Reliable Numeric Fault Detection During Edge TPU Inference

Md. Azahar Alam
Electrical Engineering and Computer Science
The University of Toledo
Toledo, OH, USA
Md.Azahar.Alam@rockets.utoledo.edu

Kishwar Ahmed
Electrical Engineering and Computer Science
The University of Toledo
Toledo, OH, USA
Kishwar.Ahmed@utoledo.edu

***Abstract*—Detecting floating-point exceptions during Edge TPU inference is structurally hard: the accelerator has no floating-point unit, and attaching it fuses the mapped partition into one opaque operation whose values never leave the chip, so tensor inspection is impossible there. We instead read the host CPU's floating-point status register before and after each inference, which every floating-point operation in the pipeline sets because all of them execute on the host. Across five stock models, register polling detected 100% of exception-raising faults, tensor scanning 98.8%, and output inspection 24.6%. Polling adds under 1% to latency; the optional tensor scan costs 7% to 58%. Detection holds on the accelerator, where 90% to 99% of tensors become unobservable. The tool also counts `int8` saturation, targeting the 72% of output-corrupting faults that raise no exception.**

***Index Terms*—edge accelerators, quantized inference, IEEE 754, fault injection, numerical reliability, Edge TPU**

I. Introduction

Quantized neural networks increasingly run on edge accelerators in safety- and cost-sensitive deployments, where a silent numerical error can be worse than a crash [1], [2]. On general-purpose processors, IEEE 754 [3] exceptions provide a hardware-level signal that existing detectors can monitor [4]. The Coral Edge TPU provides no equivalent signal: it executes `int8` operations, where out-of-range accumulations are clamped, and execution continues silently. Moreover, its closed compiler and binary `libedgetpu` [5] prevent conventional instrumentation, while delegation hides 90–99% of a model's tensors behind a single custom operation.

We address this observability gap by monitoring host processor state. Because the Edge TPU has no floating-point unit, every floating-point operation an inference performs (dequantization and post-processing such as NMS) executes on the host CPU inside the same `Invoke()` call, and it is these host operations, not the accelerator, that set the host FPU status register. Clearing that register before `Invoke()` and reading it afterward therefore captures every floating-point exception raised during inference without modifying the compiler, runtime, or model. To detect silent `int8` failures that raise no exception, we additionally count activations pinned at quantization clamp bounds (saturation counting). This paper contributes (i) this register-polling technique for delegated Edge TPU inference, (ii) a fault injector and a five-model evaluation of fault reachability, observability, and detection on both the CPU path and the accelerator, and (iii) an open-source artifact with the tool, injector, and all campaign outputs.

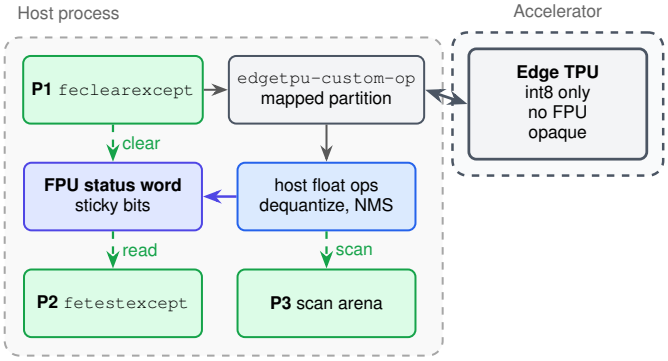


Fig. 1. Probe placement. P1 and P2 bracket `Invoke()`; P3 reads the tensor arena.

II. Methodology

A. Instrumentation

The tool observes processor state and memory that the unmodified runtime already produces. Two probes bracket a single `Invoke()` and a third reads the tensor arena afterward, so nothing is hooked or patched. Status-register polling detects IEEE exceptions on the host floating-point surface: the register holds one sticky bit per exception class, set by any host floating-point instruction and left set until cleared, so `feclearexcept` before `Invoke()` and `fetestexcept` after it report every class raised in between. Tensor scanning reads each `float32` tensor in the arena after `Invoke()` and flags NaN, $\pm\infty$, and subnormal elements, localizing an exception to individual tensors. Saturation counting covers the quantized surface the accelerator executes, where no exception is possible: it counts `int8` activations pinned at a clamp bound (the clamping described in §I). Alg. 1 gives the procedure, and Fig. 1 places the probes relative to the partition boundary.

Algorithm 1 One instrumented inference

Require: model m, input x, interpreter with `preserve_all_tensors`
Ensure: flags f, anomalies A, saturation S

SetInput(x)
FeClearExcept(ALL) ▷ no arithmetic below
Invoke ▷ unmodified
$f \leftarrow$ FeTestExcept(ALL)
$A, S \leftarrow \emptyset, \emptyset$
for all activations t of m **do** ▷ constants skipped
  if t is `float32` **then**
    $A \leftarrow A \cup \{i : t_i \in \{\text{NAN}, \pm\infty\} \vee |t_i| < \tau\}$
  else
    $S \leftarrow S \cup \{i : t_i \in \{lo, hi\} \wedge t_i \neq zp(t)\}$
return f, A, S

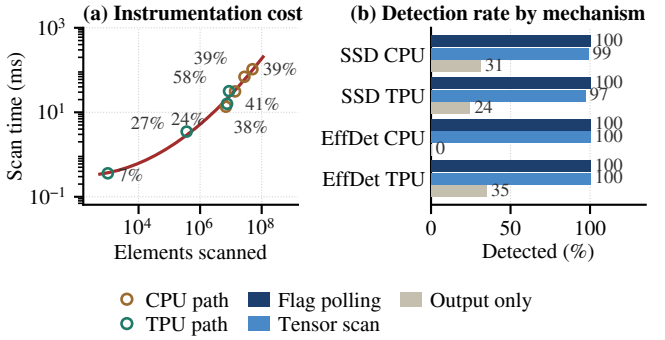


Fig. 2. (a) Scan cost against elements observed. (b) Detection rate by mechanism, both execution paths.

TABLE I
IEEE 754 COVERAGE. TARGETED PROTOCOL, BOTH PATHS, 338 TRIALS; A TRIAL MAY RAISE SEVERAL CLASSES.

Exception	Reachable via	N	Det.
invalid	NaN scale; Inf arithmetic in NMS	332	332
overflow	`exp()` after dequantization	40	40
underflow	scale attenuation at boundary	44	44
div-by-zero	unbounded divisor; none in 5 models	0	—
inexact	every float operation	—	excl.

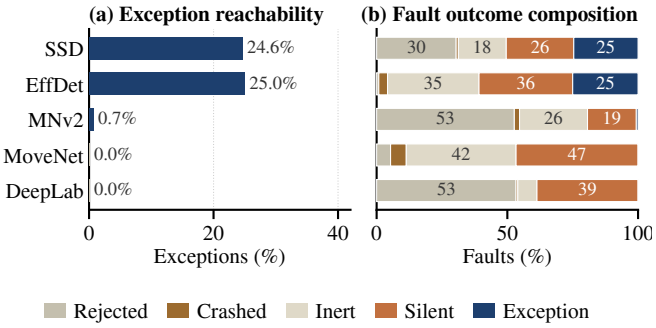


Fig. 3. (a) Exception reachability per model. (b) Fault outcome composition, uniform sampling.

B. Fault Injection

Every Coral model takes integer input and Python cannot write intermediate tensors, so faults are introduced into the model file [6]–[9]. Three kinds of sites are modified. Float constants become NaN, zero, or Inf. Integer constants move to a clamp bound. Quantization metadata is perturbed by rescaling, by setting the scale to NaN or zero, or by shifting the zero point. Each trial runs in a child process, since malformed metadata can abort the interpreter. Outcomes are recorded in five classes: rejected at prepare time, crashed, inert, output corrupted with no exception, and exception raised. Detection rate is computed over the last class only. Sampling is either uniform over all metadata sites or targeted at the dequantization inputs feeding post-processing.

III. Evaluation

A. Detection and Overhead

Under the targeted protocol, flag polling detected 100% of exception-raising faults on both paths, 338 of 338. Tensor scanning detected 97–100%, and inspecting model outputs for NaN and Inf detected 0–35% (Fig. 2b). Of the 338 exceptions raised, 228 (67%) were absorbed before reaching any output tensor by non-maximum suppression filtering and score thresholding. These rates are defined over trials that raise an exception. The integer-constant faults of §II-B never do: saturating an `int8` activation at its clamp bound is ordinary quantized arithmetic, so the model runs to completion and returns a wrong answer, as it did in 27 of 30 trials on DeepLabV3, 25 on EfficientDet, 21 on SSD, 17 on MoveNet, and 1 on MobileNetV2. No floating-point mechanism can see these faults; saturation counting is the only one that can. Polling is two `libm` register operations per inference and is not separately measurable, while tensor scanning costs 7–58% of inference latency, scaling linearly in the number of elements observed at ≈0.5M elements/ms (Fig. 2a).

Table I summarizes coverage of the IEEE 754 exception set. Three of the four meaningful classes were demonstrated with full detection. Division by zero was not reachable in any of the five models: it requires an unbounded divisor, and quantized models in our set do not produce one.

B. Platform Characterization

The campaign also characterized the accelerator itself. Exceptions are reachable where an unbounded amplifying operation follows dequantization: MoveNet contains `DIV` and `SQRT` yet raised none in 150 trials; its divisor is bounded below at 1.747 (Fig. 3a). The one case outside this pattern was a corrupted scale reaching TFLite's own arithmetic inside a `SOFTMAX` kernel on MobileNetV2, a model with no float tensors, where tensor scanning could not respond at all. Most faults are therefore silent: 72% of those that corrupted an output raised no exception (Fig. 3b). Delegation removes 63% to 98% of quantized activations but no floating-point activation, which is why detection holds on both paths (Fig. 4a).

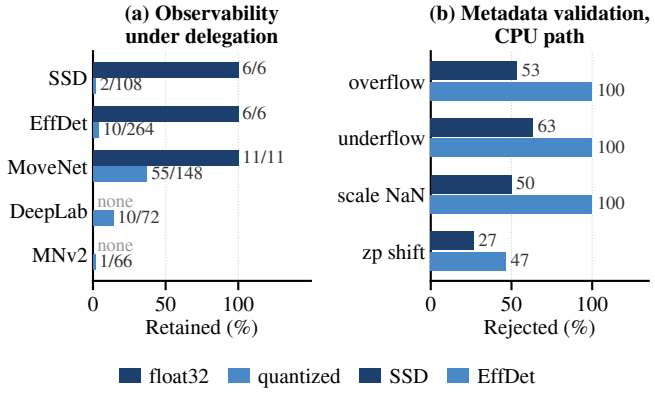


Fig. 4. (a) Activation retention under delegation; DeepLabV3 and MobileNetV2 have no float32 activations. (b) Injections rejected on the CPU path.

Compilation also disables TFLite's quantization checks, since the fused partition never runs the kernels that perform them: injections rejected 8 to 30 times out of 30 on the CPU path execute unchecked on the device (Fig. 4b).

C. Limitations

Our tool reports which exception fired, not which operation raised it. Per-operation attribution needs a C++ kernel wrapper.

IV. Conclusion

This paper presented a runtime technique for detecting numeric faults during Coral Edge TPU inference. Because the accelerator cannot execute floating-point arithmetic, every IEEE 754 exception is raised on the host CPU, and polling its status register captures all of them without modifying the compiler, runtime, or model. On five stock models, polling detected every exception-raising fault at under 1% latency cost.

References

[1] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proc. IEEE/CVF Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2018, pp. 2704–2713.

[2] R. Krishnamoorthi, "Quantizing deep convolutional networks for efficient inference: A whitepaper," *arXiv preprint arXiv:1806.08342*, 2018.

[3] "IEEE standard for floating-point arithmetic," IEEE Std 754-2019 (Revision of IEEE 754-2008), 2019.

[4] I. Laguna, "FPChecker: Detecting floating-point exceptions in GPU applications," in *Proc. 34th IEEE/ACM Int. Conf. on Automated Software Engineering (ASE)*, 2019.

[5] Google Coral, "Edge TPU compiler," https://coral.ai/docs/edgetpu/compiler/, 2026, accessed: Aug. 2026.

[6] G. Li, S. K. S. Hari, M. Sullivan, T. Tsai, K. Pattabiraman, J. Emer, and S. W. Keckler, "Understanding error propagation in deep learning neural network (DNN) accelerators and applications," in *Proc. Int. Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*, 2017.

[7] B. Reagen, U. Gupta, L. Pentecost, P. Whatmough, S. K. Lee, N. Mulholland, D. Brooks, and G.-Y. Wei, "Ares: A framework for quantifying the resilience of deep neural networks," in *Proc. 55th ACM/ESDA/IEEE Design Automation Conf. (DAC)*, 2018.

[8] Z. Chen, G. Li, K. Pattabiraman, and N. DeBardeleben, "TensorFI: A flexible fault injection framework for TensorFlow applications," in *Proc. IEEE 31st Int. Symp. on Software Reliability Engineering (ISSRE)*, 2020.

[9] A. Mahmoud, N. Aggarwal, A. Nobbe, J. R. S. Vicarte, S. V. Adve, C. W. Fletcher, I. Frosio, and S. K. S. Hari, "PyTorchFI: A runtime perturbation tool for DNNs," in *Proc. 50th Annual IEEE/IFIP Int. Conf. on Dependable Systems and Networks Workshops (DSN-W)*, 2020.

Appendix A
Artifact Description

Artifact. Tool, injection harness, and all campaign outputs at https://github.com/azaharalam/edgetpu-fpe.

Environment. x86_64 host with a Coral USB Accelerator, Python 3.9, `tflite-runtime` 2.5.0.post1, `pycoral` 2.0.0, NumPy 1.26.4, and Edge TPU Compiler 16.0.384591198.

Reproduction. `check_env.py` verifies the setup, `fetch_models.sh` and `make_inputs.py` prepare models and inputs, `smoke_test.sh` exercises every stage on the CPU path in about two minutes, and `run_all.sh` runs the CPU-path campaigns; the accelerator campaigns and the figure scripts (`make_paper_figures.py`, `make_poster_figure.py`) are listed in the repository README.

Variance. Injection sites are not fully seeded, so per-cell counts vary slightly between runs.