ModelDB is a system that works to capture context from pipeline development, helping users track and analyze the models that they have developed [12]. Other similar systems intend to streamline model development and include MLFlow and Wandb [14, 13]. Both MLFlow and Wandb occupy broader roles than Flor does, each containing multiple components that can be utilized independently. Both systems include a tracking component that is able to log and track metrics or parameters. Unfortunately,

the same issues that applied to other loggers apply to MLFlow and Wandb: logging requires foresight and limits its usefulness for debugging.

Chapter 8

Future Work

One of the current issues with Flor is that it currently only supports GPU re-execution. As such, the amount of parallelism we can achieve is limited by the number of GPUs available in the system. However, GPUs are not the only resource available to us in systems, as unused CPU cores can potentially be put to work to add speed-ups to replay. Not only will this allow us to get even more speed-ups out of GPU machines, but it allows Flor to be able to perform some degree of replay on systems that aren't equipped with GPUs.

Additionally, we can begin developing other systems that complement Flor's capabilities and work to provide better data context. Ground Flor is a work-in-progress system meant to collect data context from the data cleaning and model training processes with the help of some Flor mechanics. It also takes advantage of Ground [5], a data context service, to help store and service our captured data context.

8.1 Ground Flor

Ground Flor is a system intended to bridge the gap between Flor and Ground by providing post-hoc data provenance ingestion from model training. Ground's internal metamodel consists of a layered graph structure that supports storage of application context, behavioral context, and change context, making it a powerful tool for providing a high level of abstraction for data context. In order to provide the information needed for Ground's model, Ground Flor will work in a record-replay fashion that takes advantage of both static and dynamic analysis. It can populate Ground at runtime with information gathered via static analysis, and return richer context with dynamic analysis during the replay phase. Combining Flor mechanics with the Ground model allows us to support more complex queries about data change including data change over time, data drift between projects, and queries over sets of experiments.

Model training bugs, particularly ones that stem from the training data itself, are difficult to manage because they may require the user to aggregate or compare information before finding a solution. For example, a data formatting error in a table of training data is difficult

to spot just by inspection, but it becomes more obvious after comparing it to previous, correctly formatted versions of the table. Other problems, like identifying what models are affected by an upstream data change, can compel model developers to repeatedly look through their training code for answers. In these situations, developers essentially meander through code and data, guessing and checking their way to find answers and bugs. This method is both time consuming and error-prone. Being able to make relevant data queries, such as via Ground, would help drastically decrease the amount of time spent gathering and comparing data for debugging.

To this end, we have isolated three questions focused on that are necessary to support data provenance queries:

1. What rows or columns are derived from other rows and columns in the dataframe?
2. What rows or columns have changed and how have they changed?
3. What rows and columns were ultimately used to train the model?

These are relatively simple questions representing the minimal amount of information that needs to be captured to be useful. Even though these questions are simple, they can be difficult to answer without instrumentation or domain knowledge. Vamsa is a similar system that provides coarse-grained data provenance tracking on data science scripts [7], but it relies on human assistance for additional domain knowledge. Vamsa is able to identify functions, but not what a function does, so it gets additional information from a curated set of function descriptions called the knowledge base.

We envision Ground Flor utilizing a combination of static and dynamic analysis to provide better behavioral context and answer the three questions presented above. During runtime, Flor will use static analysis to build an underlying workflow graph, containing the relationships and lineage between variables. However, static analysis is not able to recognize side effects of functions or changes applied to data. If the user's query requires richer context, like specific changes to data, Ground Flor will perform re-execution with dynamic analysis to update the workflow graph. Dynamic analysis can be achieved by using Flor checkpointing to capture object states before and after each operation. These changes can be categorized and used to describe the effects of unknown functions, eliminating the need for additional domain knowledge.

By combining static and dynamic analysis, we are able to achieve an approach that has the strengths of both processes. Information on data relationships is collected by static analysis during runtime, providing some information at a low initial cost. If more information is needed, dynamic information can provide additional insight on data change and lineage. The workflow graphs that are generated are able to capture the lineage of data, how it changes, and what parts of it are used by model training. To start, we plan to keep our scope narrow and we only address data in pandas dataframes.

The workflow graph that Ground Flor generates is an intermediate representation of our captured data context, and can be used as a visual aid for users. To be compatible

with Ground, we will need to translate these representations into ones that are compatible with the Ground metamodel. Once this is done, however, we can fully take advantage of Ground's services and allow users to perform more complex queries, like queries over aggregates. Ultimately, we are seeking to populate Ground by using Ground Flor, enabling us to take advantage of Ground's high level semantics for data context management.

Chapter 9

Conclusion

Machine learning is a costly and time intensive task, taking up to hundreds or thousands of hours to train one model. Additionally, bugs in ML are difficult to detect and no effective method exists at speeding up post-hoc analysis. In this paper, we have introduced Flor, a system for record and replay of ML training that decreases the time needed to perform analysis and debug. Flor allows model developers to quickly analyze their ML code by facilitating fast replay of previous training runs, with minor checkpointing overhead on the first run. It is able to achieve this low checkpointing overhead with the multiprocessing materializer, a solution that uses forking to both create worker processes and support one-way interprocess communication.

Without using the multiprocessing materializer, we see Flor achieving a 4.76% average overhead on record. Adding in the multiprocessing materializer yields a 73.5% reduction in overhead, leading to an overall cost of just 1.74%. Having an overhead this low makes it almost unnoticeable to the user, while still being able to supply the time-saving replay benefits later down the line.

Bibliography

[1] *Checkpointing Tutorial for TensorFlow, Keras, and PyTorch*. Nov. 2017. URL: https://blog.floydhub.com/checkpointing-tutorial-for-tensorflow-keras-and-pytorch/.

[2] Andrew McCallum Emma Strubell Ananya Ganesh. "Energy and Policy Considerations for Deep Learning in NLP". In: *arXiv preprint arXiv:1906.02243* (2019).

[3] Google. *TensorBoard*. tensorflow.org/tensorboard.

[4] Klaus Greff et al. "The Sacred Infrastructure for Computational Research". In: *Proceedings of the 16th Python in Science Conference*. Ed. by Katy Huff et al. 2017, pp. 49–56. DOI: 10.25080/shinma-7f4c6e7-008.

[5] Joseph M Hellerstein et al. "Ground: A Data Context Service." In: *CIDR*. 2017.

[6] Ku Wee Kiat. *Deep Learning Best Practices: Checkpointing your Deep Learning Model Training*. Dec. 2018. URL: https://nusit.nus.edu.sg/services/hpc-newsletter/deep-learning-best-practices-checkpointing-deep-learning-model-training/.

[7] Mohammad Hossein Namaki et al. "Vamsa: Tracking Provenance in Data Science Scripts". In: *ArXiv* abs/2001.01861 (2020).

[8] Robert Nishihara and Philipp Moritz. *Fast Python Serialization with Ray and Apache Arrow*. Oct. 2017. URL: https://arrow.apache.org/blog/2017/10/15/fast-python-serialization-with-ray-and-arrow/.

[9] Julien Simon. *Blogs*. Dec. 2019. URL: https://aws.amazon.com/blogs/aws/amazon-sagemaker-debugger-debug-your-machine-learning-models/.

[10] Leslie N Smith and Nicholay Topin. "Super-convergence: Very fast training of neural networks using large learning rates". In: *Artificial Intelligence and Machine Learning for Multi-Domain Operations Applications*. Vol. 11006. International Society for Optics and Photonics. 2019, p. 1100612.

[11] Ion Stoica et al. *A Berkeley View of Systems Challenges for AI*. Tech. rep. UCB/EECS-2017-159. EECS Department, University of California, Berkeley, Oct. 2017. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-159.html.

[12] Manasi Vartak et al. “ModelDB: a system for machine learning model management”. In: *Proceedings of the Workshop on Human-In-the-Loop Data Analytics*. ACM. 2016, p. 14.

[13] *Weights&Biases*. `wandb.com`.

[14] Matei Zaharia et al. “Accelerating the Machine Learning Lifecycle with MLflow”. In: *Data Engineering* (2018), p. 39.