# Understanding Silent Data Corruption in LLM Training

**Jeffrey Ma**[1] **Hengzhi Pei**[2] **Leonard Lausen**[2] **George Karypis**[2]
[1]Department of Computer Science, Harvard University [2]Amazon Web Services
jeffreyma@g.harvard.edu
{philepei,lausen,gkarypis}@amazon.com

## Abstract

As the scale of training large language models (LLMs) increases, one emergent failure is silent data corruption (SDC), where hardware produces incorrect computations without explicit failure signals. In this work, we are the first to investigate the impact of real-world SDCs on LLM training by comparing model training between healthy production nodes and unhealthy nodes exhibiting SDCs. With the help from a cloud computing platform, we access the unhealthy nodes that were swept out from production by automated fleet management. Using deterministic execution via XLA compiler and our proposed synchronization mechanisms, we isolate and analyze the impact of SDC errors on these nodes at three levels: at each submodule computation, at a single optimizer step, and at a training period. Our results reveal that the impact of SDCs on computation varies on different unhealthy nodes. Although in most cases the perturbations from SDCs on submodule computation and gradients are relatively small, SDCs can lead models to converge to different optima with different weights and even cause spikes in the training loss. Our analysis sheds light on further understanding and mitigating the impact of SDCs.

## 1 Introduction

Large language models (LLMs) have demonstrated remarkable advancements in different tasks. To further explore the potentials of LLMs, recent efforts have been put into scaling up the model size to hundred-billions or even trillions of parameters. As a result, training such large models requires extensive computational resources over a long training period. For example, Llama 3 405B was trained on 16K H100 GPUs (Llama Team, 2024).

However, as training scale increases, the likelihood of hardware failures also increases. Silent data corruption (SDC) is an emerging error that causes impacted hardware to inadvertently output wrong calculation results silently without any user indication (Dixit et al., 2021). Meta reported 6 unplanned job interruptions were attributed to SDC during a 54-day pre-training snapshot (Llama Team, 2024) and Google estimated an SDC event occurs every week or two during Gemini training (Gemini Team, 2024). In practice, SDCs observed during large-scale training usually result from latent hardware defects that cause corruption only under certain conditions or after sufficient stress over the hardware's lifetime. Once a machine begins to be affected by SDCs, it pollutes training outputs and can impact the model optimization trajectory (He and Li, 2023). Although many works studied the effect of SDCs in large-scale CPU systems (Dixit et al., 2021; Wang et al., 2023a), autonomous systems (Wan et al., 2022; Hsiao et al., 2024) and deep learning accelerators (Zhang et al., 2018; Li et al., 2017; Rech and Rech, 2022), no public work has characterized the impact of real-world SDCs on LLM training in detail.

In this work, we are the first to investigate the impact of real-world SDCs on LLM training. We work with a cloud-computing platform to gain access to unhealthy nodes that failed production fleet management testing due to SDCs. While unhealthy hardware is generally excluded from production workloads, latent defects and hardware failures can turn previously healthy hardware unhealthy, emphasizing the importance of our investigation.

By leveraging deterministic execution from the XLA compiler and adopting the same training setup, we can compare the results from unhealthy nodes and healthy nodes to characterize the impact of SDCs during training. We break down our investigation into three levels: (1) the impact on submodule computation; (2) the impact on the gradients of model weights at a single optimizer step; and (3) the impact on the model quality over a training period. Since SDC error can accumulate, we isolate the impact of SDCs at different levels by overwriting the intermediate results computed on the unhealthy node with results from the healthy node. Specifically, we design a computation synchronization mechanism to ensure the input of every submodule is the same on healthy nodes and on unhealthy nodes for (1), and a parameter synchronization mechanism to ensure the model weights are the same before each optimizer step for (2).



*Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 20372–20394

We conduct quantitative comparisons with the computations on healthy node at different levels. To investigate the impact of SDCs on submodule computation, we check the forward and backward computation of the self-attention and feed-forward network (FFN) across unhealthy nodes. To investigate the impact at each optimizer step, we examine the difference in gradients. To investigate the accumulated impact on the model quality, we track the loss and parameter difference during pretraining and also examine the finetuning performance on unhealthy nodes.

Our empirical results show that SDCs do not occur uniformly during training and exhibit different patterns on different unhealthy nodes. We find that SDCs can cause certain values in the submodule computation results to differ by large factors, while the average mismatch frequency is generally low. Furthermore, the noise to gradients caused by SDC error within an optimizer step is small relative to the true gradient norm. For the accumulated impact over training steps, although the pretraining loss remains similar, SDCs can cause model parameters to drift away from ground-truth weights, which indicates that models on different nodes converge to different optima. Meanwhile, although the models fine-tuned on most unhealthy nodes have similar performance compared to the models fine-tuned on the healthy node, loss spikes do occur during fine-tuning on some unhealthy nodes, which can fully corrupt the model weights in some cases.

In summary, our contributions are:

- We are the first to investigate the impact of real-world SDCs on LLM training in detail by obtaining access to realistic unhealthy nodes flagged by the production fleet management.
- By pairing unhealthy nodes with healthy nodes and introducing synchronization mechanisms, we design experiments to precisely isolate the impact of SDCs at different levels.
- We reveal the characteristics of SDCs at various levels of model training empirically and further provide insightful analysis which sheds light on the future work on understanding and mitigating the impact of SDCs.

## 2 Background

### 2.1 Silent Data Corruption Errors

Silent Data Corruption (SDC) errors are incorrect computation that silently occur during normal usage (Papadimitriou et al., 2023). Generally, SDCs can arise from hardware faults (Dixit et al., 2021; Hochschild et al., 2021), environmental factors like radiation (Ziegler, 1996; Mukherjee et al., 2005; Baumann, 2005), or software bugs (Lou et al., 2022). With growth of large scale distributed systems, SDC is observed to be caused by device-specific hardware defects which show errors at certain utilization levels or temperatures (Dixit et al., 2021; Hochschild et al., 2021; Wang et al., 2023a).

Our work lies in the broader area of understanding the effect of SDCs on deep learning and we leave a detailed discussion for the related work in Appendix A. There are two limitations in this area. First, fault-injection methods are commonly used for evaluation. Although SDC can be simulated at the hardware level (Rech and Rech, 2022; Li et al., 2017) or the software level (He et al., 2020; Agarwal et al., 2022), simulated SDCs could be different from those observed in production. Second, most works study the impact of SDCs on model inference (Li et al., 2017; Agarwal et al., 2023; Ma et al., 2023) while few study the impact of SDCs on model training dynamics. Although some empirical findings of SDC like NaN (Not-a-Number) issues and degradation during training are reported (Elsen et al., 2023; He et al., 2023a), there is no work that attempts to characterize and break down the impacts of real-world SDCs especially on large language model training.

### 2.2 Large Language Model Training

In this work, we consider a Large Language Model (LLM) as a transformer decoder (Vaswani et al., 2017). Training an LLM at scale requires a combination of data parallelism and model parallelism. Tensor parallelism (TP) is a widely used model parallelism approach that partitions each individual layer of a model across accelerators (Shoeybi et al., 2020). For large-scale LLM training, tensor-parallel size is usually set as the number of accelerators within a node to leverage high bandwidth intra-node communication (Narayanan et al., 2021).

Given that hardware failure is usually flagged at the node level and majority of training failures are caused by one node (Wang et al., 2023b), we focus on the impact of SDCs on the computation of a single node and use tensor parallelism only.

## 3 Methodology

In this section, we discuss our methodology for investigating the effect of real-world SDCs on LLM

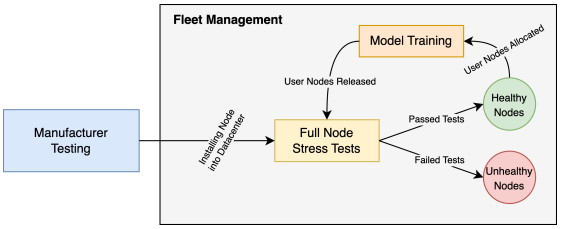



Figure 1: Illustration of fleet management flow, where nodes are vetted through several rounds of testing at different granularities.

training. We first describe a high-level mechanism used to collect the unhealthy nodes for our experiments. Then, we break down the investigation of the SDC impact into three granularities and ask three key research questions (RQs). To answer these RQs, we further propose two synchronization mechanisms to isolate the impact of SDC.

### 3.1 Hardware Collection in the Real World

To study the effect of real-world SDCs, we identify *nodes that failed production margin stress tests and thus were not allowed into production availability*. Figure 1 shows a high-level production fleet management flow for identifying failing nodes. Before entering this flow, components such as machine learning accelerators go through stages of testing at the manufacturer and after system assembly.

Once a node is installed in a datacenter, additional stress tests are triggered to identify any compound hardware issue. These tests include full-node communication collective stress tests, compute unit stress tests, and a small LLM training run where training outputs are compared with precomputed golden truth values. With deterministic workload execution, the difference from ground truth values or other non-determinism indicates SDC on the node. To guard against hardware degradation over time, tests are also run when a node is reclaimed from customer usage either due to customer workloads ending without any indication of error or because the customer returned the hardware after receiving a signal indicating a hardware health issue from the cloud provider.

Using this flow, we define two types of nodes:

- *Unhealthy nodes* are nodes that fail the fleet management tests due to exhibiting SDC.
- *Healthy nodes* are nodes from production that have passed the aforementioned tests.

Each category contains fifteen (15) nodes. We have confirmed that *all healthy nodes in our experiments will output the same result for the same computation* and we denote them as the healthy node for simplicity. However, unhealthy nodes can exhibit different symptoms for SDC and we assign each unhealthy node a unique identifier, namely Node 1 to Node 15.

Unhealthy nodes flagged by fleet management are meaningful for studying real-world SDCs. Given that fleet management can only be run when the nodes are not used by customers, during multi-month periods of a large-scale LLM training run, healthy nodes that were originally healthy can degrade and produce SDCs affecting training before fleet management can isolate them. We confirm that some of the unhealthy nodes in our experiments were initially healthy and began to fail the pre-checks after being used in training.

### 3.2 Key Research Questions

To better understand the impact of SDCs, we break down our investigation into three levels and ask three main research questions (RQs):

**RQ1**: *What is the impact of SDCs on Transformer submodule computation results?*

**RQ2**: *What is the impact of SDCs on the gradients of model weights at a single optimizer step?*

**RQ3**: *What is the accumulated impact of SDCs on the model quality over training steps?*

Investigating **RQ1** helps us understand the frequency and severity of SDCs, critical for designing a solution to detect SDC at the submodule level. Investigating **RQ2** and **RQ3** gives us more insight into understanding optimization dynamics when SDCs occur. Most importantly, these research questions help prioritize future directions of detecting, mitigating, and recovering from SDC by providing concrete real-world SDC characteristics.

To compare SDC-induced incorrect computations with ground truth, we pair each unhealthy node with a healthy node and train identical models simultaneously on each node with exactly the same training setup. We employ the XLA compiler (Sabne, 2020) to ensure fully deterministic instruction ordering to isolate away non-SDC sources of non-determinism like floating point error. In other words, we confirm that *the computation results are exactly the same on any two healthy nodes with the same compiler and the difference in computation results can be entirely attributed to SDC.*

Since SDC error can accumulate over computation, we need to correct the error on unhealthy nodes with the ground-truth results on the healthy node to isolate the impact of SDCs at different levels. Specifically, we design two investigative

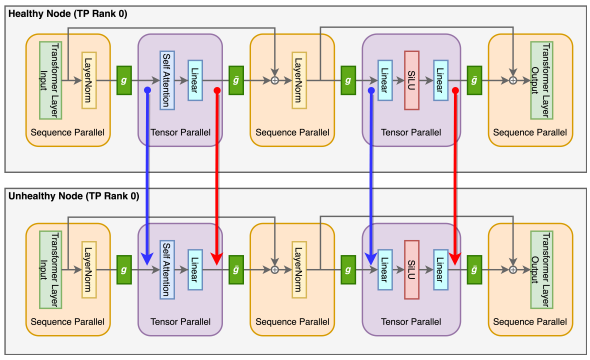



Figure 2: Illustration of our lock-step parallelism works in a Transformer decoder layer. The arrows indicate intermediate tensors corrected by communicating values from the healthy to the unhealthy node (red in forwards pass, blue in backwards pass). In forward pass, $g$ is an all-gather and $\bar{g}$ is a reduce-scatter, while in the backwards pass $g$ is an reduce-scatter and $\bar{g}$ is an all-gather.

synchronization mechanisms for **RQ1** and **RQ2**.

**Computation Synchronization.** To isolate the impact of SDCs at submodule level (**RQ1**), we set up a novel communication mesh called **lock-step parallelism** shown in Figure 2. For each pair of same TP ranks on the unhealthy and healthy node, in the forward pass, we communicate and compare the output values after the forward computation of each submodule (self-attention or FFN) before the reduce-scatter of sequence parallelism, which avoids incorrect values on certain ranks spreading to tensor shards on other ranks after communication. Then, we overwrite the values on the unhealthy node with those from the healthy node (red arrows) to prevent SDC error from accumulating to the next submodule. Likewise, in the backward pass, we compare the input gradient after the backward computation of each submodule before the reduce-scatter (blue arrow) and overwrite the gradient values to prevent error from accumulating through backpropagation. More details can be found in Appendix C.

**Parameter Synchronization.** To isolate the impact of SDCs at a single training step (**RQ2**), we do parameter synchronization at the end of each training step. After taking an optimizer step, we broadcast the updated model parameters from the healthy node to overwrite the parameters on the unhealthy node. In this way, both nodes start from the same parameters for the next optimizer step.

For our synchronization mechanisms, we assume that *SDCs do not occur when communicating tensors between the unhealthy and healthy node*. First, all nodes used in our experiments consistently passed stress tests for communication collectives. Second, the communication primitives used in our synchronization mechanism do not involve any compute unit. Finally, parity checks or error correcting codes are utilized for communication across the network. To avoid SDCs occurring during the arithmetic of tensor comparisons between healthy and unhealthy nodes, we always compute the comparisons on the healthy node.

## 4 Experiments for RQ1

**RQ1**: *What is the impact of SDCs on Transformer submodule computation outputs?*

In this section, we first introduce our experiment setups for RQ1 and analyze experimental results to understand the impact of SDCs on submodule computation. We follow the same structure for RQ2 in Section 5 and RQ3 in Section 6.

### 4.1 Experiment Setups

To provide a better understanding of SDC impact at Transformer submodule granularity, we design experiments to understand the frequency and severity of SDC impacts on intermediate tensors and check whether contemporary methods like algorithm-based fault tolerance (ABFT) can detect these errors.

**Experiment I**: *How prevalent and severe are SDC occurrences on Transformer submodule outputs?* We focus on four kinds of Transformer submodule computation, namely the forward computation of a self-attention module (FWD/ATTN) and an FFN module (FWD/FFN), and the backward computation of a self-attention module (BWD/ATTN) and an FFN module (BWD/FFN).

We train two models on each pair of the unhealthy node and the healthy node simultaneously and use the **computation synchronization** mechanism discussed in Section 3.2. We use a decoder-only Transformer architecture similar to the Llama3-8B configuration (Llama Team, 2024) with $D = 16$ decoder layers and hidden state size of $H = 4096$ and use the tensor parallelism to fit a model within a node. More details can be found in Appendix C.

For a submodule computation $f$ in the model, we define $f'_i(x_{t,j})$ as the tensor computed on TP rank $t$ of unhealthy node $i$ at the microstep $j$ and $f(x_{t,j})$ as the corresponding output on healthy node. To quantify differences between $f'_i(x_{t,j})$ and $f(x_{t,j})$, we define two metrics called *mismatch frequency* and *mismatch severity*. We calculate the mismatch

frequency for submodule $f$ on unhealthy node $i$ at the microstep $j$ as follows:

$$freq_{i,j}^{f} = \frac{\sum_{t=1}^{TP} Mis(f_i'(x_{t,j}), f(x_{t,j}))}{TP \cdot MBS \cdot L \cdot H} \quad (1)$$

where $Mis(y', y)$ counts the number of mismatching elements in two tensors $y$ and $y'$. We report the aggregated mismatch frequency for each submodule computation type $F$ on unhealthy node $i$ at the microstep $j$ by averaging across decoder layers:

$$freq_{i,j}^{F} = \frac{1}{D} \sum_{f \in F} freq_{i,j}^{f}, F = \{f^{(1)}, ..., f^{(D)}\} \quad (2)$$

Mismatch severity is defined as the average over non-zero values of the element-wise relative difference. Formally, we take the maximum over all TP ranks and calculate the mismatch severity for submodule $f$ on unhealthy node $i$ at microstep $j$:

$$sev_{i,j}^{f} = \max_{0 \le t < TP} \left[ Avg_{\neq 0} \left( \left| \frac{f_i'(x_{t,j}) - f(x_{t,j})}{f(x_{t,j})} \right| \right) \right] \quad (3)$$

where $Avg_{\neq 0}(x)$ computes the average value over only non-zero elements of a tensor $x$. We also calculate the mismatch severity for each type of submodule computation $F$ on unhealthy node $i$ at the microstep $j$ by taking the maximum across decoder layers as follows:

$$sev_{i,j}^{F} = \max_{f \in F} sev_{i,j}^{f}, F = \{f^{(1)}, ..., f^{(D)}\} \quad (4)$$

**Experiment II**: *Can algorithm-based fault tolerance (ABFT) detect the errors in Transformer submodule outputs?* Algorithm-based fault tolerance (ABFT) is one common scheme to detect SDC errors during deep learning computations (Zhao et al., 2021; Xue et al., 2023). Specifically, Smith and van de Geijn (2015) propose adding a checksum to matrix multiplication to flag if SDCs have occurred during computation, comparing the results of two data paths, the matrix multiplication result and the checksum element. For a floating-point matrices $A \in \mathbb{R}^{m \times k}, B \in \mathbb{R}^{k \times n}, C = AB$ and a column vector of ones $w$, ABFT reports an SDC if:

$$\|Cw - A(Bw)\|_\infty > \tau \|A\|_\infty \|B\|_\infty \quad (5)$$

where $\tau = ku$ and $u$ is the unit-roundoff determined by the floating-point precision used in the matrix multiplication.

In this experiment, we add ABFT into the computation of all linear layers in the forward and backwards pass of model training. For each kind of matrix multiplication, we record the frequency of ABFT-flags across TP ranks and layers using the condition in 5. We train the Transformer model with $D = 8$ decoder layers under the precision of `float32`, with rest of hyperparameters the same as detailed in Appendix B. Note that underlying theoretical assumptions mean that this ABFT method cannot be applied in lower precision data types like `bf16`. More discussion and details can be found in Appendix Section D.2.

| NODE ID | FWD/ATTN | FWD/FFN | BWD/ATTN | BWD/FFN |
|---|---|---|---|---|
| NODE 1 | 1.55e-5 | 5.06e-7 | 1.56e-04 | 2.81e-6 |
| NODE 4 | 3.79e-9 | 9.20e-11 | 2.99e-9 | 2.80e-11 |
| NODE 5 | 0 | 0 | 1.49e-15 | 1.25e-12 |
| NODE 6 | 1.71e-9 | 1.64e-11 | 1.49e-9 | 6.02e-11 |
| NODE 7 | 2.13e-6 | 1.18e-7 | 4.31e-6 | 6.73e-8 |
| NODE 8 | 3.21e-9 | 1.99e-11 | 1.01e-7 | 2.21e-9 |
| NODE 9 | 1.10e-5 | 5.05e-7 | 4.33e-6 | 3.86e-8 |
| NODE 10 | 4.78e-3 | 1.03e-3 | 1.92e-3 | 7.98e-5 |
| NODE 11 | 2.89e-2 | 2.25e-3 | 6.71e-3 | 1.08e-4 |
| NODE 13 | 0 | 0 | 0 | 1.21e-10 |
| NODE 14 | 6.48e-11 | 0 | 4.91e-10 | 2.99e-9 |
| NODE 15 | 0 | 0 | 0 | 7.39e-15 |

Table 1: Average mismatch frequency over microsteps for Transformer submodules. The table excludes the nodes that do not show any SDC in this setting.

### 4.2 Results

**Results for Experiment I.** Table 1 shows the mismatch frequency of submodule computation. We observe that the impact of SDCs on submodule computation varies across different unhealthy nodes, e.g. Nodes 10 and 11 have a high mismatch frequency while Nodes 2 and 3 do not show any SDC occurrence in this setting.

We further find that SDCs do not occur uniformly over time: mismatch frequency often has a large variance across steps. Figure 3 shows the mismatch frequency in the forward computation of the attention module on Nodes 7 and 14. We find that spikes of mismatch frequency sometimes occur, while during the majority of training time, no mismatch occurs. In Figure 4, we observe a high peak of mismatch frequency at the first few steps on Nodes 10 and 11, likely due to higher overall system usage when initializing the training run. The non-uniform occurrence of SDC during training suggests that SDCs might be caused by broader, compound system-level factors.

Table 2 shows the maximum mismatch severity over optimizer steps for submodule computation on different unhealthy nodes. We find that SDCs

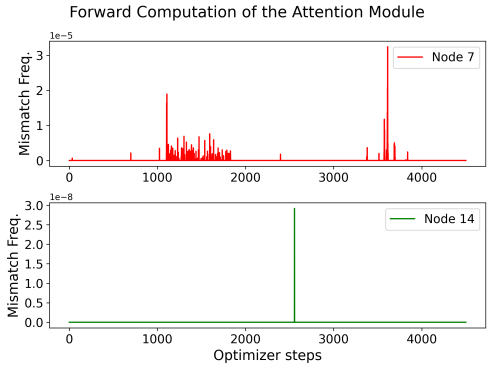



Figure 3: Non-uniform spikes of mismatch frequency in the forward computation of the attention module over time on Node 7, 14.

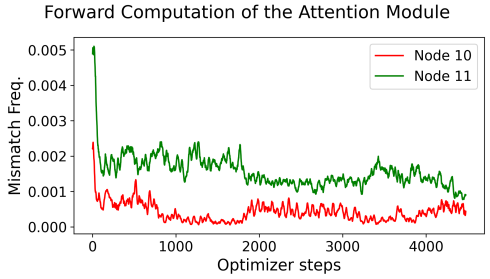



Figure 4: High SDC occurrence with large initial spikes in smoothed mismatch frequency for the forward computation of the attention module on Node 10, 11.

cause certain tensor values in the computation to differ by large factors. For example, on Node 9, the mismatch severity exceeds 100, which means SDCs can cause degraded TP ranks to have very different computation results on certain microsteps.

**Results for Experiment II.** We find ABFT fails to flag any error on most of the unhealthy nodes, except Node 14. In Figure 5, we observe ABFT only frequently flags SDCs on Node 14 while flagging little to no errors on other unhealthy nodes. We also note that the rates of SDCs flagged in this `float32` ABFT setting do not necessarily correspond to submodule output mismatches at `bfloat16` induced by SDCs in the Section 4.2 results. As from Figure 3, SDCs on Node 14 occur as a singular spike of mismatches at `bfloat16` as observed in Section 4.2, which conflicts with how often ABFT flags SDCs on the same node at `float32`. We hypothesize some possible explanations for this phenomenon in Appendix D.

## 5 Experiments for RQ2

**RQ2**: *What is the impact of SDCs on the gradients of model weights at a single optimizer step?*

| NODE ID | FWD/ATTN | FWD/FFN | BWD/ATTN | BWD/FFN |
|---|---|---|---|---|
| NODE 1 | 99 | 312 | 7392 | 3.88 |
| NODE 4 | 2.95 | 1.46 | 2.39 | 5.38 |
| NODE 5 | 0 | 0 | 0.0047 | 0.0908 |
| NODE 6 | 1.01 | 1 | 0.162 | 0.0776 |
| NODE 7 | 43.2 | 88.5 | 32.3 | 0.297 |
| NODE 8 | 13.5 | 5.69 | 6.41 | 1.59 |
| NODE 9 | 119 | 200 | $3.79e$+11 | $9.63e$+12 |
| NODE 10 | 1120 | 262 | 12.75 | 0.648 |
| NODE 11 | 318 | 976 | 2208 | 7680 |
| NODE 13 | 0 | 0 | 0 | 0.316 |
| NODE 14 | 1.12 | 0 | 3.80 | 0.380 |
| NODE 15 | 0 | 0 | 0 | 0.0176 |

Table 2: Maximum mismatch severity over microsteps for each unhealthy node. The table excludes the nodes that do not show any SDC in this setting.

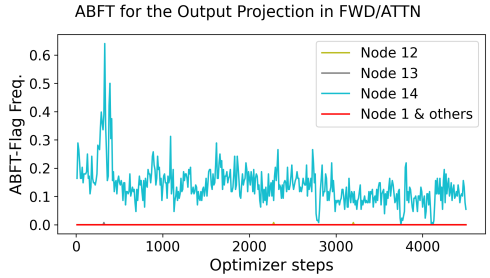



Figure 5: Frequency of ABFT flaging an SDC in the forward computation of the output projection in the attention module during training.

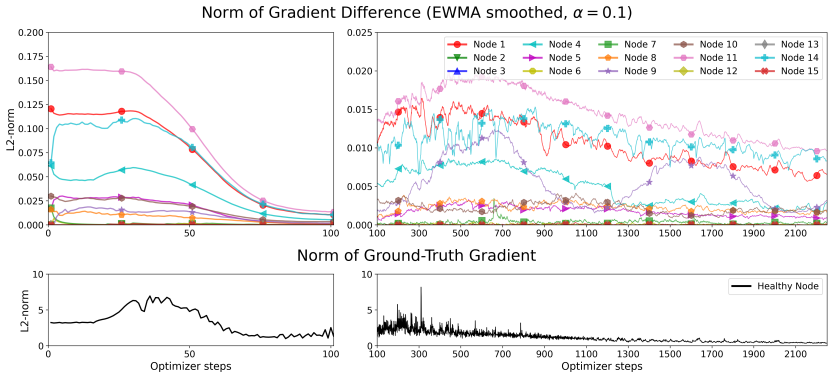
### 5.1 Experiment Setups

**Experiment III**: *How different are the gradients due to SDC error during model pre-training?*

We train same models on each pair of unhealthy and healthy nodes simultaneously and use the **parameter synchronization** mechanism discussed in Section 3.2. After the forward and backward pass are finished at step $j$, we compute the $L_2$ norm of elementwise difference between the gradients of model weights on the $i$-th unhealthy node $g'_{i,j}$ and the ground-truth gradients on the healthy node $g_j$. After taking an optimizer step, we use parameter synchronization to overwrite the model parameters on the unhealthy node. We also report the *worst case noise-to-signal (WCNTS) ratio* to measure how significant SDC-induced noise to gradients is:

$$WCNTS_i = \max_j \frac{\|g'_{i,j} - g_j\|_2}{\|g_j\|_2} \quad (6)$$

Using the same decoder block architecture as in Section 4, we train a 32-layer Transformer decoder with hidden state size as $H = 4096$. More details can be found in Appendix B.




| Node ID | WCNTS Ratio |
|---|---|
| 1 | 0.037 |
| 2 | 0.002 |
| 3 | 1.34$e$-15 |
| 4 | 0.019 |
| 5 | 0.011 |
| 6 | 0.004 |
| 7 | 0.008 |
| 8 | 0.006 |
| 9 | 0.017 |
| 10 | 0.010 |
| 11 | 0.051 |
| 12 | 0.002 |
| 13 | 0.004 |
| 14 | 0.037 |
| 15 | 4.16$e$-4 |

Figure 6: $L_2$-norm of the gradient difference and the ground-truth gradients over steps. The left table shows Worst Case Noise-to-Signal (WCNTS) ratios for unhealthy nodes.

### 5.2 Results

**Results for Experiment III.** Figure 6 shows the $L_2$ norm of gradient difference and the WCNTS ratio for the gradients over optimizer steps across unhealthy nodes. We observe that gradients computed on unhealthy nodes deviate minimally from those computed on the healthy node. Although the absolute value of $L_2$ norm of the gradient difference is large before the 100-th step, it is still relatively small compared to the $L_2$ norm of the ground-truth gradients and continues to decrease as the norm of the ground-truth gradients decreases. In the worst case on Node 11, the $L_2$ norm of gradient difference is $5.1\%$ of that of the ground-truth gradients, showing that the SDC-induced noise in the gradients is relatively small relative to the ground-truth gradients.

## 6 Experiments for RQ3

**RQ3**: *What is the accumulated impact of SDCs on the model quality over multiple training steps?*

### 6.1 Experiment Setups

To provide a better understanding of how different the learned representations and model decision boundaries from unhealthy nodes would be, we design experiments for both model pre-training from scratch and fine-tuning of a pre-trained model.

**Experiment IV**: *How different is the learned model under accumulated SDC error from the ground truth during model pre-training?* We pre-train same models on each pair of unhealthy node and healthy node simultaneously. We follow the same experiment setting in Section 5.1 except we *do not use parameter synchronization mechanism*. To observe how SDCs impact the learned models during training, we report the training loss and the *parameter difference* which is $L_2$ norm of the element-wise difference between model parameters on healthy and unhealthy node.

**Experiment V**: *For downstream tasks, how would SDCs affect model finetuning?* We want to further understand how model quality is affected by SDCs when the model is fine-tuned on a downstream task. In this experiment, we fine-tune `Mistral-7B-v0.3` (Jiang et al., 2023) on six multiple-choice question answering tasks (CosmosQA (Huang et al., 2019); MathQA (Amini et al., 2019); ScienceQA (Lu et al., 2022); OpenbookQA (Mihaylov et al., 2018); BoolQ (Clark et al., 2019); and RACE (Lai et al., 2017)) by instruction tuning (Wei et al., 2022). For each task, we use a fixed random seed for shuffling the training dataset and evaluate the test accuracy (TA) of the models fine-tuned on the healthy node and on unhealthy nodes, isolating away any variability due to randomness in data ordering or computation ordering. Using the predictions of the model fine-tuned on the healthy node as the standard, we report the disagreement percentage (DP), which is defined as the percentage of the difference in predictions on the test set. Furthermore, to better understand the prediction difference, we fine-tune a model on healthy node with a different random seed as a baseline to contextualize the impact of SDC variability against the impact of randomness in data ordering. More details can be found in Appendix E.

### 6.2 Results

**Results for Experiment IV.** Figure 7 shows the parameter difference, gradient norm and training loss on unhealthy nodes during pre-training. Despite training loss on each unhealthy node nearly identical to the healthy node, model weights on unhealthy nodes incrementally drift away from those on the

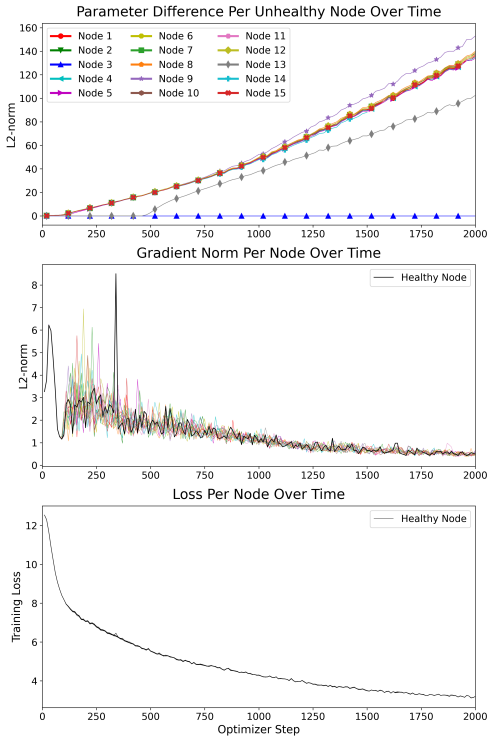



Figure 7: The curves for parameter difference, gradient norms and training loss on unhealthy nodes. Note that the loss curves on all unhealthy nodes are plotted but identical to that on the healthy node.

healthy node, suggesting that SDCs are pushing models towards different local minima.

Note that Node 13 shows no parameter difference before step 450, which indicates that no SDC occurs during this period. It is aligned with the finding in Section 4.2 that SDCs do not occur uniformly during training. After step 450, the model on Node 13 begins to quickly drift away from the ground-truth weights. We further find that the rates at which the parameter differences increase are similar on most unhealthy nodes, although the unhealthy nodes produce SDCs with different degrees of frequency and severity as shown in Section 4.2 This suggests that the rate of parameter drift is more likely to be driven by the sharp loss surface than purely by SDCs. In other words, SDCs serve as a trigger to push the optimization trajectory onto a different path through this sharp loss landscape, leading to the divergence of the parameters.

**Results for Experiment V.** Table 3 shows the fine-tuning results for three of the question-answering tasks on different nodes. The full results can be found in Appendix E.5. We find that the models fine-tuned on most unhealthy nodes are significantly better than the base model without fine-tuning and also have similar performance compared to the models fine-tuned on the healthy node.

| Configuration | CosmosQA TA (DP) | MATH TA (DP) | OpenbookQA TA (DP) |
|---|---|---|---|
| without fine-tuning | 56.33 (44.80) | 24.02 (82.35) | 74.70 (29.30) |
| Healthy Node | 90.79 (-) | 37.22 (-) | 83.80 (-) |
| Healthy Node (Seed=43) | 89.50 (6.70) | 38.83 (56.75) | 86.30 (18.70) |
| Unhealthy Node 1 | 90.53 (5.15) | 36.78 (42.24) | 85.00 (16.80) |
| Unhealthy Node 2 | 90.79 (0.00) | 37.22 (0.00) | 83.80 (0.00) |
| Unhealthy Node 3 | 90.77 (4.96) | 34.47 (41.84) | 83.40 (16.10) |
| Unhealthy Node 4 | 90.32 (5.59) | 36.42 (37.76) | 84.30 (16.60) |
| Unhealthy Node 5 | 90.79 (0.00) | 37.19 (34.91) | 85.10 (14.10) |
| Unhealthy Node 6 | 0.00 (100.00) | 36.92 (36.82) | 84.70 (16.30) |
| Unhealthy Node 7 | 90.32 (4.99) | 37.22 (0.00) | 83.80 (0.00) |
| Unhealthy Node 8 | 89.84 (6.23) | 38.22 (36.62) | 85.20 (15.50) |
| Unhealthy Node 9 | 89.97 (5.38) | 35.78 (37.49) | 85.00 (17.40) |
| Unhealthy Node 10 | 89.97 (4.93) | 37.05 (37.05) | 84.10 (17.20) |
| Unhealthy Node 11 | 90.61 (4.54) | 36.82 (38.53) | 87.10 (15.60) |
| Unhealthy Node 12 | 90.79 (0.00) | 37.22 (0.00) | 83.80 (0.00) |
| Unhealthy Node 13 | 90.79 (0.00) | 37.22 (0.00) | 83.80 (0.00) |
| Unhealthy Node 14 | 89.74 (6.12) | 38.63 (32.29) | 85.00 (15.90) |
| Unhealthy Node 15 | 90.77 (3.27) | 38.53 (40.87) | 83.80 (0.00) |

Table 3: Finetuning results for three question answering tasks on different nodes. For each task, we report the test accuracy (TA) and the disagreement percentage (DP).

The disagreement percentage caused by SDCs on unhealthy nodes is not larger than using a different random seed for data shuffling. Aligned with the findings in Experiment IV, this again affirms that SDCs on unhealthy nodes push the models towards different local minima.

However, SDCs are not necessarily harmless to model fine-tuning. Figure 8 shows the training loss during fine-tuning on CosmosQA and we find that significant loss spikes can occur on some unhealthy nodes. On Node 4, Node 6 and Node 7, the loss spikes occur in the middle of fine-tuning while later the training is again stabilized, which makes the final models still have benign performance. However, on Node 6, the loss spike occurs near the end of fine-tuning, which leads to the resulting model having zero test accuracy on CosmosQA as shown in Table 3. It indicates that the loss spikes caused by SDCs pose a threat to the model quality. We also note that loss spikes do not occur in every fine-tuning task. For example, training loss curves for OpenbookQA on unhealthy nodes are all identical to that on the healthy node, similar to the situation in Experiment IV. Therefore, we conclude that the impact of SDC on model training is closely related to the loss surface of the training task.

## 7 Discussion

### 7.1 Silent Nature of SDCs in LLM Training

Our results show that SDCs can silently occur without any clear indication from training loss. For example, we find from Section 4.2 and Section 5.2 that SDCs on Node 10 consistently perturb the submodule computation and the gradients, but the training loss on Node 10 is still identical to that

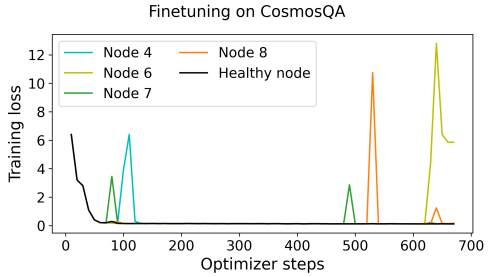


Figure 8: Training loss curves on different nodes during fine-tuning on CosmosQA dataset. The loss curves for other unhealthy nodes that are identical to the healthy node are not shown in this figure.

the on healthy node without any loss spike during both pre-training and fine-tuning in Section 6.2. It indicates that before a loss spike appears, SDCs may have already affected model training for an unknown period.

### 7.2 Mitigating the Impact of SDCs

Given its silent nature, it is important to mitigate the impact of SDC on LLM training. One direction is timely SDC detection. We examine if algorithm-based-fault tolerance (ABFT), specifically checksummed matrix multiplication (Smith and van de Geijn, 2015), could be used at the submodule level to detect SDCs. We find that ABFT fails to reliably detect SDCs with high precision and recall in Section 4.2, and we hypothesize further why this occurs in Appendix D: therefore, additional recomputation may be the best solution to reliably detect SDCs in practice.

One possible approach is using an additional shadow data-parallel replica during training. At each training step, the shadow replica chooses a target data-parallel rank whose inputs are used for its computation. The gradients from the shadow data-parallel replica and the target data-parallel rank can be compared during gradient all-reduce over data-parallel ranks to identify if an SDC has occurred. This idea is aligned with the SDC scanners on hot standbys used in training Gemini (Gemini Team, 2024).

Another direction is to mitigate the impact of SDCs on the training trajectory and the model quality when SDCs are not detected timely. As discussed in Section 6.2, the impact of SDCs on model quality is closely related to the sharpness of the loss surface. Therefore, one future direction is to conduct a deeper analysis of the connections between loss spikes and SDC. Future work could examine whether methods that reduce the sharpness of the loss surface or avoid optimization towards sharp regions (Lee and Park, 2023; Bahri et al., 2022) can help reduce the divergence of model parameters and loss spikes caused by SDCs.

## 8 Conclusion

In this work, we propose a study setting that enables us to thoroughly investigate the effects of real-world SDC on LLM training. Pairing healthy and unhealthy nodes together using our synchronization mechanisms, we isolate and examine the impacts of SDCs at different levels. We show that although in most cases the perturbations from SDCs on submodule computation and gradients are relatively small, SDCs can make models converge to different optima with different weights. We further show that SDCs can be evasive if we only monitor the training loss while in the extreme case, they can cause training loss spikes and fully corrupt the model. Our study reveals that the impact of SDC varies on different unhealthy nodes and is closely related to the loss surface of the training task. Our work further provides concrete insights for improving SDC detection and mitigating the impact of SDC in the future.

## 9 Limitations

We note a few limitations in this work. First, the number of unhealthy nodes we could access for our experiments was restricted because unhealthy nodes are rare due to rigorous manufacturer testing and filtering of hardware components before reaching the data-center. The unhealthy nodes retained after detection by the production fleet management flow and used in our study *were only temporarily held for the purpose of our study*. They are subsequently being repaired and returned to the production pool. Furthermore, we did not have physical access to hardware to investigate individual hardware components and more granular environment variable. For future work, we hope to investigate the effect of these system factors on SDC occurrence and training behavior.

Second, our study focuses on training with tensor parallelism only on a single node. Since each node exhibits different degrees of SDC, we could not run large-scale training with more complex parallelism strategies for every unhealthy node as we did in the tensor parallel setting due to resources and limited time. However, our experimental setting is still meaningful for understanding SDCs in LLM training. For large-scale LLM training, tensor

parallelism is commonly adopted with the degree set to be the number of accelerators in a single node (Narayanan et al., 2021), which makes our setting highly relevant. Furthermore, in our work, each model's training is fully computed on an unhealthy node, which makes the effects of SDCs more visible. Due to the heterogeneous SDC patterns across nodes, expanding this study and generalizing our findings to multi-node require exhaustive reruns across various node configurations. Therefore, we plan to explore more advanced mechanisms to dive deeper into the training dynamics and study how large the impact of SDC could be under the multi-node training setting in the future.

Third, our work shows a necessary trade-off between fine-grained analysis and SDC occurrence. Our synchronization mechanisms inevitably introduce additional overhead and reduce accelerator utilization, which can lead to a decrease in the frequency of SDCs. As shown in Figure 3 where we use computation synchronization at each submodule computation, Node 14 exhibits SDCs very infrequently over 4000 optimizer steps. By contrast, in Figure 6 where we use parameter synchronization at each step, the same node shows mismatched gradient norms at every step. The difference in SDC occurrence on the same node suggests more synchronization will decrease the accelerator utilization and further reduce the frequency of SDC. However, this trade-off is necessary to analyze SDCs at a finer granularity apart from high-level metrics like loss curves or gradient norms. Different from previous work that simulates SDCs, we do not know the exact occurrence pattern of SDC ahead of time on the real-world unhealthy nodes. As a result, without using the synchronization mechanisms, we cannot exactly isolate the impacts of SDCs at different granularities because multiple SDC occurrences and SDC-impacted outputs can be accumulated and propagate over consecutive computations. As such, the synchronization methods are required in our investigation setting to precisely isolate SDC impact. We will study how to mitigate the impact of this trade-off and propose more effective methods to analyze SDCs in the future.

Finally, we only observe the loss spikes in some fine-tuning experiments but not in our pre-training experiments. However, loss spikes can occur in practice during pre-training (Chowdhery et al., 2023). This discrepancy might be because the number of optimizer steps is not large enough to enter into a region where SDCs cause loss spikes or because the size of our model is not large enough. We also note that it can be challenging to reproduce and investigate the loss spikes caused by SDCs due to the randomness of SDCs. As observed in our work, if we monitor intermediate computation tensors during training, it decreases accelerator utilization, which affects the frequency of SDC and potentially prevents the occurrence of loss spikes. Future work can continue pre-training for a longer period on unhealthy nodes, potentially in a multi-node setting, to characterize the relation of loss spikes or NaN issues with SDC.

## Acknowledgments

We thank Thomas Fussell, Catalin Gabriel Manciu, Alexander Zhipa, Tushar Sharma, Manish Reddy, Stan Ivashkevich, Mohammad El-Shabani, Dave Goodell and Ron Diamant for providing advice.

## References

Udit Kumar Agarwal, Abraham Chan, and Karthik Pattabiraman. 2022. Lltfi: Framework agnostic fault injection for machine learning applications (tools and artifact track). In *2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*, pages 286–296. IEEE.

Udit Kumar Agarwal, Abraham Chan, and Karthik Pattabiraman. 2023. Resilience assessment of large language models under transient hardware faults. In *2023 IEEE 34th International Symposium on Software Reliability Engineering (ISSRE)*, pages 659–670. IEEE.

Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. 2023. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. *Preprint*, arXiv:2305.13245.

Aida Amini, Saadia Gabriel, Peter Lin, Rik Koncel-Kedziorski, Yejin Choi, and Hannaneh Hajishirzi. 2019. Mathqa: Towards interpretable math word problem solving with operation-based formalisms. *Preprint*, arXiv:1905.13319.

Dara Bahri, Hossein Mobahi, and Yi Tay. 2022. Sharpness-aware minimization improves language model generalization. *Preprint*, arXiv:2110.08529.

Robert C Baumann. 2005. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Transactions on Device and materials reliability*, 5(3):305–316.

Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. Training deep nets with sublinear memory cost. *Preprint*, arXiv:1604.06174.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. 2023. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240):1–113.

Christopher Clark, Kenton Lee, Ming-Wei Chang, Tom Kwiatkowski, Michael Collins, and Kristina Toutanova. 2019. Boolq: Exploring the surprising difficulty of natural yes/no questions. *Preprint*, arXiv:1905.10044.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Preprint*, arXiv:2205.14135.

Harish Dattatraya Dixit, Sneha Pendharkar, Matt Beadon, Chris Mason, Tejasvi Chakravarthy, Bharath Muthiah, and Sriram Sankar. 2021. Silent data corruptions at scale. *Preprint*, arXiv:2102.11245.

Erich Elsen, Curtis Hawthorne, and Arushi Somani. 2023. The adventure of the errant hardware.

Gemini Team. 2024. Gemini: A family of highly capable multimodal models. *Preprint*, arXiv:2312.11805.

Gene H. Golub and Charles F. Van Loan. 2013. *Matrix Computations*, fourth edition. The Johns Hopkins University Press.

Yi He, Prasanna Balaprakash, and Yanjing Li. 2020. Fidelity: Efficient resilience analysis framework for deep learning accelerators. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 270–281. IEEE.

Yi He, Mike Hutton, Steven Chan, Robert De Gruijl, Rama Govindaraju, Nishant Patil, and Yanjing Li. 2023a. Understanding and mitigating hardware failures in deep learning training systems. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–16.

Yi He, Mike Hutton, Steven Chan, Robert De Gruijl, Rama Govindaraju, Nishant Patil, and Yanjing Li. 2023b. Understanding and mitigating hardware failures in deep learning training systems. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, New York, NY, USA. Association for Computing Machinery.

Yi He and Yanjing Li. 2023. Understanding permanent hardware failures in deep learning training accelerator systems. In *2023 IEEE European Test Symposium (ETS)*, pages 1–6.

Peter H Hochschild, Paul Turner, Jeffrey C Mogul, Rama Govindaraju, Parthasarathy Ranganathan, David E Culler, and Amin Vahdat. 2021. Cores that don't count. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 9–16.

Yu-Shun Hsiao, Zishen Wan, Tianyu Jia, Radhika Ghosal, Abdulrahman Mahmoud, Arijit Raychowdhury, David Brooks, Gu-Yeon Wei, and Vijay Janapa Reddi. 2024. Silent data corruption in robot operating system: A case for end-to-end system-level fault analysis using autonomous uavs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 43(4):1037–1050.

Lifu Huang, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. 2019. Cosmos QA: Machine reading comprehension with contextual commonsense reasoning. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 2391–2401, Hong Kong, China. Association for Computational Linguistics.

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7b. *Preprint*, arXiv:2310.06825.

Vijay Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. 2022. Reducing activation recomputation in large transformer models. *Preprint*, arXiv:2205.05198.

Jack Kosaian and K. V. Rashmi. 2021. Arithmetic-intensity-guided fault tolerance for neural network inference on gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, volume 09183 of *SC '21*, page 1–15. ACM.

Guokun Lai, Qizhe Xie, Hanxiao Liu, Yiming Yang, and Eduard Hovy. 2017. Race: Large-scale reading comprehension dataset from examinations. *arXiv preprint arXiv:1704.04683*.

Joonhyung Lee, Jeongin Bae, Byeongwook Kim, Se Jung Kwon, and Dongsoo Lee. 2024. To fp8 and back again: Quantifying the effects of reducing precision on llm training stability. *Preprint*, arXiv:2405.18710.

Yonghyeon Lee and Frank Chongwoo Park. 2023. On explicit curvature regularization in deep generative models. *Preprint*, arXiv:2309.10237.

Guanpeng Li, Siva Kumar Sastry Hari, Michael Sullivan, Timothy Tsai, Karthik Pattabiraman, Joel Emer, and Stephen W. Keckler. 2017. Understanding error propagation in deep learning neural network (dnn) accelerators and applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, New York, NY, USA. Association for Computing Machinery.

Llama Team. 2024. The llama 3 herd of models. *Preprint*, arXiv:2407.21783.

Chang Lou, Yuzhuo Jing, and Peng Huang. 2022. Demystifying and checking silent semantic violations in large distributed systems. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 91–107.

Pan Lu, Swaroop Mishra, Tony Xia, Liang Qiu, Kai-Wei Chang, Song-Chun Zhu, Oyvind Tafjord, Peter Clark, and Ashwin Kalyan. 2022. Learn to explain: Multimodal reasoning via thought chains for science question answering. In *The 36th Conference on Neural Information Processing Systems (NeurIPS)*.

Dongning Ma, Xun Jiao, Fred Lin, Mengshi Zhang, Alban Desmaison, Thomas Sellinger, Daniel Moore, and Sriram Sankar. 2023. Evaluating and enhancing robustness of deep recommendation systems against hardware errors. *Preprint*, arXiv:2307.10244.

Dongning Ma, Fred Lin, Alban Desmaison, Joel Coburn, Daniel Moore, Sriram Sankar, and Xun Jiao. 2024. Dr. dna: Combating silent data corruptions in deep learning using distribution of neuron activations. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS '24, page 239–252, New York, NY, USA. Association for Computing Machinery.

Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. 2018. Mixed precision training. *Preprint*, arXiv:1710.03740.

Todor Mihaylov, Peter Clark, Tushar Khot, and Ashish Sabharwal. 2018. Can a suit of armor conduct electricity? a new dataset for open book question answering. In *EMNLP*.

Shubhendu S Mukherjee, Joel Emer, and Steven K Reinhardt. 2005. The soft error problem: An architectural perspective. In *11th International Symposium on High-Performance Computer Architecture*, pages 243–247. IEEE.

Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. 2021. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15.

George Papadimitriou, Dimitris Gizopoulos, Harish Dattatraya Dixit, and Sriram Sankar. 2023. Silent data corruptions: The stealthy saboteurs of digital integrity. In *2023 IEEE 29th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pages 1–7.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. Zero: Memory optimizations toward training trillion parameter models. *Preprint*, arXiv:1910.02054.

Rubens Luiz Rech and Paolo Rech. 2022. Reliability of google's tensor processing units for embedded applications. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 376–381. IEEE.

Amit Sabne. 2020. Xla : Compiling machine learning for peak performance.

Alan Sguigna. 2023. Silent Data Corruption: A Survey Article | ASSET InterTech — asset-intertech.com. `https://www.asset-intertech.com/resources/blog/2023/10/silent-data-corruption-a-survey-article/`. [Accessed 30-08-2024].

Noam Shazeer. 2020. Glu variants improve transformer. *Preprint*, arXiv:2002.05202.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2020. Megatron-lm: Training multi-billion parameter language models using model parallelism. *Preprint*, arXiv:1909.08053.

Tyler M. Smith and Robert A. van de Geijn. 2015. cs.utexas.edu. `https://www.cs.utexas.edu/~flame/pubs/FLAWN76.pdf`. [Accessed 09-09-2024].

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Preprint*, arXiv:1706.03762.

Zishen Wan, Karthik Swaminathan, Pin-Yu Chen, Nandhini Chandramoorthy, and Arijit Raychowdhury. 2022. Analyzing and improving resilience and robustness of autonomous systems. In *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*, pages 1–9.

Shaobu Wang, Guangyan Zhang, Junyu Wei, Yang Wang, Jiesheng Wu, and Qingchao Luo. 2023a. Understanding silent data corruptions in a large production cpu population. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 216–230, New York, NY, USA. Association for Computing Machinery.

Zhuang Wang, Zhen Jia, Shuai Zheng, Zhen Zhang, Xinwei Fu, TS Eugene Ng, and Yida Wang. 2023b. Gemini: Fast failure recovery in distributed training with in-memory checkpoints. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 364–381.

Jason Wei, Maarten Bosma, Vincent Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M Dai, and Quoc V Le. 2022. Finetuned language models are zero-shot learners. In *International Conference on Learning Representations*.

Wikimedia Foundation. 2023. Wikimedia downloads.

Panruo Wu, Nathan DeBardeleben, Qiang Guan, Sean Blanchard, Jieyang Chen, Dingwen Tao, Xin Liang, Kaiming Ouyang, and Zizhong Chen. 2017. Silent data corruption resilient two-sided matrix factorizations. In *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '17, page 415–427, New York, NY, USA. Association for Computing Machinery.

Panruo Wu, Chong Ding, Longxiang Chen, Teresa Davies, Christer Karlsson, and Zizhong Chen. 2013. On-line soft error correction in matrix–matrix multiplication. *Journal of Computational Science*, 4(6):465–472. Scalable Algorithms for Large-Scale Systems Workshop (ScalA2011), Supercomputing 2011.

Xinghua Xue, Cheng Liu, Haitong Huang, Bo Liu, Ying Wang, Bing Yang, Tao Luo, Lei Zhang, Huawei Li, and Xiaowei Li. 2023. Approxabft: Approximate algorithm-based fault tolerance for vision transformers. *Preprint*, arXiv:2302.10469.

Yujia Zhai, Elisabeth Giem, Kai Zhao, Jinyang Liu, Jiajun Huang, Bryan M. Wong, Christian R. Shelton, and Zizhong Chen. 2023. Ft-blas: A fault tolerant high performance blas implementation on x86 cpus. *IEEE Trans. Parallel Distrib. Syst.*, 34(12):3207–3223.

Jeff Jun Zhang, Tianyu Gu, Kanad Basu, and Siddharth Garg. 2018. Analyzing and mitigating the impact of permanent faults on a systolic array based neural network accelerator. In *2018 IEEE 36th VLSI Test Symposium (VTS)*, pages 1–6. IEEE.

Kai Zhao, Sheng Di, Sihuan Li, Xin Liang, Yujia Zhai, Jieyang Chen, Kaiming Ouyang, Franck Cappello, and Zizhong Chen. 2021. Algorithm-based fault tolerance for convolutional neural networks. *IEEE Transactions on Parallel and Distributed Systems*, page 1–1.

James F Ziegler. 1996. Terrestrial cosmic rays. *IBM journal of research and development*, 40(1):19–39.

## A Additional Background on SDCs

We further discuss background and prior work on detecting and correcting silent data corruption (SDC). Generally, SDCs cannot be guarded against effectively by common mechanisms like ECC memory (Sguigna, 2023) as they arise during computation. Therefore, several prior works have investigated SDC detection and correction.

**Simulating SDC Faults.** Some prior works focus on understanding how simulated SDC-like faults propagate through training and inference in deep learning systems. For example, He et al. (2023b) simulate hardware faults during training of several different deep learning architectures, characterize the error modes that result from simulated faults and error propagating through model training, and understand the conditions in which these faults must occur to destabilize training. However, relative to modern LLMs, the model sizes in their experiments are small and the training workloads are different compared with LLMs. Although the simulation is comprehensive, it is unknown how real-world SDC would affect LLM training in practice. In contrast, our work uses the real-world unhealthy hardware to train Transformer models with billions of parameters to understand how real-world SDCs behave in LLM training.

**Detection via Aggregate Statistics.** Some prior works focus on SDC detection primarily through only monitoring aggregate training statistics. For example, He and Li (2023) examines monitoring Inf/NaN results and loss spikes during training to identify when SDCs occur. Likewise, Ma et al. (2024) detect SDCs with high precision using error signatures derived from analyzing the distribution of model neuron activations.

**Soft Redundancy and Protecting Inference Computation.** However, since SDCs can also occur silently without impacting aggregate quantities like loss or gradient norm, other works add minor levels of redundancy to detect SDCs with higher precision. For example, algorithm based fault tolerance (ABFT) approaches compute low overhead checksum computations alongside the original operation to check against (Wu et al., 2017; Zhai et al., 2023). Most prior work using ABFT examines using detection in safety-critical applications (Kosaian and Rashmi, 2021) and specifically to protect deep learning inference computations(Zhao et al., 2021). More recent work examined using this to protect the inference of vision transformers (Xue et al., 2023).

**Detection with Exact Recomputation.** Finally, some works fully recompute values to check for SDC occurrence during training. For example, Gemini uses additional hardware to scan for SDCs and isolates incorrect computations by deterministically replaying computations (Gemini Team, 2024).

## B Model Pretraining Details

### B.1 Dataset and Preprocessing

In this section, we describe the dataset preprocessing done during the training experiments described

in Sections 4, 5, 6 and Appendix Section D. We train on the `20220301.en` split of the Wikipedia dataset (Wikimedia Foundation, 2023) with a sequence length of $L = 4096$ tokens. Using the full `20220301.en` split, we first tokenize the dataset using the pre-trained Byte-Pair Encoding (BPE) Llama-3 tokenizer (Llama Team, 2024) . We then concatenate the entire token sequence and chunk the dataset into chunks of sequence length 4096 to maximize context and accelerator utilization during training. The entire epoch of sequences is then shuffled with a fixed random seed by sequence then saved to disk. At training time, using `torch.distributed`, each tensor parallel rank ingests the same subset of data at the same time using a distributed dataloader via PyTorch/XLA's parallel dataloader[1], which provides buffering to hide CPU to device data load latency.

### B.2 Model Architecture

We use a Llama3-8B style model architecture (Llama Team, 2024) trained from Kaiming and Xavier uniform initialization on the weights and biases, where each decoder layer has 32 self-attention heads with group query attention (GQA) over 8 Key-Value heads and an feed-forward network using SwiGLU. All models are trained using tensor parallelism and ZeRO-1 with sequence parallelism (Rajbhandari et al., 2020; Korthikanti et al., 2022) with the XLA backend corresponding to the respective healthy-unhealthy node pair.

In this section, we describe the model architecture trained in the experiments described in Sections 4, 5, 6 and Appendix Section D. In the experiments described in Sections 4 and Appendix Section D, the model trained contains only 16 decoder block layers (and half the number of parameters as the Llama3-8B configuration), while in Sections 5 and 6, the model trained contains 32 decoder block layers and is equivalent to the Llama3-8B model configuration (with number of parameters). The details of each decoder block are given below, with sequence length $n = 4096$ and token dimension $d = 4096$:

1. Given an input of token embeddings $X \in \mathbb{R}_{n \times d}$ sharded by sequence length (dimension $n$), we perform an all-gather such that each TP rank has the entire input $X$ and perform Group Query Attention (GQA) (Ainslie et al., 2023) with 32 heads, head dimension of $d/32 = 128$, and 8 key-value heads. We do not use FlashAttention (Dao et al., 2022) for our model architecture, instead using a standard GQA implementation.
2. After the concatenation of head results and subsequent output row-parallel linear projection, the results are reduce-scattered to sequence parallelism, such that each TP rank has an equal split of tokens. We add a residual connection (add the original token embeddings) followed by a Layer Normalization.
3. We then enter the FFN primitive by all-gathering, such that each TP rank has the entire new input. We perform an standard FFN with Swish-Gated Linear Unit (SwiGLU) activation (Shazeer, 2020), projecting to an intermediate dimension of 16384 (4x), performing SwiGLU, then projecting back down into $d = 4096$. We perform the FFN with gradient checkpointing (Chen et al., 2016) on the intermediate dimension to save memory and avoid needing to persist forward matrices of size $4096 \times 16384$ to HBM for the backwards pass.

### B.3 Model Hyperparameters

Our model training hyperparameters are given below. For the primitive investigation in Section 4 and Appendix Section D, we train at a global batch size of 16, due to increased throughput from cross-node communication, while for the single and multiple optimizer step settings in Sections 5 and 6, we train at a global batch size of 256.

- Sequence length: 4096
- Embedding dimension: 4096
- Sharding strategy: ZeRO-1 with sequence parallelism
- Optimizer: Adam with $\beta_1 = 0.9$ and $\beta_2 = 0.99$
- Weight decay ($L_2$ regularization): 0.01
- Learning rate (LR) schedule: linear aramup, cosine annealing
- Total Steps (for LR schedule): 100,000
- Warmup Steps (for LR schedule): 2,000
- Training precision: `bfloat16`
- Mixed precision: False
- Micro-batch size (for gradient accum.): 1
- Gradient norm clipping with max norm 1.0
- Rounding mode: Round to Nearest

Specifically for Sections 4 (submodule investigation), we train $M = 4500$ optimizer steps using mixed-precision Adam (Micikevicius et al., 2018)

[1] https://github.com/pytorch/xla/blob/master/torch_xla/distributed/parallel_loader.py

and a global batch size of $B = 16$. Note that we use a smaller batch size and number of decoder layers than those in later experiments because computation synchronization brings additional cross-node communication and comparison, which greatly decreases training throughput and increases required memory usage.

Specifically for Sections 5, 6 (gradient and synchronization-free setting), we train $M = 2500$ optimizer steps using mixed-precision Adam (Micikevicius et al., 2018).

# C Submodule Investigation Implementation Details

## C.1 Submodule Investigation Integration with TorchXLA and Autograd

To integrate the lock-step communication mesh used in the primitive investigation in Section 4 into the forwards and backwards computations of a LLM training run, we developed a set of 'torch.autograd.Function' implementations, which implement the communication mesh as forwards or backwards behavior. We can then insert and call these functions at the locations in which we'd like to investigate the transformer primitive outputs. When `ComparisonForwardAutograd` is called on a tensor, it does no checks in the backwards pass but checks and outputs the computes SDC infrequency and severity for that tensor in the forward pass. When `ComparisonBackwardAutograd` is called on a tensor, it does no checks in the forwards pass but checks and outputs the computed SDC infrequency and severity for the gradient of the activation corresponding to that tensor (i.e. the tensor propogated backwards at that location in the backwards pass). For implementations, see Figure 10 below.

As in Figure 9, we insert `ComparisonFwdAutograd` calls at any of the red arrow locations, as we want to compare the computed forward tensor values for each corresponding set of TP ranks between healthy and unhealthy hosts prior to a reduce-scatter. Likewise, to check the values prior to the reduce scatter after backwards primitives, we insert calls to `ComparisonBwdAutograd` at the locations of the blue arrows, so that autograd will communicate and compare the backward pass input-gradients and return mismatch statistics.

This implementation allows the PyTorch autograd engine to automatically handle our primitive investigation communication mesh as it computes forwards and backwards passes for the model.

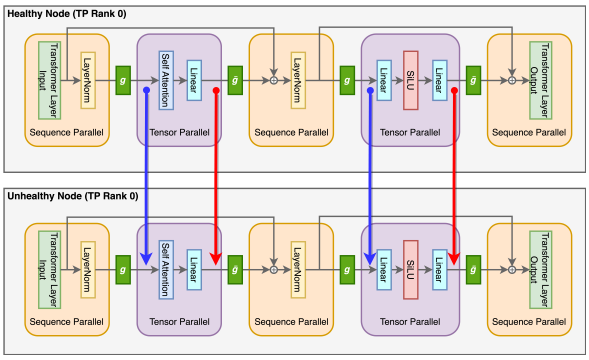



Figure 9: Illustration of a transformer decoder block under our "lockstep parallelism" in the primitive investigation setting, where the arrows indicate the intermediate tensors from the unhealthy corrected by corresponding tensors from the healthy host (red in forwards pass, blue in backwards pass) and where the PyTorch autograd functions in Figure 10 are inserted in implementation. Note that in the forward pass $g$ is an all-gather and $\bar{g}$ is a reduce-scatter, while in the backwards pass $g$ is an reduce-scatter and $\bar{g}$ is an all-gather.

## C.2 Detailed Primitive Investigation Results

We generally do not observe any per decoder layer trends in our results and provide detailed breakdowns of mismatch frequency and severity below.

### C.2.1 Frequency of Mismatching Elements

Detailed per node and per decoder results on the frequency of mismatching tensor elements in the forward and backwards passes are shown in Tables 6 and 7, respectively.

### C.2.2 Severity of Mismatching Elements

Detailed per node results on the average maximum severity of mismatching tensor elements in the forward and backwards passes are shown in Tables 8 and 9, respectively.

# D Algorithm Based Fault Tolerance (ABFT) Discussion

## D.1 Possible Explainations for ABFT Detection Failure

There are several possible reasons for why ABFT cannot reliably detect SDCs in our experiments. First, due to the low mismatch frequency, the overall impact on matrix norm could be smaller than floating point error bounds, which breaks the assumption of ABFT. Second, introducing ABFT changes the executed workload and decreases compute utilization in our implementation, which might

**Autograd Implementation**

```
@torch.no_grad()
def check_across_data_parallel_ranks(
  tensor_to_check: torch.tensor,
  check_args: CheckConfig,
  layer_count: int,
  tag: Optional[str] = None,
) -> Tuple[torch.Tensor, Optional[Dict[str, Any]]]:
  # For a given tensor:
  # (1) gathers the copy of the tensor on both the healthy and unhealthy host.
  # (2) computes the difference between them and calculates statistics on
  #     infrequency/severity of mismatching values.
  # (3) returns the correctly computed version of the tensor (from the healthy
  #     host) and computed error stats.
  ...
  return new_tensor, error_statistics

class ComparisonFwdAutograd(torch.autograd.Function):
  @staticmethod
  def forward(
    ctx, input_tensor, forward_args: CheckConfig, layer_count: int, tag: str
  ):
    # Check mismatching values and return error statistics in fwd pass.
    input_tensor, forward_error_statistics = check_across_data_parallel_ranks(
        input_tensor, forward_args, layer_count + 1, tag=tag
    )
    return input_tensor, forward_error_statistics

  @staticmethod
  def backward(ctx, grad_tensor, _unused_error_dict):
    # Do nothing on the bwd pass.
    return grad_tensor, None, None, None

class ComparisonBwdAutograd(torch.autograd.Function):
  @staticmethod
  def forward(
    ctx,
    input_tensor,
    backward_args: CheckConfig,
    layer_count: int,
    backwards_error_statistics: ErrorDict,
    prefix: str,
  ):
    # Do nothing in the fwd pass.
    ...
    return input_tensor

  @staticmethod
  def backward(ctx, grad_tensor):
    # Check mismatching values and return error statistics in bwd pass.
    new_grad_tensor, backwards_error_statistics = check_across_data_parallel_ranks(
        grad_tensor, ctx.backward_args, ctx.layer_count, tag=ctx.prefix
   )
   backwards_error_statistics = backwards_error_statistics.add_prefix(ctx.prefix)
   ctx.backwards_error_statistics.add_inplace(backwards_error_statistics)
   return new_grad_tensor, None, None, None, None
```

Figure 10: Abbreviated implementation of helper autograd functions, which are inserted into transformer primitive locations prior to the reduce-scatter to analyze tensors of interest. The autograd engine then handles the primitive investigation communication as we compute forward and backwards passes.

also decrease the rate of SDC occurrence. Finally, SDCs may occur outside the ABFT-protected matrix multiplication. More detail on each is provided below:

- *Low frequency and severity of SDC-induced tensor mismatches:* As noted in the submodule results in Section 4.2, the mismatch severity and frequency of silent data corruption errors results in low impact on matrix norms, which ABFT relies on to avoid flagging false positives. Assuming the SDC arises during matrix multiplication, the observed SDC occurrence is well within floating point error bounds and thus not flagged by ABFT methods. For example, for Node 10, we observe that 4.78$e$-3 of the forward attention output elements are perturbed by an worst-case factor of 1120 times. Using Eq. 5, we see that, in the worst case, when all of these errors lie on a single row, one of the row-sum values in $Cw$ changes by an estimated factor of $4.78\text{e}{-3} \times 1120 = 5.35$. Likewise, the LHS quantity $\|Cw - A(Bw)\|_\infty$ would be impacted by roughly the same estimated factor, which is less than an order of magnitude increase from the previous value and unlikely to cause a failure against the RHS threshold in Eq. 5.
- *ABFT overhead decreases accelerator utilization*: We observe that frequency of SDC occurrence is dependent on accelerator utilization adding ABFT overhead decreases utilization, which might decrease the rate of SDC occurrence. In Section 4.2, our results on the frequency of SDC-induced mismatches show that SDC occurrence potentially is a function of system-level metrics like power draw and overall system-level utilization. ABFT decreases accelerator utilization and potentially changes this system-level profile, reducing SDC occurrence.
- *SDCs occurring outside the matrix multiplication:* We hypothesized that checksummed matrix multiplication would be able to flag SDC errors due to matrix multiplications generally being the most compute-intensive and highest utilization stage of a deep learning model. However, ABFT flagging no SDCs on known unhealthy hosts may suggest that, preconditioned on existing manufacturer testing and vetting, SDCs might more commonly arise outside of matrix multiplication.

### D.2 ABFT and Reduced Precision Datatypes

We run our ABFT detection at `float32` to respect the assumptions to derive the threshold limits for ABFT. Despite lower precision datatypes like `bf16` and even `fp8` are commonly used in LLM training (Lee et al., 2024), the error bounds for ABFT are derived from IEEE-754 floating point (`fp32` or higher precision) error bounds for matrix multiplication (Golub and Van Loan, 2013). These bounds are derived under strict assumptions on the number of mantissa bits (and size of unit-roundoff), which do not hold for `fp16` or `bf16` datatypes. In the ABFT experiment, we ran our ABFT detection at `float32`: when running the check as-is at `bf16`, we observed several false positives on healthy nodes, confirming this conjecture.

We elaborate on reasons why algorithm-based fault tolerance (ABFT), specifically checksummed matrix multiplication, required additional consideration to be deployed in real-world LLM training settings. Specifically, we noted that checksummed matrix multiplication of floating point matricies was based in bound on floating point error, which make several assumptions on the unit-roundoff or machine epsilon of computation data types.

Recall that Wu et al. (2013) and Smith and van de Geijn (2015) proposed adding a checksum row and column to two-sided matrix multiplication for online floating-point aware detection of silent data corruption errors in matrix multiplication. For some matrix product $C = A \times B$, column vector $w$, we say an checksum error (and SDC) has occurred if:

$$\|Cw - A(Bw)\| > \tau \|A\|_\infty \|B\|_\infty \tag{7}$$

where $\tau = ku$, where $k$ is the contraction dimension of the matrix multiplication and $u$ is the unit-roundoff of the datatype precision used in the matrix multiplication. We can determine $u$ in PyTorch by using the function `torch.finfo(<dtype>).eps`. The above threshold is derived under the assumptions of Equation 2.7.11 from Golub and Van Loan (2013), a bound for round-off error in dot products, restated below for convenience.

*Equation 2.7.11*: If $n\mathbf{u} \leq 0.01$ where $n$ is the size of the dot-product shared dimension, $\mathbf{u}$ is the unit round-off of the datatype used for computation, and $fl(x^T y)$ then

$$|fl(x^T y) - x^t y| \leq 1.01 n \mathbf{u} |x|^T |y|$$

For types like `float32` or `float64` (and their corresponding $\mathbf{u}$ values) the assumption $n\mathbf{u} \leq 0.01$ is very reasonable. In our model configuration case, for a model with embedding dimension 4096:

1. `float32`: $\mathbf{u} = 1.19215e-07, n\mathbf{u} \approx 0.0005 < 0.01$
2. `float16`: $\mathbf{u} = 0.0009765625, n\mathbf{u} \approx 4 \not< 0.01$
3. `bfloat16`: $\mathbf{u} = 0.0078125, n\mathbf{u} \approx 32 \not< 0.01$

We see that for types less precise than `float16`, ABFT error bound in Equation 7 no longer always holds true and checksummed matrix multiplication methods would possibly raise false positives. We observed this empirically to be true, where the above threshold at `bfloat16` raised false positive flags on healthy nodes that passed production margin tests, while raising no errors on the same node at `float32`. Thus, more work in error analysis is needed to extend ABFT methods properly to low precision datatypes.

# E Finetuning Experiment Details

## E.1 Dataset Details

Details and a brief description of each dataset are provided below:

1. ScienceQA (Lu et al., 2022) is a multiple-choice question answering (QA) dataset of grade school science and humanities questions: note that we remove all questions with an image context.
2. BoolQ (Clark et al., 2019) is a QA dataset for naturally occuring yes/no questions each with a page of contextual and relevant information.
3. OpenbookQA (Mihaylov et al., 2018) is a dataset containing multiple-choice questions each with context modeled like an open-book exam, requiring multi-step reasoning, use of additional common and commonsense knowledge, and rich text comprehension.
4. MathQA (Amini et al., 2019) is a datset of English multiple-choice math word problems covering multiple math domain categories.
5. RACE (Lai et al., 2017) is a large-scale reading comprehension multiple-choice question answering dataset, collected from English examinations in China, which are designed for middle school and high school students.

## E.2 Finetuning Prompts

In Figure 11, we show our prompts for finetuning, which are structrued in the following general form. During training we include the full context including the correct answer, while during evaluation we remove everything after `### CORRECT ANSWER` and ask the model to continue generating the answer.

**Finetuning Prompt Structure**

```
### QUESTION
{question}

### CONTEXT
{context}

### CHOICES
A: {choice1}
B: {choice2}
...

### CORRECT ANSWER
A: {choice 1}
```

Figure 11: General finetuning prompt structure.

## E.3 Masking and Padding Details

For finetuning, we right-pad sequences to a fixed sequence length of 2048 tokens (with the exception of 4096 for the RACE dataset) using the Mistral tokenizer `EOS` token. Furthermore, we mask out padding tokens during training so that they do not contribute to loss and gradient computation by setting the `labels` for padding token positions to `-100`, so that they are ignored by PyTorch cross-entropy loss calculation [2].

## E.4 Hyperparameters and Finetuning Configuration

We use the HuggingFace `optimum` package to finetune the `mistralai/Mistral-7B-v0.3` model on our experiment nodes using tensor parallelism and ZeRO-1 optimizer with sequence parallelism. For finetuning, we use the following hyperparameters.

- Sequence length: 4096 for RACE, 2048 otherwise
- Sharding strategy: ZeRO-1 with sequence parallelism
- Optimizer: Adam with $\beta_1 = 0.9$ and $\beta_2 = 0.999$
- Weight decay ($L_2$ regularization): 0.001
- Learning rate schedule: constant with linear warmup.
- Warmup Steps (for computing cosine LR scheduler): 5% of total steps

[2] https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html

- Training precision: `bfloat16`
- Global batch size: 32
- Micro-batch size (for gradient accum.): 1
- Gradient norm clipping with max norm 0.3
- Rounding mode: Round to Nearest

For learning rates, we chose the following values, noted alongside their training dataset size. These were tuned such that the finetuning on the train split improves the model on evaluation set performance on a healthy, non-SDC producing node.

Table 4: Finetuning learning rate (LR) and dataset size for each dataset used in Experiment V.

| DATASET | LR | DATASET SIZE (# TRAIN SEQS.) |
|---|---|---|
| BOOLQ | 5$e$-6 | 9,430 |
| COSMOSQA | 5$e$-6 | 25,260 |
| MATHQA | 1$e$-5 | 29,837 |
| OPENBOOKQA | 2$e$-6 | 11,920 |
| SCIENCEQA | 1$e$-6 | 12,700 |
| RACE | 1$e$-5 | 87,900 |

### E.5 Full Finetuning Test Accuracy and Disagreement Percentage Results

The full table of finetuning results across all datasets is shown below in Table 5. We again note that across all datasets, finetuned models on unhealthy nodes achieve improved performance over the unfinetuned baseline and similar (though differing performance) compared to models deterministically finetuned on healthy nodes. Specficially, we observe, that this disagreement percentage (or delta from the healthy node finetuning) is comparable to finetuning under a differnt dataset shuffling seed.

| Configuration | CosmosQA | | MathQA | | ScienceQA | | OpenbookQA | | BoolQ | | RACE | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | TA | DP | TA | DP | TA | DP | TA | DP | TA | DP | TA | DP |
| without fine-tuning | 56.33 | 44.80 | 24.02 | 82.35 | 72.62 | 34.22 | 74.70 | 29.30 | 70.21 | 27.89 | 73.63 | 25.44 |
| Healthy Node | 90.79 | - | 37.22 | - | 84.17 | - | 83.80 | - | 90.06 | - | 87.47 | - |
| Healthy Node (seed=43) | 89.50 | 6.70 | 38.83 | 56.75 | 87.90 | 14.57 | 86.30 | 18.70 | 90.31 | 5.26 | 87.62 | 9.95 |
| Unhealthy Node 1 | 90.53 | 5.15 | 36.78 | 42.24 | 86.74 | 13.26 | 85.00 | 16.80 | 89.94 | 3.49 | 87.82 | 6.61 |
| Unhealthy Node 2 | 90.79 | 0.00 | 37.22 | 0.00 | 84.17 | 0.00 | 83.80 | 0.00 | 90.06 | 0.00 | 87.47 | 0.00 |
| Unhealthy Node 3 | 90.77 | 4.96 | 34.47 | 41.84 | 85.79 | 14.61 | 83.40 | 16.10 | 90.21 | 3.64 | 87.47 | 0.00 |
| Unhealthy Node 4 | 90.32 | 5.59 | 36.42 | 37.76 | 85.48 | 15.38 | 84.30 | 16.60 | 90.61 | 4.10 | 88.12 | 6.75 |
| Unhealthy Node 5 | 90.79 | 0.00 | 37.19 | 34.91 | 84.17 | 0.00 | 85.10 | 14.10 | 90.06 | 0.00 | 87.47 | 6.45 |
| Unhealthy Node 6 | 0.00 | 100.00 | 36.92 | 36.82 | 84.26 | 2.52 | 84.70 | 16.30 | 90.03 | 2.66 | 0.00 | 100.00 |
| Unhealthy Node 7 | 90.32 | 4.99 | 37.22 | 0.00 | 85.21 | 17.81 | 83.80 | 0.00 | 90.80 | 3.36 | 87.41 | 6.36 |
| Unhealthy Node 8 | 89.84 | 6.23 | 38.22 | 36.62 | 85.66 | 12.14 | 85.20 | 15.50 | 90.49 | 3.00 | 87.11 | 6.61 |
| Unhealthy Node 9 | 89.97 | 5.38 | 35.78 | 37.49 | 85.30 | 21.49 | 85.00 | 17.40 | 90.12 | 3.43 | 87.41 | 6.57 |
| Unhealthy Node 10 | 89.97 | 4.93 | 37.05 | 37.05 | 88.31 | 15.15 | 84.10 | 17.20 | 90.64 | 3.09 | 87.94 | 6.59 |
| Unhealthy Node 11 | 90.61 | 4.54 | 36.82 | 38.53 | 85.57 | 16.41 | 87.10 | 15.60 | 90.31 | 3.73 | 87.15 | 6.95 |
| Unhealthy Node 12 | 90.79 | 0.00 | 37.22 | 0.00 | 84.17 | 0.00 | 83.80 | 0.00 | 90.06 | 0.00 | 87.47 | 0.00 |
| Unhealthy Node 13 | 90.79 | 0.00 | 37.22 | 0.00 | 84.17 | 0.00 | 83.80 | 0.00 | 90.06 | 0.00 | 87.11 | 6.77 |
| Unhealthy Node 14 | 89.74 | 6.12 | 38.63 | 32.29 | 84.17 | 0.00 | 85.00 | 15.90 | 90.06 | 0.00 | 87.62 | 6.40 |
| Unhealthy Node 15 | 90.77 | 3.27 | 38.53 | 40.87 | 84.17 | 0.00 | 83.80 | 0.00 | 90.06 | 0.00 | 87.39 | 6.61 |

Table 5: Finetuning results for six question answering tasks on different nodes. For each task, we report the test accuracy (TA) and the disagreement percentage (DP). By default, the random seed for data shuffling is set as 42.

| | | SDC Node ID | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Decoder Layer | Primitive | Node 1 | Node 4 | Node 6 | Node 7 | Node 8 | Node 9 | Node 10 | Node 11 | Node 14 |
| 1 | fwd/attn | 2.84e-5 | 3.31e-9 | 0.00e+0 | 1.02e-6 | 1.59e-9 | 2.92e-6 | 0.00e+0 | 1.43e-2 | 0.00e+0 |
| | fwd/ffn | 5.17e-7 | 4.37e-11 | 0.00e+0 | 5.18e-7 | 9.95e-12 | 1.52e-7 | 9.75e-6 | 2.40e-3 | 0.00e+0 |
| 2 | fwd/attn | 1.45e-5 | 2.47e-9 | 3.89e-9 | 2.94e-6 | 1.70e-9 | 9.83e-6 | 8.62e-4 | 3.89e-2 | 1.82e-10 |
| | fwd/ffn | 2.86e-7 | 1.02e-10 | 2.39e-14 | 3.48e-8 | 1.42e-11 | 2.58e-7 | 2.18e-4 | 2.04e-3 | 0.00e+0 |
| 3 | fwd/attn | 9.94e-6 | 1.78e-9 | 0.00e+0 | 1.76e-6 | 4.44e-9 | 6.31e-6 | 1.78e-3 | 2.67e-2 | 0.00e+0 |
| | fwd/ffn | 5.21e-7 | 1.42e-10 | 0.00e+0 | 9.43e-8 | 6.45e-12 | 3.74e-7 | 4.75e-4 | 2.24e-3 | 0.00e+0 |
| 4 | fwd/attn | 8.82e-6 | 2.40e-9 | 0.00e+0 | 2.02e-6 | 3.86e-9 | 5.24e-6 | 1.63e-3 | 2.02e-2 | 2.61e-10 |
| | fwd/ffn | 4.83e-7 | 1.05e-10 | 0.00e+0 | 5.85e-8 | 1.85e-11 | 3.84e-7 | 4.66e-4 | 2.38e-3 | 0.00e+0 |
| 5 | fwd/attn | 1.10e-5 | 2.99e-9 | 0.00e+0 | 1.99e-6 | 2.45e-9 | 6.85e-6 | 2.77e-3 | 2.34e-2 | 3.11e-10 |
| | fwd/ffn | 4.96e-7 | 9.04e-11 | 0.00e+0 | 1.07e-7 | 6.52e-12 | 3.53e-7 | 6.79e-4 | 2.19e-3 | 0.00e+0 |
| 6 | fwd/attn | 1.23e-5 | 3.57e-9 | 0.00e+0 | 2.13e-6 | 5.14e-10 | 8.70e-6 | 3.98e-3 | 2.59e-2 | 0.00e+0 |
| | fwd/ffn | 6.07e-7 | 1.27e-10 | 8.67e-11 | 1.35e-7 | 2.77e-11 | 4.48e-7 | 9.17e-4 | 2.36e-3 | 0.00e+0 |
| 7 | fwd/attn | 1.05e-5 | 1.29e-9 | 2.33e-8 | 1.78e-6 | 1.03e-8 | 5.37e-6 | 4.39e-3 | 2.15e-2 | 0.00e+0 |
| | fwd/ffn | 6.27e-7 | 9.07e-11 | 0.00e+0 | 1.20e-7 | 7.15e-11 | 4.72e-7 | 1.18e-3 | 2.36e-3 | 0.00e+0 |
| 8 | fwd/attn | 1.69e-5 | 3.78e-9 | 0.00e+0 | 2.14e-6 | 4.47e-9 | 1.13e-5 | 5.08e-3 | 2.91e-2 | 0.00e+0 |
| | fwd/ffn | 5.01e-7 | 1.28e-10 | 0.00e+0 | 1.43e-7 | 1.24e-11 | 5.12e-7 | 1.19e-3 | 2.32e-3 | 0.00e+0 |
| 9 | fwd/attn | 1.61e-5 | 1.04e-8 | 2.39e-14 | 2.45e-6 | 2.12e-9 | 1.70e-5 | 6.94e-3 | 3.63e-2 | 1.02e-10 |
| | fwd/ffn | 6.52e-7 | 1.03e-10 | 0.00e+0 | 1.27e-7 | 6.00e-11 | 5.22e-7 | 1.44e-3 | 2.23e-3 | 0.00e+0 |
| 10 | fwd/attn | 1.36e-5 | 1.74e-9 | 8.56e-11 | 2.34e-6 | 8.23e-9 | 1.45e-5 | 6.57e-3 | 3.20e-2 | 0.00e+0 |
| | fwd/ffn | 5.62e-7 | 6.93e-11 | 0.00e+0 | 1.38e-7 | 8.89e-12 | 5.67e-7 | 1.40e-3 | 2.19e-3 | 0.00e+0 |
| 11 | fwd/attn | 1.96e-5 | 4.25e-9 | 0.00e+0 | 2.29e-6 | 7.92e-11 | 1.46e-5 | 6.78e-3 | 3.45e-2 | 0.00e+0 |
| | fwd/ffn | 5.03e-7 | 7.97e-11 | 8.68e-11 | 9.69e-8 | 8.55e-12 | 6.67e-7 | 1.36e-3 | 2.23e-3 | 0.00e+0 |
| 12 | fwd/attn | 1.97e-5 | 3.46e-9 | 0.00e+0 | 2.58e-6 | 2.03e-9 | 1.50e-5 | 6.81e-3 | 3.49e-2 | 0.00e+0 |
| | fwd/ffn | 5.63e-7 | 7.24e-11 | 8.88e-11 | 1.10e-7 | 9.51e-12 | 6.78e-7 | 1.35e-3 | 2.38e-3 | 0.00e+0 |
| 13 | fwd/attn | 2.28e-5 | 9.72e-9 | 8.21e-11 | 3.01e-6 | 4.83e-14 | 1.86e-5 | 7.05e-3 | 4.09e-2 | 8.56e-11 |
| | fwd/ffn | 4.35e-7 | 8.06e-11 | 0.00e+0 | 5.00e-8 | 5.72e-12 | 6.67e-7 | 1.26e-3 | 2.20e-3 | 0.00e+0 |
| 14 | fwd/attn | 1.78e-5 | 3.91e-9 | 4.79e-14 | 2.37e-6 | 4.67e-9 | 1.47e-5 | 7.66e-3 | 3.50e-2 | 0.00e+0 |
| | fwd/ffn | 3.83e-7 | 6.99e-11 | 2.39e-14 | 2.84e-8 | 8.84e-12 | 7.20e-7 | 1.49e-3 | 2.13e-3 | 0.00e+0 |
| 15 | fwd/attn | 1.39e-5 | 1.33e-9 | 0.00e+0 | 1.75e-6 | 2.81e-9 | 1.31e-5 | 7.18e-3 | 2.61e-2 | 0.00e+0 |
| | fwd/ffn | 4.66e-7 | 8.00e-11 | 0.00e+0 | 8.01e-8 | 4.49e-11 | 6.76e-7 | 1.65e-3 | 2.20e-3 | 0.00e+0 |
| 16 | fwd/attn | 1.25e-5 | 4.28e-9 | 4.79e-14 | 1.43e-6 | 2.02e-9 | 1.11e-5 | 6.93e-3 | 2.43e-2 | 9.60e-11 |
| | fwd/ffn | 4.86e-7 | 8.85e-11 | 0.00e+0 | 4.40e-8 | 5.02e-12 | 6.33e-7 | 1.46e-3 | 2.11e-3 | 0.00e+0 |

Table 6: Results for mismatch frequency for each transformer primitive forward, separated by decoder layer and averaged across all microsteps. The unhealthy Nodes 2, 3, 5, 12, 13 and 15 did not exhibit any mismatching tensors in forward passes in this experimental setting and thus are excluded from the table.

| | | SDC Node ID | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Decoder | Primitive | Node 1 | Node 4 | Node 5 | Node 6 | Node 7 | Node 8 | Node 9 | Node 10 | Node 11 | Node 13 | Node 14 | Node 15 |
| 16 | BWD/FFN | 3.12e-06 | 5.89e-11 | 6.21e-13 | 0 | 7.04e-08 | 2.55e-09 | 3.71e-08 | 7.89e-05 | 9.92e-05 | 4.91e-11 | 1.44e-09 | 7.09e-14 |
| | BWD/ATTN | 2.11e-04 | 3.08e-09 | 0 | 0 | 4.44e-06 | 1.40e-07 | 5.57e-06 | 2.59e-03 | 7.19e-03 | 0 | 3.09e-10 | 0 |
| 15 | BWD/FFN | 3.11e-06 | 6.19e-11 | 1.58e-12 | 0 | 7.21e-08 | 2.46e-09 | 3.87e-08 | 8.45e-05 | 1.06e-04 | 9.57e-11 | 7.33e-10 | 0 |
| | BWD/ATTN | 1.81e-04 | 2.10e-09 | 0 | 0 | 4.07e-06 | 1.22e-07 | 5.07e-06 | 2.43e-03 | 6.56e-03 | 0 | 0 | 0 |
| 14 | BWD/FFN | 3.09e-06 | 1.87e-11 | 1.46e-12 | 2.13e-11 | 6.89e-08 | 2.43e-09 | 4.12e-08 | 8.10e-05 | 1.02e-04 | 1.33e-10 | 2.14e-09 | 0 |
| | BWD/ATTN | 2.63e-04 | 2.24e-09 | 0 | 0 | 5.96e-06 | 1.62e-07 | 6.75e-06 | 2.78e-03 | 8.51e-03 | 0 | 1.64e-11 | 0 |
| 13 | BWD/FFN | 3.09e-06 | 2.08e-11 | 4.06e-12 | 4.74e-10 | 7.04e-08 | 2.45e-09 | 4.02e-08 | 8.03e-05 | 1.01e-04 | 3.24e-10 | 9.38e-10 | 2.36e-14 |
| | BWD/ATTN | 2.85e-04 | 8.18e-09 | 0 | 6.14e-10 | 6.88e-06 | 1.89e-07 | 7.73e-06 | 2.70e-03 | 8.91e-03 | 0 | 4.79e-14 | 0 |
| 12 | BWD/FFN | 3.08e-06 | 3.69e-11 | 1.62e-12 | 3.45e-11 | 7.37e-08 | 2.43e-09 | 4.19e-08 | 8.54e-05 | 1.13e-04 | 1.23e-10 | 6.67e-10 | 0 |
| | BWD/ATTN | 2.48e-04 | 3.10e-09 | 0 | 0 | 6.49e-06 | 1.56e-07 | 6.14e-06 | 2.56e-03 | 8.17e-03 | 0 | 2.39e-14 | 0 |
| 11 | BWD/FFN | 3.05e-06 | 2.71e-11 | 5.02e-13 | 2.98e-11 | 7.21e-08 | 2.35e-09 | 4.24e-08 | 8.55e-05 | 1.06e-04 | 1.70e-10 | 1.04e-09 | 0 |
| | BWD/ATTN | 2.00e-04 | 4.44e-09 | 0 | 0 | 4.82e-06 | 1.21e-07 | 5.52e-06 | 2.30e-03 | 7.42e-03 | 0 | 7.18e-14 | 0 |
| 10 | BWD/FFN | 3.02e-06 | 2.42e-11 | 3.35e-13 | 0 | 7.83e-08 | 2.34e-09 | 4.00e-08 | 8.64e-05 | 1.00e-04 | 3.44e-11 | 1.49e-09 | 0 |
| | BWD/ATTN | 1.65e-04 | 2.88e-09 | 0 | 5.40e-10 | 4.48e-06 | 1.05e-07 | 5.13e-06 | 2.30e-03 | 6.94e-03 | 0 | 2.39e-14 | 0 |
| 9 | BWD/FFN | 2.97e-06 | 2.79e-11 | 3.01e-12 | 0 | 7.12e-08 | 2.32e-09 | 3.99e-08 | 8.70e-05 | 1.09e-04 | 2.41e-10 | 5.32e-10 | 0 |
| | BWD/ATTN | 1.77e-04 | 4.29e-09 | 0 | 0 | 4.65e-06 | 1.20e-07 | 5.41e-06 | 2.38e-03 | 7.29e-03 | 0 | 3.03e-10 | 0 |
| 8 | BWD/FFN | 2.91e-06 | 2.29e-11 | 1.19e-13 | 0 | 7.08e-08 | 2.26e-09 | 4.05e-08 | 8.10e-05 | 1.12e-04 | 6.64e-11 | 3.18e-10 | 0 |
| | BWD/ATTN | 1.31e-04 | 4.38e-09 | 0 | 0 | 3.70e-06 | 8.33e-08 | 3.62e-06 | 1.91e-03 | 6.14e-03 | 0 | 4.79e-14 | 0 |
| 7 | BWD/FFN | 2.83e-06 | 2.09e-11 | 2.70e-12 | 0 | 6.70e-08 | 2.21e-09 | 3.95e-08 | 8.51e-05 | 1.12e-04 | 1.04e-10 | 2.98e-09 | 0 |
| | BWD/ATTN | 9.79e-05 | 1.14e-09 | 0 | 1.96e-08 | 2.91e-06 | 6.57e-08 | 2.35e-06 | 1.77e-03 | 5.66e-03 | 0 | 2.39e-14 | 0 |
| 6 | BWD/FFN | 2.73e-06 | 2.90e-11 | 2.44e-12 | 3.31e-11 | 6.51e-08 | 2.15e-09 | 3.92e-08 | 7.85e-05 | 1.13e-04 | 1.74e-10 | 3.99e-09 | 2.36e-14 |
| | BWD/ATTN | 1.05e-04 | 1.66e-09 | 0 | 0 | 2.98e-06 | 6.47e-08 | 3.14e-06 | 1.54e-03 | 5.96e-03 | 0 | 2.85e-10 | 0 |
| 5 | BWD/FFN | 2.63e-06 | 2.75e-11 | 9.56e-14 | 0 | 5.98e-08 | 2.07e-09 | 3.72e-08 | 7.64e-05 | 1.06e-04 | 2.26e-10 | 8.34e-09 | 0 |
| | BWD/ATTN | 9.13e-05 | 3.35e-09 | 0 | 0 | 3.19e-06 | 5.74e-08 | 2.67e-06 | 1.29e-03 | 5.54e-03 | 0 | 1.73e-09 | 0 |
| 4 | BWD/FFN | 2.54e-06 | 2.14e-11 | 7.17e-14 | 0 | 5.38e-08 | 2.02e-09 | 3.77e-08 | 7.44e-05 | 1.13e-04 | 8.00e-11 | 3.22e-09 | 0 |
| | BWD/ATTN | 7.84e-05 | 1.84e-09 | 0 | 1.27e-10 | 2.97e-06 | 5.82e-08 | 2.28e-06 | 1.05e-03 | 5.14e-03 | 0 | 1.70e-09 | 0 |
| 3 | BWD/FFN | 2.43e-06 | 2.24e-11 | 7.17e-14 | 0 | 5.32e-08 | 1.97e-09 | 3.78e-08 | 7.65e-05 | 1.07e-04 | 3.70e-11 | 8.23e-09 | 0 |
| | BWD/ATTN | 6.95e-05 | 1.62e-09 | 2.39e-14 | 0 | 2.56e-06 | 4.71e-08 | 2.45e-06 | 1.05e-03 | 5.87e-03 | 0 | 2.24e-09 | 0 |
| 2 | BWD/FFN | 2.28e-06 | 1.63e-11 | 0 | 0 | 4.31e-08 | 1.80e-09 | 3.36e-08 | 6.95e-05 | 1.02e-04 | 7.85e-12 | 2.47e-09 | 0 |
| | BWD/ATTN | 8.15e-05 | 1.15e-09 | 0 | 3.03e-09 | 3.86e-06 | 6.08e-08 | 3.32e-06 | 1.06e-03 | 7.60e-03 | 0 | 1.27e-09 | 0 |
| 1 | BWD/FFN | 2.12e-06 | 1.14e-11 | 1.34e-12 | 3.70e-10 | 8.74e-08 | 1.62e-09 | 3.03e-08 | 6.46e-05 | 1.33e-04 | 6.57e-11 | 9.29e-09 | 0 |
| | BWD/ATTN | 1.03e-04 | 2.30e-09 | 0 | 0 | 5.05e-06 | 5.76e-08 | 2.10e-06 | 1.07e-03 | 4.40e-03 | 0 | 2.39e-14 | 0 |

Table 7: Frequency of mismatching tensor elements for each transformer primitive backward, separated down by decoder layer and averaged across all microsteps. The unhealthy Nodes 2, 3, 5, 12, and 13 did not exhibit any mismatching tensors in backward passes in this experimental setting and thus are excluded from the table.

| | | SDC Node ID | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Decoder Layer | Primitive | Node 1 | Node 4 | Node 6 | Node 7 | Node 8 | Node 9 | Node 10 | Node 11 | Node 14 |
| 1 | fwd/attn | 1.82 | 0.3047 | 0 | 4.7812 | 13.5 | 0.15039063 | 0 | 25.875 | 0 |
| | fwd/ffn | 63.25 | 0.0512 | 0 | 16.6250 | 0.3867 | 21.625 | 2.6875 | 442 | 0 |
| 2 | fwd/attn | 1.04 | 0.3515 | 0.0366 | 24.625 | 0.1289 | 1.4140625 | 4.84375 | 90.5 | 0.4746 |
| | fwd/ffn | 312 | 0.0266 | 1 | 1.3046 | 0.0981 | 60.25 | 80 | 668 | 0 |
| 3 | fwd/attn | 3.97 | 2.9531 | 0 | 6.125 | 2.7031 | 6.4375 | 12.9375 | 230 | 0 |
| | fwd/ffn | 84.5 | 0.0439 | 0 | 9.8125 | 0.2793 | 15.5625 | 42.5 | 848 | 0 |
| 4 | fwd/attn | 4.91 | 0.375 | 0 | 6.8438 | 0.7148 | 4.59375 | 24.25 | 188 | 1.1172 |
| | fwd/ffn | 92 | 0.8555 | 0 | 1.7734 | 0.7734 | 130 | 145 | 640 | 0 |
| 5 | fwd/attn | 11.63 | 0.1699 | 0 | 9.375 | 0.6406 | 7.75 | 1120 | 127 | 0.3613 |
| | fwd/ffn | 18.38 | 0.0556 | 0 | 17.125 | 0.1338 | 30.75 | 118 | 708 | 0 |
| 6 | fwd/attn | 99 | 0.3281 | 0 | 11.9375 | 4.2187 | 16.5 | 58.75 | 152 | 0 |
| | fwd/ffn | 143 | 0.0737 | 0.0933 | 25 | 4.0312 | 24.75 | 67.5 | 298 | 0 |
| 7 | fwd/attn | 14.63 | 0.1069 | 0.3847 | 43.25 | 0.7695 | 12.9375 | 74.5 | 76.5 | 0 |
| | fwd/ffn | 88.5 | 0.0181 | 0 | 2.4219 | 0.1719 | 200 | 40.5 | 334 | 0 |
| 8 | fwd/attn | 12.94 | 0.0933 | 0 | 6.9375 | 0.5898 | 7.15625 | 36.75 | 76 | 0 |
| | fwd/ffn | 145 | 0.0913 | 0 | 6.125 | 4.375 | 97 | 47.25 | 422 | 0 |
| 9 | fwd/attn | 38.5 | 0.7852 | 0.9961 | 1.9063 | 1.1016 | 18.25 | 67 | 318 | 0.0786 |
| | fwd/ffn | 148 | 0.0579 | 0 | 25.625 | 5.6875 | 21.875 | 53 | 214 | 0 |
| 10 | fwd/attn | 9.81 | 1.125 | 0.1289 | 4.5938 | 4.8125 | 6.46875 | 42.75 | 136 | 0 |
| | fwd/ffn | 119.5 | 0.0287 | 0 | 28.375 | 0.1533 | 30.625 | 107.5 | 398 | 0 |
| 11 | fwd/attn | 11.69 | 0.1650 | 0 | 6.0625 | 0.6015 | 14.3125 | 53 | 121 | 0 |
| | fwd/ffn | 33.75 | 0.0425 | 0.1064 | 13.8125 | 1.3437 | 35 | 76.5 | 346 | 0 |
| 12 | fwd/attn | 25.5 | 0.2832 | 0 | 25.75 | 0.4160 | 3.21875 | 27.625 | 34 | 0 |
| | fwd/ffn | 23.25 | 0.0349 | 1 | 88.5 | 0.4394 | 139 | 262 | 296 | 0 |
| 13 | fwd/attn | 1.36 | 0.4648 | 0.3632 | 3.4843 | 0.0054 | 4.78125 | 43.75 | 48.5 | 0.0198 |
| | fwd/ffn | 37.75 | 1.4609 | 0 | 7.8437 | 0.3105 | 18.75 | 80.5 | 976 | 0 |
| 14 | fwd/attn | 2.58 | 0.4102 | 1.0078 | 7.25 | 0.2617 | 4.96875 | 93.5 | 28.5 | 0 |
| | fwd/ffn | 36.5 | 0.0232 | 0.9961 | 5.8125 | 0.21875 | 79.5 | 73.5 | 432 | 0 |
| 15 | fwd/attn | 2.25 | 0.0928 | 0 | 7.2813 | 0.1445 | 12.5625 | 47.25 | 49.25 | 0 |
| | fwd/ffn | 97.5 | 0.0322 | 0 | 1.6953 | 0.5 | 23.375 | 31.625 | 366 | 0 |
| 16 | fwd/attn | 11.94 | 0.9023 | 1 | 4.625 | 0.0971 | 119 | 40.75 | 103 | 0.4707 |
| | fwd/ffn | 39.75 | 0.0273 | 0 | 1.6641 | 0.1035 | 115 | 45.75 | 536 | 0 |

Table 8: Severity of SDCs in transformer primitive forwards, separated down by decoder layer and maximized over all microsteps. The unhealthy Nodes 2, 3, 5, 12, and 13 and 15 did not exhibit any mismatching tensors in forward passes in this experimental setting and thus are excluded from the table.

| | | SDC Node ID | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Decoder Layer | Primitive | Node 1 | Node 4 | Node 5 | Node 6 | Node 7 | Node 8 | Node 9 | Node 10 | Node 11 | Node 13 | Node 14 | Node 15 |
| 16 | bwd/ffn | 0.7383 | 0.0942 | 0.0376 | 0 | 0.1367 | 0.1143 | $9.621e{+12}$ | 0.1436 | 236 | 0.0588 | 0.0757 | 0.0053 |
| | bwd/attn | 1256 | 0.3223 | 0 | 0 | 2.0938 | 3.3281 | $3.758e{+11}$ | 1.1094 | 13.1875 | 0 | 0.0344 | 0 |
| 15 | bwd/ffn | 0.9023 | 0.2598 | 0.0206 | 0 | 0.1406 | 0.7266 | $8.934e{+12}$ | 0.2129 | 127 | 0.2344 | 0.0564 | 0 |
| | bwd/attn | 968 | 0.1885 | 0 | 0 | 1.3516 | 0.4648 | $3.737e{+11}$ | 0.3828 | 8.4375 | 0 | 0 | 0 |
| 14 | bwd/ffn | 1.5156 | 0.3359 | 0.0223 | 0.0267 | 0.0972 | 0.8008 | $6.769e{+12}$ | 0.4805 | 692 | 0.3164 | 0.1245 | 0 |
| | bwd/attn | 1040 | 1.1641 | 0 | 0 | 1.1563 | 6.4063 | $1.154e{+11}$ | 0.3125 | 39.75 | 0 | 0.0165 | 0 |
| 13 | bwd/ffn | 3.8750 | 0.5117 | 0.0679 | 0.0371 | 0.2676 | 0.1338 | $8.212e{+12}$ | 0.1563 | 352 | 0.0674 | 0.0596 | 0.0176 |
| | bwd/attn | 7392 | 2.0313 | 0 | 0.0469 | 0.7070 | 0.1318 | $1.943e{+11}$ | 0.3145 | 9.3125 | 0 | 0.0791 | 0 |
| 12 | bwd/ffn | 2.7656 | 0.3398 | 0.0908 | 0.0309 | 0.2969 | 0.1914 | $5.429e{+12}$ | 0.2559 | 1800 | 0.1768 | 0.0608 | 0 |
| | bwd/attn | 1640 | 0.3633 | 0 | 0 | 0.9961 | 4.0000 | $2.212e{+11}$ | 0.3379 | 68.5 | 0 | 0.0075 | 0 |
| 11 | bwd/ffn | 0.6641 | 0.4434 | 0.0310 | 0.0525 | 0.2061 | 0.1250 | $8.624e{+12}$ | 0.1631 | 7680 | 0.1299 | 0.0781 | 0 |
| | bwd/attn | 1992 | 0.2852 | 0 | 0 | 1.3516 | 0.1143 | $1.750e{+11}$ | 0.5625 | 2208 | 0 | 0.0952 | 0 |
| 10 | bwd/ffn | 0.7852 | 1.0078 | 0.0078 | 0 | 0.1709 | 1.3359 | $5.601e{+12}$ | 0.1924 | 332 | 0.0571 | 0.1494 | 0 |
| | bwd/attn | 3488 | 1.5000 | 0 | 0.0625 | 1.6406 | 0.2793 | $1.299e{+11}$ | 0.8828 | 55.25 | 0 | 0.0272 | 0 |
| 9 | bwd/ffn | 0.6914 | 0.1025 | 0.0209 | 0 | 0.0977 | 1.5938 | $4.948e{+12}$ | 0.3184 | 1776 | 0.0449 | 0.0562 | 0 |
| | bwd/attn | 4672 | 1.1484 | 0 | 0 | 0.8086 | 0.6719 | $2.835e{+11}$ | 0.2539 | 77.5 | 0 | 0.0239 | 0 |
| 8 | bwd/ffn | 0.6445 | 0.1289 | 0.0164 | 0 | 0.1196 | 0.1816 | $3.522e{+12}$ | 0.6367 | 256 | 0.0718 | 0.1367 | 0 |
| | bwd/attn | 864 | 1.8359 | 0 | 0 | 1.1172 | 1.5469 | $9.073e{+10}$ | 0.7617 | 29.5 | 0 | 0.0124 | 0 |
| 7 | bwd/ffn | 0.6406 | 0.2539 | 0.0679 | 0 | 0.1021 | 0.1040 | $4.364e{+12}$ | 0.2119 | 1880 | 0.1045 | 0.1465 | 0 |
| | bwd/attn | 748 | 0.3027 | 0 | 0.1621 | 7.6563 | 0.3359 | $7.999e{+10}$ | 4.0000 | 104.5 | 0 | 0.0266 | 0 |
| 6 | bwd/ffn | 0.4434 | 0.1826 | 0.0320 | 0.0322 | 0.0835 | 0.5000 | $2.749e{+12}$ | 0.2656 | 173 | 0.1172 | 0.1084 | 0.0082 |
| | bwd/attn | 3248 | 0.8242 | 0 | 0 | 5.3438 | 0.9102 | $7.892e{+10}$ | 12.7500 | 76 | 0 | 0.0223 | 0 |
| 5 | bwd/ffn | 0.3320 | 0.6172 | 0.0080 | 0 | 0.1533 | 0.1040 | $2.233e{+12}$ | 0.6484 | 1016 | 0.0742 | 0.3789 | 0 |
| | bwd/attn | 664 | 0.5469 | 0 | 0 | 6.4063 | 0.6875 | $7.087e{+10}$ | 0.9492 | 190 | 0 | 0.3828 | 0 |
| 4 | bwd/ffn | 0.4883 | 0.2852 | 0.0082 | 0 | 0.1074 | 0.1738 | $1.563e{+12}$ | 0.2246 | 334 | 0.0664 | 0.0781 | 0 |
| | bwd/attn | 1784 | 0.1816 | 0 | 0.0356 | 3.6094 | 0.1895 | $8.536e{+10}$ | 3.3750 | 304 | 0 | 3.7969 | 0 |
| 3 | bwd/ffn | 0.4043 | 5.3750 | 0.0093 | 0 | 0.1729 | 0.1064 | $1.623e{+12}$ | 0.2178 | 556 | 0.0879 | 0.2295 | 0 |
| | bwd/attn | 1020 | 2.3906 | 0.0048 | 0 | 1.6172 | 0.5000 | $9.342e{+10}$ | 1.0859 | 74 | 0 | 1.0156 | 0 |
| 2 | bwd/ffn | 1.6328 | 0.8008 | 0 | 0 | 0.1387 | 0.0796 | $1.512e{+12}$ | 0.1416 | 836 | 0.1157 | 0.1128 | 0 |
| | bwd/attn | 644 | 0.3594 | 0 | 0.0537 | 0.9844 | 0.2080 | $8.966e{+10}$ | 1.3359 | 216 | 0 | 2.2500 | 0 |
| 1 | bwd/ffn | 0.6680 | 0.1152 | 0.0195 | 0.0776 | 0.1641 | 0.0801 | $1.044e{+12}$ | 0.2344 | 406 | 0.0356 | 0.1270 | 0 |
| | bwd/attn | 596 | 0.3535 | 0 | 0 | 32.2500 | 0.1436 | $1.463e{+10}$ | 0.2676 | 241 | 0 | 0.2656 | 0 |

Table 9: Severity of SDCs in transformer primitive backwards, separated down by decoder layer and maximized over all microsteps. The unhealthy Nodes 2, 3, and 12 did not exhibit any mismatching tensors in forward passes in this experimental setting and thus are excluded from the table.