Automated Root-Cause Analysis of GPU Pipeline Corruptions in Graphics and Compute Workloads

Pravesh Dangwal pravesh.dangwal@intel.com, Himanshu Somaiya himanshu.somaiya@intel.com

Abstract: Data corruption in graphics and compute pipelines presents one of the most difficult challenges in GPU design verification and validation. Such issues may appear as pixel mismatches, visual artifacts, or incorrect compute results, and can originate from hardware logic flaws, race conditions, or driver-level software bugs. Traditional debug methods rely on extensive manual analysis and waveform inspection, often requiring several days of engineering effort to isolate the root cause. This paper introduces an internal automated corruption root-cause analysis framework that accelerates debug across RTL simulation, emulation, and post-silicon environments. The tool is proprietary and not publicly available. It leverages a golden reference run to perform systematic comparisons at RTL unit interfaces, identifying and localizing data mismatches. It then traces data dependencies across the pipeline to pinpoint the exact transaction, hardware block, or software programming issue responsible for the corruption. The proposed methodology provides a scalable and platform-independent debug solution, significantly reducing debug turnaround time and human effort. Case studies from real GPU workloads demonstrate precise and consistent identification of corruption sources, improving both functional correctness and design reliability during GPU validation.

Introduction and Motivation

Modern Graphics Processing Units (GPUs) are among the most complex computing architectures in existence, powering high-performance graphics in gaming as well as large-scale compute workloads for AI and machine learning. The rendering process or GPU based general purpose compute is a highly intricate collaboration between GPU hardware and Graphics driver software, involving a complex chain of interactions between programmable shaders, fixed-function units, and software-managed synchronization mechanisms.

A single GPU integrates hundreds of tightly coupled hardware intellectual property (IP) blocks, including geometry, rasterization, compute, copy, and memory subsystems, operating concurrently across multiple pipelines. These pipelines - 3D, compute, and copy, execute distinct tasks but frequently share resources and depend on each other's outputs, introducing intricate interdependencies.

To maintain correctness, GPU operation relies on explicit synchronization mechanisms such as barriers, semaphores, and fences to coordinate concurrent execution and avoid race conditions. Additionally, shared resources like textures and buffers must transition between valid usage states (e.g., from copy destination to shader resource) as they move across pipelines. Any misconfiguration, such as a missing barrier, or invalid resource state transition, incorrect cache coherency policy, can lead to subtle and intermittent data corruption.

A single game frame or AI kernel may produce terabytes of trace data, making manual inspection infeasible. Identifying the root cause of corruption, whether stemming from a hardware logic bug, memory error, or driver-level software bug, involves deep expertise in GPU microarchitecture, driver internals, and advanced low-level tracing techniques, making the process slow, costly, and error prone.

The key challenges in GPU corruption debug include:

- Pipeline complexity: Hundreds of interacting IP blocks with shared caches and global memory create numerous failure points.
- Data volume: Massive amounts of interface and trace data make anomaly isolation time-consuming.
- Model mismatch: Behavioral differences between golden reference models (e.g., C++) and RTL implementations complicate direct comparison.
- Workload diversity: Debug methods must be effective for both graphic pipelines (requiring pixel-level accuracy) and compute pipelines (requiring precise numerical validation).

Traditional interface checkers and manual waveform inspection are inefficient and unsustainable for debugging such large, parallel systems. These challenges underscore the need for a systematic and automated debug framework capable of tracing data corruption across the full GPU pipeline, from software API commands to low-level RTL transactions.

Our Approach

We introduce ARGO (**A**utomated **R**oot-Cause for **G**PU pipeline c**O**rruptions), an automated debug framework implemented in Python. ARGO is designed to:

1. Automate corruption isolation by identifying the exact RTL interface and transaction responsible for a failure.
2. Support multiple environments including pre-silicon simulation, pre-silicon graphics driver-based Emulator/FPGA platform runs, and post-silicon validation employing capture replay on emulator.
3. Target both workload classes: graphics (pixel level tracking) and compute (memory-based analysis).
4. Handle mismatched models by incorporating GPU microarchitectural knowledge to reconcile golden reference vs. RTL datasets.
5. Improve efficiency by restricting analysis only to transactions directly contributing to corrupted data, avoiding terabyte-scale overhead.

Methodology

ARGO implements an interface decision tree that mirrors the GPU pipeline's datapath. Each pipeline interface corresponds to a finite state machine (FSM) state within the tool.

- **Data Collection:** ARGO ingests interface trackers from both passing and failing runs. Passing datasets may be derived from a C++ golden model or a verified RTL run, while failing datasets are obtained from the faulty RTL model.
- **Comparison Engine:** At each interface, ARGO compares inputs and outputs across passing and failing runs. Mismatches identify the corrupted interface.
- **Microarchitectural Awareness:** Differences in implementation between golden and RTL runs are normalized using GPU-specific microarchitectural rules, ensuring robust comparison.

Finite-State Framework and Modes of Operation

At start, ARGO enters the FSMInitState (Figure 1) where it determines the mode of operation. The tool supports three modes: Pixel Mode, Address Mode, and Compute Mode, corresponding to different testbench types and workload categories. Although they differ in entry conditions, they all follow a unified FSM framework that traverses interface-level states to backtrace the corruption source.

Pixel Mode

In Pixel Mode, user provides the {X,Y} screen space coordinates where the corruption is observed when comparing the images generated by failing and the golden run. ARGO collects and compares all the writes that occur to this pixel in the PixelFrontendState. This is the front end of the output merger stage and the pixel writes are in order according to actual rendering at this stage of the pipeline. If there is no mismatch in the pixel color values at PixelFrontEndState, tool will grab the address that the pixel updates and pass the information to PixelBackendState. At PixelBackendState the tool collects and compares all the transactions for the address provided. If the data for the address mismatches, the tool passes that address to PixelBlendRdState; else, if there is no mismatch, it passes the address to L2CacheState. In PixelBlendRdState for the mismatching write the tool checks if there was any blend read done and whether the data for blend read matches. If there is no blend read, or the blend read matches and Pixel backend output mismatches then ARGO exits providing the user the transaction information and points to debug further in pixel backend units. If the blend read mismatches, the tool compares the previous writes to this cacheline in the color cache. If previous writes match and read data by color cache from L2 also matches, then ARGO exits with transaction information and indicates that the issue lies in the color cache. If read data from L2 mismatches, then ARGO passes the address to L2CacheState.

Execution-Unit and Shared-Function Correlation

At PixelFrontEndState, if the data for the pixel mismatches, the tool will grab the thread[1] identifier and pass it to the EUOutputState_Init (Figure 2). In this state, the tool looks at the output of Execution Units for the given message[2] of the identified thread. If the Execution output mismatches for the message of the given thread, then the tool moves to check the thread dispatch payload in the ThreadDispatchState. If there is no mismatch in the thread dispatch payload then tool moves to EUOutputState where it starts checking from first message of the same thread leading upto the message count of the initial problematic message. In this EUOutputState ARGO compares the output of the execution unit for the message identifier 0 (first message of the shader program), if output is good then it figures which shared function is the message targeted to. If it is a texture sample message, then tool moves to TextureEUState. Here it figures if the texture unit is returning wrong sampled value to Execution unit. If the return sampled data is matching, the tool increments the message count and moves to next message. But if the texture to EU output is mismatching,

then tool moves to L2TextureState to figure out if texture unit received correct texel values from memory/L2. If the L2 reads to texture unit match, then ARGO exits giving the exact thread and points to debug further in texture unit.

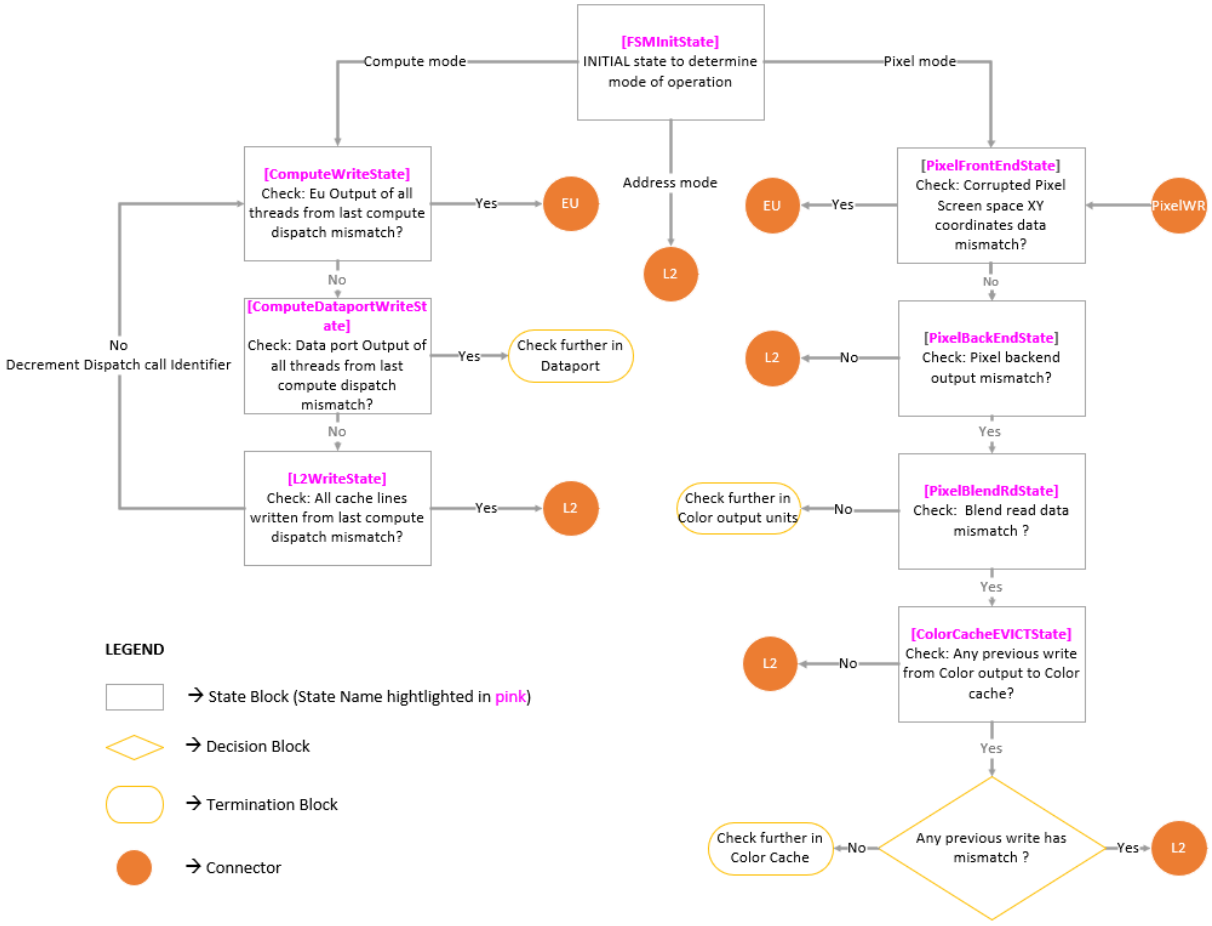

Figure 1.

A similar operation is done for other shared functions like Dataport (handles untyped and typed loads/stores), Ray Tracing. ARGO keeps on incrementing the message count of the thread till it identifies a message which is giving mismatching return to the Execution unit. If at any message, output of Execution unit towards shared function is mismatching, then tool exits pointing to check further in the EU program. If a shared function returns a mismatching data to Execution unit, then ARGO moves to L2<SharedFunction>State to determine if L2/memory gave mismatching data to the shared function. If L2 gave the correct data then ARGO exits pointing to issue in shared function and provides the thread and message information. Shared function like texture, dataport and Ray Tracing contain L1 caches which requires special handling for cases when there are hits in L1 vs miss in comparing run. Shared Local Memory[3] (SLM) is another shared function which is not backed by L2 so it is treated differently by ARGO. If SLM response to EU mismatches, then ARGO moves to SLMWriteState to determine which thread of the Threadblock/Threadgroup updated the address that current message is reading. If the SLM write matches then ARGO points to debug further internal to SLM. If write mismatches then ARGO moves to EUOutputState_Init for the new writing thread identifier.

At EuOutputState_Init (Figure 2) if the output matches, then tool goes to DataportState with the thread and message identifier. In DataportState tool decodes the incoming message to find if there is any control surface state associated with the message. If the message is stateful, then it compares the control state data and exits if state data does not match pointing to programming issue in the cotrol surface. If the control state matches then tool compares the output of DataportState. If it is a render target write message and dataport output is matching then tool exits pointing issue in color pipe. If the dataport output is mismatching then regardless of message type tool points to issue in dataport.

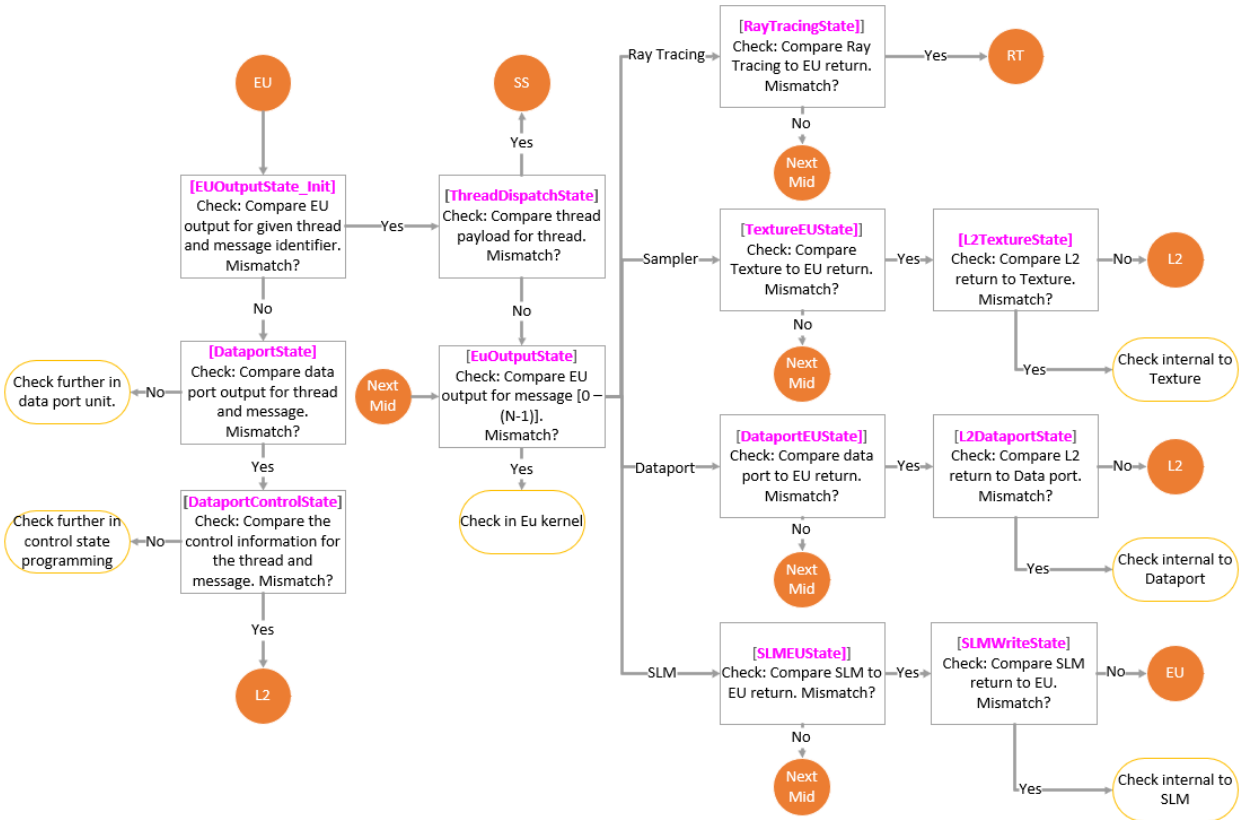

Figure 2.

L2 Cache Correlation and Client Resolution

In cases where L2 is returning mismatching data, ARGO moves to L2CacheState (Figure 3). The input to L2Cachestate is the address for which the return to the client was mismatching. L2 can operate in virtual or physical address domain. If L2 is in physical space, then an extra step of virtual to physical transalation is required. The goal of L2CacheState is to determine the previous client who wrote to the same address. If the previous write does not match, then ARGO moves to the client responsible for the mismatching write in L2. There are multiple clients which can write to L2. Below is the list of clients that can perform writes and the specific action taken by the tool for each client

- **Dataport**: Move to DataportWriteState and determine the thread identifier resulting in the mismatching write. Move to EUOutputState_Init with the thread identifier information.
- **Command Streamer**: Find the command buffer instruction resulting in the mismatching write. Tool exits providing information on the instruction.
- **Copy Engine**: Move to CopyEngineState and determine the copy call whose destination is the address resulting in mismatching write. For this call determine the source address and compare the source read data. If source read data mismatches then move to L2CacheState, else tool exits pointing to check further inside copy engine.
- **Color**: Move to PixelFrontEndState to determine which thread is doing this render target write resulting in bad data in L2. In PixelfrontEndState, if mismatch is observed then move to EUOutputState_Init else tool moves to PixelBackEndState.
- **Ray Tracing**: Move to RayTracingState, find the ray which is generating the write, compare the Bounding Volume Hierarchy[4] (BVH) read data for the ray. If the BVH read data mismatches then move to L2CacheState with the BVH read address. If the BVH read data matches then tool exits pointing to debug further in ray tracing where possible issues could be in ray traversal or color accumulation for the ray.
- **Depth**: Move to DepthWriteState. Find the pixel and polygon for the mismtaching depth write and move to EarlyDepthState.
- **Streamout**: Move to StreamoutState. Find the input from Unified Return Buffer[5] (URB) to the streamout engine resulting in mismatching write to L2. If URB read matches tool exits pointing to debug internal to streamout engine, otherwise move to URBWriteState with URB addresses as input.

- **All previous write to L2 match**: In this case since the return to client from L2 is mismatching but all previous writes to L2 are matching so the tool exits pointing to debug further in L2 cache unit.
- **No previous write**: Since none of the internal Graphics client has written to L2 for this address, the data in L2 is the result on intial memory load and thus tool exits pointing to check further on the input data to graphics pipeline.
- **Stale Data**: In this case, a missing write is observed before the L2 is read. There are two scenarios possible
    - The missing write occurs after the L2 read, representing a Read-After-Write (RAW) hazard. In this scenario, the tool analyzes the passing run to identify the client responsible for the write and determines that the issue stems from a missing synchronization in the graphics driver's software programming. This condition reflects a producer-consumer relationship violation, where the producer (the stage performing the write) has not properly synchronized with the consumer (the stage performing the read). For example, if a texture read in the failing run mismatches with the corresponding depth stage write from the passing case, the tool flags a missing depth flush as the root cause, indicating that the data produced by the depth stage was not made visible to the texture stage before it was consumed.
    - The missing write never happens. In this case, tool figures the client performing the write in passing case and moves to the corresponding client state. Now here tool compares the transactions for the entire draw call or the dispatch call coming into the client and moves appropriately to the previous state based on mismatch obsereved or not.

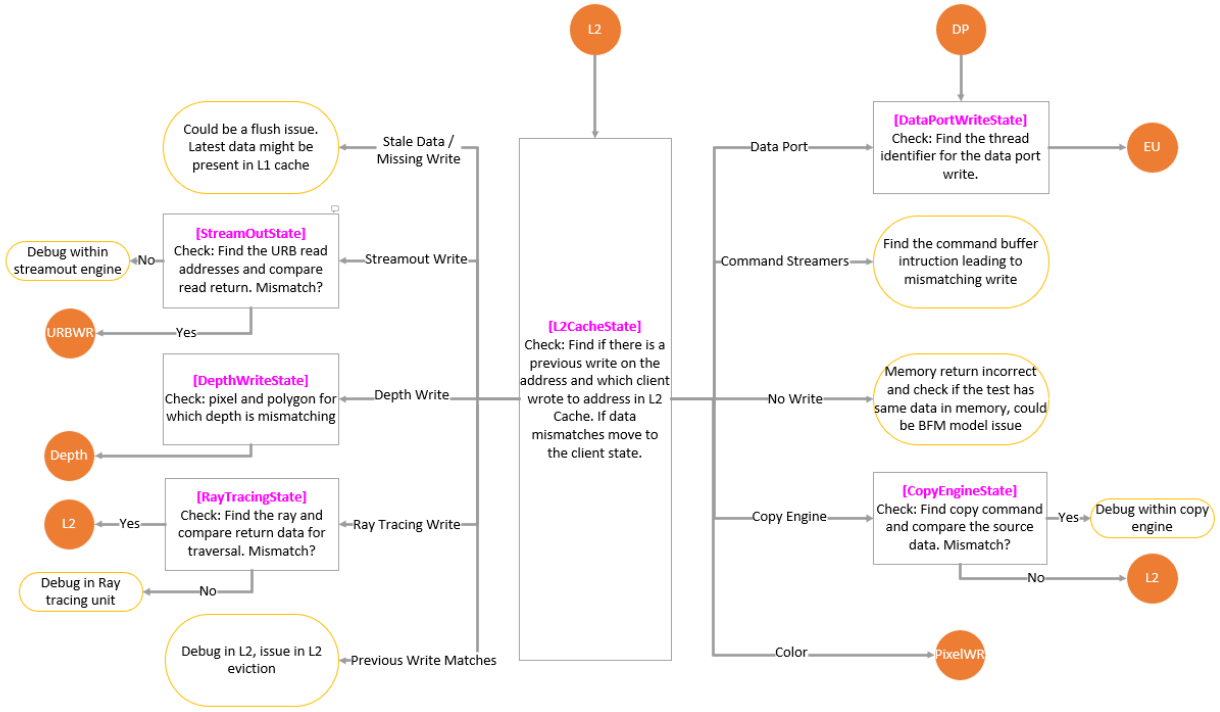

Figure 3.

Geometry and Raster Correlation

In the cases where thread dispatch payload mismatches at the ThreadDispatchState (Figure 2), tool moves to ShaderState (SS in diagram, Figure 4). Here ARGO figures which shader is the dispatch mismatching for.

In case of Geometry based shaders (Vertex Shader, Hull Shader, Domain Shader, Geometry Shader, Amplification Shader, Mesh Shader) tool moves to URBReadState where it finds the URB addresses that thread is reading using the thread identifier from dispatch state. If the URB read data matches, then tool exits pointing to check internal to Thread Dispatch Unit. If the URB read data mismatches, then tool moves to URBWriteState. Here tool finds which client is performing the latest write to the URB read address. If the latest write matches, then tool exits pointing to check further in URB unit. If the write mismatches, then tool moves to different state based on the client.

- If the client is dataport, then it is one of the previous shaders which wrote to the address and tool moves to DataportWriteState.
- If the client is vertex fetch, then tool moves to VertexFetchState. This is the input assembler stage of the pipeline and here the Vertex fetch unit reads data from memory from the programmed vertex buffer location and writes it to the URB. Tool compares the read data from memory and if it matches then tool exits pointing to issue in vertex fetch unit. If read data mismatches, then tool moves to the L2CacheState to find if any client wrote to this address.
- If the client is constant data unit, then tool moves to ConstantDataWrState. Here tool figures the constant buffer programming for the shader and draw call for which the URB read was mismatching. Then tool finds the read data for the constant buffer from memmory and if any mismatch is observed on read reaturn then tool moves to L2CacheState. If the read return data matches then tool exits pointing to check further in constant data unit.
- If the client is tessellator, then tool figures the input addresses read by tesselation fix function to get the tesselation factors and patch control points generated by Hull shader. If the input read data matches then tool exits saying issue in tessellation fix function, otherwise tool moves to URBWriteState to figure the hull shader output for any mismatch.

In case of Pixel shader, tool moves to PixelShaderState. Here the tool determines which phase of the dispatch is mismatching. There are 3 phases - Barycentric data, attribute data, constant data. Tool follows different paths for mismatch in each of these phases as below

- **Barycentric data mismatch**: Tool moves to BaryCentricState. Barycentric coordinates are used in the GPU's rasterization phase to interpolate vertex attributes, such as color and texture coordinates, across the surface of a triangle. Here, tool determines which pixels are being drawn for the mismatching thread payload and checks if the input from early depth for those pixels are matching. If the data matches, tool exits pointing to issue in barycentric unit. If the earlydepth data mismatches then tool moves to EarlyDepthState. The depth stage performs a depth test using the depth buffer to determine which pixels are closest to the camera, ensuring that farther objects are correctly occluded by nearer ones. This depth test basically compares the source depth value to the destination depth value for the given pixel. In EarlyDepthState the input from rasterizer is checked for the pixels and the polygon generating these pixels. If the data from rasterizer matches, then tool moves to SourceDepthState where tool determines the source depth data for these pixels. The source depth data could be residing in depth L1 cache or a fetch from L2 cache. If the source depth data matches, then tool exits pointing to issue in depth calculation unit. If the source depth data mismatches then tool moves to L2CacheState to figure out which client wrote to this address earlier.
  If rasterizer output to early depth mismatches, then tool moves to RasterState. The function of rasterizer is to use the polygon data from triangle setup function and calculate the pixels which will be covered by the incoming polygon. If the incoming polygon data from triangle set up is matching then tool exits pointing to issue in rasterizer unit. If the incoming data mismatches then tool moves to TriangleSetupState. The function of triangle setup state is to provide data for efficient rasterization of the polygon (usually triangles) using the three vertices in screen space as provided by previous primitive assembly stage. In this state the tool checks the input objects from primitive assembly, if input matches then tool exits pointing to issue in triangle setup. If input mismatches then tool moves to ClipperState. In this state, primitve assembly and frustrum clipping functions are performed. The input to clipper state are the vertices shaded by previous geometry stages. In this state the tool figures the address of the URB where these three vertices are stored. Tool gets the read return data for the vertices and if they match then tool exits pointing to issue in clipper unit. If the vertex return data mismatches, then tool moves to URBWriteState.
- **Attribute data mismatch**: Tool moves to AttributeInterpolationState. In this state vertex attribute data is read and is interpolated across the covered fragments. The tool figures the address from where the attribute data for the vertices is stored and reads it for passing and failing cases. If the input attribute data matches, then tool exits pointing to issue in attribute interpolation calculation unit. If the per vertex attribute incoming data mismatches then tool moves to URBWriteState.
- **ConstantData mismatch**: Tool moves to ConstantDataRDState. Constants is the data which remains same for all threads launched by the shader and typical usage is to provide matrices used in tranformation or lighting parameters. Tool figures the constant address for the mismatching polygon and compares the read data. If the read data matches, then tool exits pointing to issue in constant unit. If the read data mismatches, then tool moves to L2CacheState to figure out which client wrote to this address earlier.

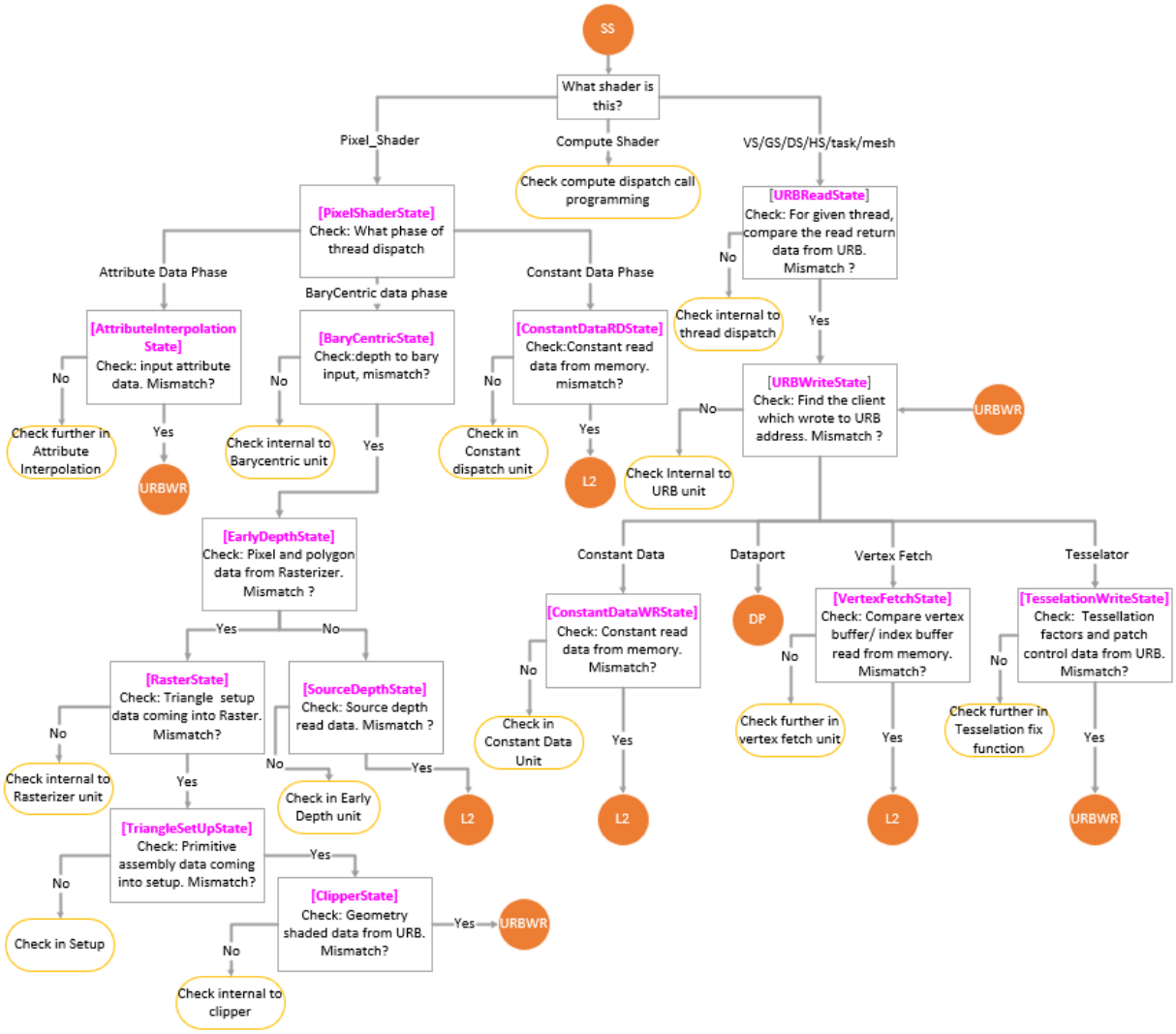

Figure 4.

Compute Mode and Address Mode

If the workload is purely compute specific like AI/ML workloads, then ARGO follows a different approach. For such cases, the tool begins at FSMinitState and then moves to ComputeWriteState (Figure 1). Here the tool identifies the last compute dispatch call and compares the output of execution units for all threads generated by this dispatch. If any mismatch is observed for a thread, then ARGO moves to EUOutputState_Init passing the thread and message identifier. If no mismatch is observed for any thread, then ARGO moves to ComputeDataportWriteState. Here tool compares all the write data for all the threads of last dispatch call. If mismatch is seen, then the tool exits pointing to check further in dataport unit. If there is no mismatch, then tool moves to L2WriteState passing all the addresses that the threads from last dispatch call were updating. Here tool compares the output of L2 eviction and if mismatch is seen for a comparable address, then tool moves to L2CacheState passing the address to figure out which client wrote to this address earlier. If there is no mismatch observed in L2WriteState then the tool decrements the dispatch call identifier moving to the previous dispatch and moves to ComputeWriteState and the loop follows until it finds the mismatch. In the address mode, the testbench provides an address where the final corruption is observed. In this case, ARGO moves from FSMinitState to L2CacheState to figure out which client wrote to this address and thus move appropriately to the defined states.

Results

Deployment of ARGO in production GPU debug environments has demonstrated:

- **100x – 1000x reduction** in debug turnaround time in simulation and emulation compared to manual debug.
  - Typically, a full end to end graphics pipeline corruption debug takes anywhere from 1 day to multiple weeks to root cause. ARGO can find the root cause quickly with the time taken ranging from a few seconds for pre-silicon simulation test cases to 2-3 hours in larger frames depending on the size of the test and tracker database. The test cases debugged using ARGO include large compute kernels, 3d benchmark frames like Time Spy, Microsoft API WHQL certification tests and pre-silicon synthetic tests.
- **Significant accuracy improvements** in root-cause localization and emulator capacity saving
  - Manual debugging relies heavily on the engineer's expertise; an inexperienced debugger often hops from one design team to another while following the pipeline stage by stage before finding the true root cause, consuming substantial engineering bandwidth. ARGO eliminates these unnecessary hops by pinpointing the exact root cause in a single step.
  - During these multiple debug hops, debuggers tend to take multiple emulation captures falsely assuming to have reached the final root cause. Emulation captures are very expensive, and ARGO helps save the emulator capacity by eliminating these unnecessary intermediate hops.
- **Efficient bug categorization**, preventing duplication of effort across teams.
  - In a typical regression which involves tens of thousands of test failing with corruptions, ARGO helps bucketize failures with similar root cause thus preventing significant duplication of effort. At the same time this also helps prioritize the bucket having maximum failures improving regression pass rate at a faster pace.

These results validate ARGO as an essential framework for scalable GPU debug across diverse workloads and environments.

Conclusion

Debugging GPU pipeline corruptions has been one of the most time-consuming and resource intensive aspects of GPU development. ARGO introduces a scalable, automated approach that combines architectural awareness, targeted analysis, and cross-platform applicability. Its ability to isolate corruption to exact RTL interfaces and transactions, while drastically reducing turnaround time, represents an advancement in GPU validation methodology.

ARGO not only accelerates debug but also enhances engineering efficiency, enabling teams to deliver higher-quality GPUs with faster time-to-market. Its design principles are broadly applicable to other large-scale SoC debug environments, making ARGO a valuable contribution to the field of hardware verification and validation.

Acknowledgments

We would like to thank the engineers who have contributed to the implementation of ARGO in Python - Ihsan Majid, Linus Maximo, Matthew Feblowitz, Vagdevi Chidella, Atul Bhosale, Chandrika Vengala, Quinn Whitaker and Christopher Hules, for their valuable support and collaboration.

References

[1] DirectX12 Graphics Pipeline https://microsoft.github.io/DirectX-Specs/

[2] Intel Xe GPU whitepaper https://cdrdv2-public.intel.com/758302/introduction-to-the-xe-hpg-architecture-white-paper.pdf

Appendix

[1] *Thread* in a GPU is the smallest unit of execution that runs a shader or compute program independently on the GPU's parallel processing cores.

[2] *Message* in a shader program refers to a load or store instruction that GPU core (Execution Unit) uses to communicate with memory or any other specialized shared function like texture or load store cache

[3] *Shared Local Memory* is a small, fast, on -chip memory shared among threads within the same workgroup in GPU programming model

[4] *Bounding Volume Hierarchy* is a tree-like data structure that organizes a scene's geometry into nested bounding shapes, and helps accelerate the ray triangle intersection tests.

[5] *Unified Return Buffer* is a shared memory region used by different stages of graphics pipeline like vertex shader, tessellation and geometry to share data efficiently.