GLACI: Arbitrary Code Instrumentation Tool for OpenGL

Shotaro Tsuboi
Graduate School of Informatics,
Nagoya University
Nagoya, Japan
e-mail: s_tsuboi@ertl.jp

Yixiao Li
Graduate School of Informatics,
Nagoya University
Nagoya, Japan
e-mail: liyixiao@ertl.jp

Yutaka Matsubara
Graduate School of Informatics,
Nagoya University
Nagoya, Japan
e-mail: yutaka@ertl.jp

Hiroaki Takada
Graduate School of Informatics,
Nagoya University
Nagoya, Japan
e-mail: hiro@ertl.jp

***Abstract*—Modern embedded systems usually run multiple graphics applications concurrently, making efficient Graphics Processing Unit (GPU) resource management a critical challenge. To address this need, we present Arbitrary Code Instrumentation tool for OpenGL (GLACI), a flexible tool that enables transparent interception of Open Graphics Library (OpenGL) Application Programming Interface (API) calls to instrument arbitrary code without modifying the application or the graphics stack. GLACI-based module can cooperate with the GPU resource manager to support advanced features such as real-time Frames Per Second (FPS) monitoring, Quality of Service (QoS) based resource limiting and on-demand tracing. A prototype is created and evaluated on Intel and NVIDIA platforms to show the portability and usefulness of GLACI. By offering a unified, hardware-independent and lightweight solution, GLACI broadens the scope of GPU resource control and provides a practical foundation for both development and production environments.**

Keywords-embedded systems; OpenGL; code instrumentation, GPU resource management.

I. Introduction

In modern embedded systems, multiple graphics applications with varying reliability and requirements can share a single Graphics Processing Unit (GPU). For example, in automotive systems, the GPU is used for displaying the speedometer, navigation In-Vehicle Infotainment (IVI) displays and other third-party applications concurrently. Currently, Open Graphics Library (OpenGL) [1] is the most commonly supported and widely used graphics Application Programming Interface (API) for such applications.

However, due to the lack of tracing and resource management techniques for production environment, it is difficult to debug and develop such systems with necessary Quality of Service (QoS) satisfied. For instance, if a third-party application installed by the user consumes too much GPU resource, it can cause interference with the critical services (e.g., speedometer). A reliable system should be able to detect such kind of performance issues, record useful traces for analysis, and adjust the GPU resource allocation according to the QoS settings while it is running.

Most of the previous studies about scheduling GPU-sharing tasks focus on the scope of General-Purpose computing on Graphics Processing Unit (GPGPU) applications rather than the graphics applications [2]. These studies typically assume that the source code of the GPGPU tasks is available and can be modified to assist the GPU resource management. Meanwhile, many graphics applications, especially the third-party ones, are only available in binary executable files. The programming models of GPGPU tasks (computation-intensive functions offloading) and graphics tasks (complex rendering pipeline) also have a significant difference. Therefore, it is difficult to reuse these techniques for GPGPU tasks on graphics applications.

Some previous studies on improving the QoS of graphics applications have been proposed, but these methods face many challenges in terms of practicality. A common approach is to override the default scheduler with a QoS-aware one by modifying the kernel-space GPU driver [3][4], which has poor portability and maintainability since it depends on a specific GPU model and kernel version. It is also possible to manage the GPU resource by extending the implementation of OpenGL library [5][6] but many popular GPU vendors, including NVIDIA, only provide unmodifiable proprietary OpenGL libraries. Therefore, the usefulness of these studies is highly restricted in real-world systems.

In this paper, we propose Arbitrary Code Instrumentation tool for OpenGL (GLACI), an open-source tool which can assist the system developer to overcome the above limitations. With GLACI, hardware-independent modules for OpenGL API instrumentation can be effortlessly implemented. It allows us to dynamically trace and change the behavior of graphics applications, by adding custom code around OpenGL API calls, without acquiring and modifying any source code of the application and OpenGL library. If an application is executed with GLACI-based module loaded, the GPU resource manager can attach to it for monitoring and controlling.

The main contributions are listed as follows.

- GLACI, a generic tool for implementing hardware-independent OpenGL API instrumentation modules, which can change the behavior of application and library without modifying any source code, is proposed.

- A prototype including examples of GLACI-based module and GPU resource manager is created to show the usefulness of our method.
- Two representative platforms, based on Intel and NVIDIA, are used to evaluate the functionality and overhead.
- The source code of GLACI and the prototype is publicly available for reproducing and extension [7].

The rest of the paper is organized as follows. Section II discusses and compares previous studies with similar goals and methods. The details of GLACI are explained in Section III. Section IV uses a prototype of GLACI-based module and GPU resource manager to show the usefulness. Section V assesses our method by evaluating the prototype on Intel and NVIDIA platforms. Finally, the research is concluded in Section VI.

II. Related Work

A. GPU Resource Management

Previous studies have shown that it is possible to limit GPU bandwidth or guarantee Frames Per Second (FPS) for each application by inserting some processing into the graphics stack.

In the FPS control methods using execution time prediction [5][6], OpenGL API calls are monitored to obtain parameters such as the number of vertices and fragments to predict execution time which is used for GPU task scheduling. This approach modifies the source code of OpenGL library to acquire parameters, and the low-level GPU driver to apply scheduling policies.

In the QoS-based controlling methods [3][4], graphics APIs are modified to acquire QoS metrics. These methods also modify GPU drivers to apply scheduling policies. Some studies replace the low-level GPU task scheduler with a custom one by modifying the GPU driver [2][8].

These existing control methods lack generality for different platforms and GPU drivers, and require re-implementations for various environments. The modifications to the target applications are also needed in some methods, which makes them not feasible for third-party applications without source code. Meanwhile, GLACI focuses on supporting the resource management on the high-level hardware-independent OpenGL API layer as possible, rather than modifying the source code of existing graphics stack. If necessary, GLACI-based module can also cooperate with the GPU driver for fine-grained control.

B. OpenGL Tracing Tools

Several tracing tools for OpenGL API have been proposed to support the analysis of various metrics of the rendering commands and procedures. There are mainly two types of such tools: vendor-independent tools, and vendor-specific tools.

RenderDoc [9] and Apitrace [10] are two representative vendor-independent tools for debugging, tracing, and performance analysis of multiple graphics APIs, including OpenGL. These tools always hook every single OpenGL API call when the application is running, in order to produce a detailed trace file with all inputs, outputs and states recorded. Users can use the trace file to replay the rendering process of a frame for detailed behavior and performance analysis.

GPU vendors also provide tools to visualize rendering processes on their GPUs, such as Intel GPA [11] and NVIDIA Nsight Graphics [12]. These tools offer detailed views of processing at the GPU core level and can be used for low-level optimization. However, these vendor-specific tools only work on specific platforms, and most of them are proprietary software without source code provided, which makes it difficult to extend their functionality.

These existing tracing tools have fixed tracing scope and are designed for the test environment. For example, the tracer cannot be dynamically switched on and off when the application is running. It is also not possible to specify what information should be obtained to meet different requirements. Therefore, using these tools for tracing all applications in the production environment will cost a huge amount of resource. Further, since the tracing results can not be accessed from the GPU resource manager in real time, they are only useful for the postmortem analysis.

GLACI is not only capable of implementing the fixed-purpose tracing feature equivalent to the vendor-independent tools, but can also expose interfaces to communicate with the GPU resource manager to support advanced features like live performance monitoring, on-demand tracing and resource limiting. Therefore, unlike other tools, GLACI can be used in both testing and production environments.

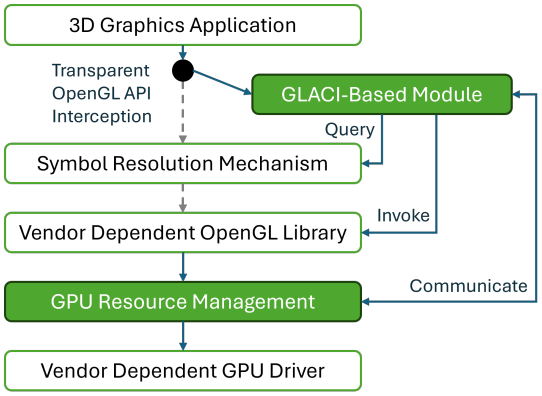


Figure 1. The overview of common functions in a GLACI-based module.

III. Proposed Method

A. Overview

GLACI is a hardware-independent tool that allows developers to effortlessly create modules capable of extending the functionality of existing graphics stack by instrumenting OpenGL API calls. Figure 1 shows an overview of how a GLACI-based module typically works in the graphics stack. The module can transparently intercept the OpenGL API calls and communicate with GPU resource manager, without modifying the source code of graphics application, OpenGL library and GPU driver.

A graphics application must be able to run on different versions of graphics stack without rebuilding the software, because the GPU vendor frequently updates the OpenGL library and GPU driver for optimization and bug fixing. Some systems even require the same application to run on GPUs from different vendors (e.g., the diversified GPU solutions for Android devices). To meet this portability requirement, OpenGL has introduced an advanced symbol resolution mechanism, instead of naively linking the application to some specific libraries.

When a GLACI-based module is loaded, it will use the symbol resolution mechanism to query the symbol addresses of all OpenGL APIs at first, and then mimic and override that mechanism to redirect the API calls to automatically generated wrapper functions with custom code instrumented.

If a system runs multiple applications with different QoS levels or priorities, there is usually a GPU resource manager located between the OpenGL library and the GPU driver to monitor and properly schedule the GPU usage of each application. However, a GPU resource manager can only attach to the applications with necessary interfaces for communication included, which means most of the third-party and proprietary applications are out of scope. A main benefit GLACI offers is that it can insert such interfaces to any OpenGL application to make it controllable from the GPU resource manager.

Figure 2 shows the flow of how GLACI will process a user-defined module project to build a loadable module binary. OpenGL is a very complex API specification with many different versions (e.g., GL 1.0 to 4.6, ES 1.0 to 3.2, SC 1.0 to 2.0) and additional extensions (e.g., ARB, GLX). Further, although OpenGL is a platform-independent specification in general, vendors also have added some special features in their proprietary library implementation. In the field of embedded systems, target boards usually support some specific versions of OpenGL (e.g., the popular Raspberry Pi 4 only runs GL 2.1 and ES 3.1). Therefore, it is impractical for us to assume the system uses and only uses the latest OpenGL version and vendor-independent features. Khronos Group has released the official Extensible Markup Language (XML) definition files of OpenGL API specification, including all versions and optional features. To address the challenge above, GLACI can load these XML files to build modules for a specific target system.

The user-defined module project consists of an instrumentation script in Python and some extra source files in C++. The instrumentation script defines the rules to instrument OpenGL API calls. The extra source files include the code with no need to be dynamically generated (e.g., data structure definitions and algorithm implementations). The GLACI core will follow the instrumentation script to generate a single source file for the module with OpenGL API wrapper functions and extra source code included. Finally, a shared library binary of the module will be built from the generated source file. We can use the `LD_PRELOAD` environment variable to start graphics application with the module loaded.

It should be noted that while the GLACI module is written in C++, it does not impose restrictions related to the programming languages of target graphics applications. Since the method operates at the binary interface level, applications developed in any language capable of invoking OpenGL APIs from the instrumented library can seamlessly benefit from GLACI without additional adaptation efforts.

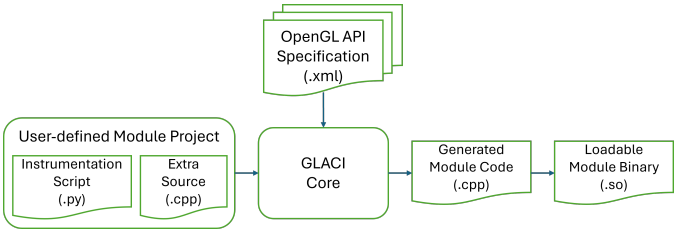


Figure 2. The process flow of how GLACI builds a module project.

B. Transparent OpenGL API Interception

GLACI-based module is loaded with the `LD_PRELOAD` feature, which allows us to override existing functions in the standard shared libraries of the graphics stack. However, to achieve portability and compatibility, OpenGL applications are not directly linked with a specific graphics stack. Instead, an advanced symbol resolution mechanism including the following three methods is provided for the applications to find the symbols at runtime.

- **Runtime linker:** In some systems, especially those using Mesa 3D graphics stack, a part of OpenGL API symbols (e.g., GLX extension for X11 window system) may be implemented in a shared library with stable Application Binary Interface (ABI). Therefore those shared libraries are directly linked to the application, and their symbols are resolved by the standard runtime linker.
- **dlopen/dlsym dynamic loader:** The graphics stack calls `dlopen` function to load OpenGL libraries by explicitly specifying the file names according to the actual running platform. It will then call `dlsym` function to dynamically search the symbol addresses of OpenGL API functions in these loaded libraries.
- **OpenGL *GetProc* functions:** OpenGL API specification also defines functions (e.g., `glXGetProcAddress`) to obtain symbol address by API name. Unlike the above two methods are provided by and dependent on the OS, this method is platform-independent.

To intercept OpenGL API transparently on various platforms and graphics stacks, GLACI must support all these methods to completely override the original symbol resolution mechanism.

To support the runtime linker method, GLACI will generate wrapper functions with the same prototypes for all OpenGL API functions, so the linker will always return the symbol addresses in our module instead of the original ones. Figure 3 shows an example of the glXSwapBuffers API.

To properly invoke the original API implementation from the generated wrapper function, GLACI shall initialize the

```
static typeof(glXSwapBuffers) *
    original_glXSwapBuffers = NULL;

void glXSwapBuffers(Display *dpy, GLXDrawable
    drawable) {
  ... /* instrumented code before glXSwapBuffers */
  original_glXSwapBuffers(dpy, drawable);
  ... /* instrumented code after glXSwapBuffers */
}
```

Figure 3. Example of generated glXSwapBuffers wrapper function.

```
__attribute__((constructor))
void load_original_functions(){
    ...
    original_glXSwapBuffers =
        dlsym(RTLD_NEXT, "glXSwapBuffers");
    ...
}
```

Figure 4. Example of initializing original_glXSwapBuffers.

function pointers to correct symbol addresses before the application starts to call any OpenGL API function as shown in Figure 4.

To support the dlopen/dlsym dynamic loader method, GLACI must solve the following issues.

- All the generated wrapper functions for OpenGL API are ignored because the `dlsym` function will only search symbols in the library file dynamically loaded by the `dlopen` function.
- The method of initializing original function pointers at startup does not work since the symbol addresses are unknown until the graphics stack calls and obtain the return value from the `dlsym` function.

GLACI addresses these issues by overriding the `dlsym` function with a modified version as shown in Figure 5. It will call the original `dlsym` function at first to get the symbol address. If the symbol is not an OpenGL API function, it will just return the address obtained. For OpenGL API symbols, the obtained symbol address will be stored in the original function pointer, and the address of corresponding wrapper function will be returned.

It must be noted that we cannot use the name `dlsym` to call

```
void *dlsym(void *handle, const char *symbol) {
    auto ptr = original_dlsym(handle, symbol);
    ... /* other OpenGL API functions */
    if (strcmp("glDrawArrays", symbol)==0) {
        /* initialize original function pointer */
        original_glDrawArrays = ptr;
        /* return GLACI wrapper function */
        return glDrawArrays;
    }
    ... /* other OpenGL API functions */
    return ptr;
}
```

Figure 5. Example of resolving glDrawArrays with modified dlsym.

```
void *original_dlsym(
  void *handle, const char *symbol)
{
  static dlsym_func_t original_dlsym_ptr
    = nullptr;
  if (original_dlsym_ptr == nullptr)
  {
    auto lib_handle =
      dlopen("libc.so.6", RTLD_LAZY);
    original_dlsym_ptr
      = dlvsym(lib_handle, "dlsym",
          GLIBC_VERSION_STR);
  }
  return original_dlsym_ptr(handle, symbol);
}
```

Figure 6. The core logic of original_dlsym.

```
void *glXGetProcAddress(const char *procName)
{
    auto procPtr = (*original_glXGetProcAddress)(
        procName);
    ... /* other OpenGL API functions */
    if (strcmp("glHint", procName) == 0)
    {
        /* initialize original function pointer */
        original_glHint = procPtr;
        /* return GLACI wrapper function */
        return glHint;
    }
    ... /* other OpenGL API functions */
    return procPtr;
}
```

Figure 7. Example of resolving glHint with modified glXGetProcAddress.

the original version of `dlsym` function, since it has already be overridden by our module. To avoid this circular reference, GLACI implements `original_dlsym` function as shown in Figure 6, which can search and call the original `dlsym` using `dlvsym` (dlsym with versioning) function.

Similarly, to support the method using the OpenGL *GetProc* functions, GLACI also implements modified versions to override them. An example of `glXGetProcAddress` is shown in Figure 7. Because the original function pointers of *GetProc* functions can be obtained from the other two methods, they are more easier to implement than the modified `dlsym` function.

With these symbol resolution methods supported, the GLACI-based module can fully intercept all OpenGL API calls to execute the instrumented code.

C. Code Instrumentation Example

GLACI instruments the OpenGL API functions by following the hooks defined in the instrumentation script of the module project. Figure 8 is an example of a hook for printing debug messages. A filter function is set to the `is_target` parameter so GLACI core will only apply this hook to OpenGL API of draw commands. The `before_run` and `after_run` parameters specify the code should be added before and after calling the hooked function.

```
def _print_enter(f: func.Func) -> str:
    return f'std::cerr␣<<␣"{f.name}␣Enter"␣<<␣std::
        endl;'

def _print_leave(f: func.Func) -> str:
    return f'std::cerr␣<<␣"{f.name}␣Leave"␣<<␣std::
        endl;'

debug_hooks = func.Hooks(
    header="#include␣<iostream>",
    hook_funcs=[
        lambda f: func.Hook(
            is_target=lambda f: "glDraw" in f.name,
            before_run=_print_enter(f),
            after_run=_print_leave(f),
        ),
    ],
)
```

Figure 8. Example of hook in the instrumentation script.

```
extern "C" PUBLIC
void glDrawBuffer(GLenum buf) {
    std::cerr << "glDrawBuffer␣Enter" << std::endl;
    (*original_glDrawBuffer)(buf);
    std::cerr << "glDrawBuffer␣Leave" << std::endl;
}

extern "C" PUBLIC
void glDrawBuffers(GLsizei n, const GLenum *bufs) {
    std::cerr << "glDrawBuffers␣Enter" << std::endl;
    (*original_glDrawBuffers)(n, bufs);
    std::cerr << "glDrawBuffers␣Leave" << std::endl;
}

... // other instrumented *glDraw* functions
```

Figure 9. Example of generated wrapper functions.

After processing this hook, GLACI will generate the wrapper functions for OpenGL *glDraw* API as shown in Figure 9.

IV. GPU Resource Manager Prototype

Although the GLACI-based module is also able to work as a standalone tool, the key characteristic distinguishing our method from existing tools is that it can communicate and cooperate with the GPU resource manager to dynamically monitor and control the running OpenGL applications. This feature makes GLACI a useful tool in both the development environment and the production environment. As a proof-of-concept, we have created a prototype that includes a GLACI-based module and a GPU resource manager. In this section, we will use it to explain how GLACI can help in implementing several real-world use cases.

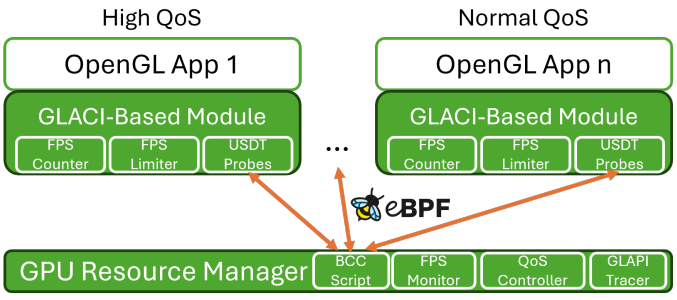


Figure 10. The overview of GPU resource manager prototype.

Figure 10 shows the overview of our prototype. The OpenGL applications are launched by the GPU resource manager with QoS priority assigned and GLACI-based module loaded. Userspace Static Defined Tracing (USDT) probes [13], generated by the GLACI-based module, are used as the communication channel between the application and the GPU resource manager to achieve dynamic control. By default, these probes are just No Operation (NOP) instructions with ignorable performance cost. The GPU resource manager includes a BPF Compiler Collection (BCC) [14] script which can dynamically generate and attach extended Berkeley Packet Filter (eBPF) programs to obtain information from and send control parameters to the running OpenGL applications. This lightweight yet extendable communication mechanism allows us to support the services of GPU resource manager with very low overhead. We have implemented the several services to demonstrate that GLACI can help to address real-world use cases as follows.

FPS monitor and alarm. OpenGL applications, especially those prebuilt ones, are usually designed to work at a specific target FPS to achieve a predictable GPU resource usage. If the actual FPS of an application differs significantly from the target FPS, there is a high probability that some issue has occurred during the execution. GLACI will insert the code and USDT probe of a frame counter around the OpenGL frame-swapping API to gather the FPS data. The GPU resource manager will attach to the related probe of each application to achieve a system-wide FPS monitoring in real time. If any unexpected FPS value has been detected, it can further generate an alarm to trigger necessary actions (e.g., start tracing the related OpenGL application).

QoS-based resource control. To deliver a sufficient quality of service, it is typically necessary to adjust the GPU resource usage limit per application at runtime. For example, if an application with normal QoS priority is the only running application, it can be allocated with full GPU resource. However, if applications with normal and high QoS priority are running at the same time, we should limit the GPU usage of the normal one to guarantee the FPS of application with high priority. The GLACI-based module has implemented a simple FPS limiter which is disabled by default. The limiter uses the control parameter to calculate the minimum render time of a frame. If the frame time of application is rendered faster than the limit's value, necessary delay duration will be inserted. When the application of high QoS priority is executed or terminated, the GPU resource manager will adjust and enable the FPS limiter of the normal QoS ones by sending control parameter to their related probes.

On-demand OpenGL API tracing. Although tracing the API calls is very helpful to analyze the performance and behavior of OpenGL applications, the usefulness of exist-

TABLE I. EVALUATION PLATFORMS

	Intel NUC	NVIDIA Jetson
CPU	Core i5-1240P	6-core Arm Cortex-A78AE v8.2
GPU	Intel Iris Xe	1024-core NVIDIA Tegra Orin
RAM	32GB	8GB
OS	Ubuntu 22.04	Ubuntu 22.04

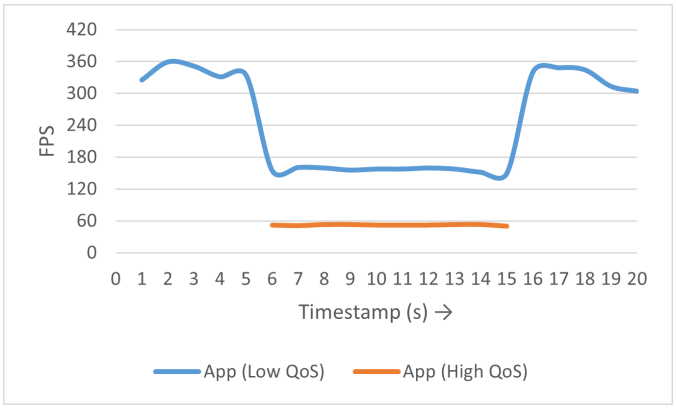
ing tools is severely restricted due to the lack of support in the production environment. The scope of these tracing tools cannot be dynamically changed while applications are running, which leads to unavoidable high overhead of system-wide continuous tracing and frequent restarts of applications. GLACI can overcome this drawback by supporting the on-demand tracing feature. It will insert USDT probes at the entry and exit points of each OpenGL API function. The tracing code of these probes can be attached or detached as needed by the GPU resource manager without restarting the running applications. This feature allows us to effortlessly create useful tracing policies. For example, we can disable all tracers by default to deliver the best overall system performance, and automatically enable tracing for a specific application when a performance issue is detected from that application.

Our experience in developing this prototype confirms that the learning curve for implementors is modest in practice. The GLACI-based module employs a simple instrumentation script written in Python, which is accessible to developers with basic scripting experience. Moreover, integrating GLACI modules with the GPU resource manager via USDT probes and eBPF programs does not require extensive prior knowledge, as these technologies are widely adopted and have substantial community support. Therefore, implementors can efficiently leverage our proposed method with minimal initial effort.

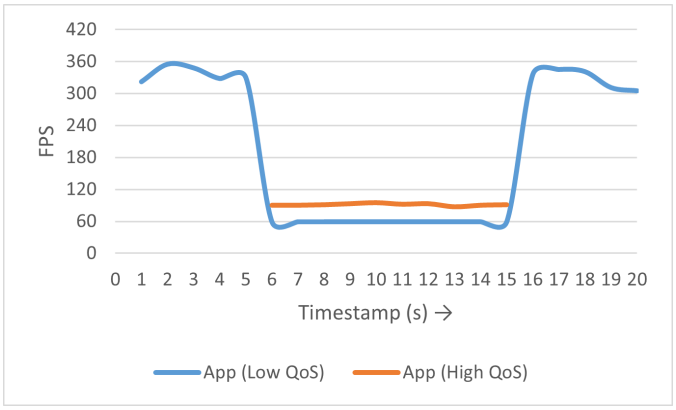
V. EVALUATION

In this section, we evaluate the GPU resource manager prototype on two mainstream platforms, Intel NUC and NVIDIA Jetson, to examine the functionality and overhead of GLACI. Intel platform provides an open source OpenGL library while the NVIDIA one is proprietary. Although the OpenGL implementations are vastly different, the high portability of GLACI allows us to use the same source code to build the prototype project without reimplementing for each platform. The specifications of the evaluation platforms are shown in Table I.

The benchmark programs from `glmark2` [15] (version 2021.02) are chosen as the graphics applications to test our prototype. `glmark2` is a lightweight OpenGL benchmark suite widely available on many platforms, with 17 representative scenes included to measure many aspects of the OpenGL specification. Since our method does not require any source code modification to the application and graphics stack, all related software components are installed using the official binary packages from Ubuntu. In this section, we always set the rendering resolution of `glmark2` to 1920x1080 for evaluation.



(a) Default settings without GLACI



(b) Prototype of GPU resource manager and GLACI-based module

Figure 11. Example to demonstrate how our prototype can improve QoS.

To demonstrate the effectiveness of QoS-based resource control, we run and measure an example using two `glmark2` program: `refract` and `terrain`. `refract` can run at around 333 FPS with full GPU resource allocated on the NVIDIA platform while `terrain` can run at around 119 FPS under the same condition. We use `refract` as the application with low QoS level and `terrain` with high QoS level. In the experiment, `refract` starts at first and keeps running, and `terrain` will start 5 seconds later and run for 10 seconds, to simulate the scenario the user launches an application with high QoS level while an application with low QoS level is running. Figure 11 shows the FPS data measured on the NVIDIA platform, and the Intel platform also has a similar trend. Without GLACI, the application with high QoS level failed to meet 60 FPS requirement (only 52 FPS on average). With the GPU resource manager and GLACI-based module, when application with high QoS level is running, the application with low QoS level will be locked to 60 FPS to deliver a desired performance for both applications.

We have measured the average FPS of `terrain` under the following conditions on the two platforms to evaluate the runtime overhead of our prototype.

- **No GLACI Loaded**: The application runs without GLACI-based module loaded.

TABLE II. OVERHEAD OF THE PROTOTYPE

	`glmark2 (terrain)` Average FPS	
	Intel NUC	NVIDIA Jetson
No GLACI Loaded	222	119
FPS Monitor & Alarm	221	118
QoS-based FPS Limiter	220	118
API Tracing (NOP)	218	117
API Tracing (Logging)	212	116

- **FPS Monitor & Alarm**: GLACI-based module is loaded and the GPU resource manager enables the function of FPS monitoring and alarming.
- **QoS-based FPS Limiter**: Besides the FPS monitoring and alarming, the QoS-based FPS limiter is also enabled.
- **API Tracing (NOP)**: GLACI-based module is loaded and the GPU resource manager enables API tracing. However, all API probes are attached with an empty function.
- **API Tracing (Logging)**: All API probes are attached with a logging function sending related messages to the trace buffer.

The measured results are shown in Table II. Compared to the average FPS without GLACI loaded, the overhead is barely perceptible to human eyes. It indicates that, the performance cost of GLACI should be small enough to be used in the production environment.

VI. CONCLUSION AND FUTURE WORK

This paper introduced GLACI as a generic, portable and low-overhead framework for dynamically instrumenting OpenGL API calls. By completely overriding the symbol resolution mechanism of OpenGL, GLACI can overcome the common and major limitation of existing methods that requires source code modifications to the application, OpenGL library or GPU driver.

We created a prototype to showcase how GLACI-based module and GPU resource manager can communicate and cooperate to support real-world use cases including performance monitoring, dynamic resource allocation adjustments and on-demand tracing. Two representative Intel and NVIDIA systems are used to evaluate the portability and usefulness of the prototype. The experimental results of overhead measurement confirm that the proposed approach remains lightweight enough for production scenarios.

Future work will focus on applying GLACI to further improve the GPU resource management, particularly in areas such as adaptive QoS management on metrics like utilization, frame time and render latency. Since some vendors have started to release open-source kernel-space GPU drivers in recent years, expanding GLACI's capabilities to support hooks at the GPU driver level and enabling closer integration with GPU driver control mechanisms are also key directions. These vendors may also embrace our approach to create open source debugging, tracing and management tools, since the lightweight and non-intrusive design of GLACI can offer enhanced instrumentation features with minimal efforts. Leveraging GPU driver abstraction layers, such as Gallium3D [16], could facilitate these advancements and further enhance the tool's applicability. Taken together, these advances can mark an important step toward a comprehensive, vendor-agnostic framework for managing the complex GPU requirements of modern embedded systems.

ACKNOWLEDGMENT

We would like to express our sincere gratitude to Suzuki Motor Corporation for their valuable collaboration and support throughout this research. Their expertise and contributions have been essential to the success of this work.

REFERENCES

[1] Khronos, "Opengl - the industry standard for high performance graphics," 2021, [Online]. Available: https://www.opengl.org/ (visited on 04/13/2025).

[2] Y. Wang, C. Liu, D. Wong, and H. Kim, "GCAPS: GPU Context-Aware Preemptive Priority-Based Scheduling for Real-Time Tasks," in *36th Euromicro Conference on Real-Time Systems (ECRTS 2024)*, 2024.

[3] Q. Lu, J. Yao, H. Guan, and P. Gao, "Gqos: A qos-oriented gpu virtualization with adaptive capacity sharing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, pp. 843–855, 2020.

[4] M. Xue *et al.*, "gScale: Scaling up GPU virtualization with dynamic sharing of graphics memory space," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 2016, pp. 579–590.

[5] S. Schnitzer, S. Gansel, F. Dürr, and K. Rothermel, "Concepts for execution time prediction of 3d gpu rendering," *Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems (SIES 2014)*, pp. 160–169, 2014.

[6] S. Schnitzer, S. Gansel, F. Dürr, and K. Rothermel, "Real-time scheduling for 3d gpu rendering," in *11th IEEE Symposium on Industrial Embedded Systems (SIES 2016)*, 2016, pp. 1–10.

[7] GLACI, "Source code," 2025, [Online]. Available: https://github.com/ertlnagoya/glaci-icons-2025 (visited on 04/13/2025).

[8] S. Kato, K. Lakshmanan, R. R. Rajkumar, and Y. Ishikawa, "Timegraph: Gpu scheduling for real-time multi-tasking environments," in *USENIX Annual Technical Conference*, 2011.

[9] B. Karlsson, "Renderdoc," 2025, [Online]. Available: https://renderdoc.org/ (visited on 04/13/2025).

[10] apitrace, "Source code," 2025, [Online]. Available: https://apitrace.github.io/ (visited on 04/13/2025).

[11] Intel, "Graphics performance analyzers," 2025, [Online]. Available: https://intel.github.io/gpasdk-doc/ (visited on 04/13/2025).

[12] NVIDIA, "Nsight graphics," 2025, [Online]. Available: https://developer.nvidia.com/nsight-graphics (visited on 04/13/2025).

[13] B. Cantrill, M. W. Shapiro, and A. H. Leventhal, "Dynamic instrumentation of production systems," in *USENIX Annual Technical Conference, General Track*, 2004.

[14] iovisor, "Bpf compiler collection (bcc)," 2025, [Online]. Available: https://github.com/iovisor/bcc (visited on 04/13/2025).

[15] glmark2, "Source code," 2025, [Online]. Available: https://github.com/glmark2/glmark2 (visited on 04/13/2025).

[16] Mesa 3D, "Gallium documentation," 2025, [Online]. Available: https://docs.mesa3d.org/gallium/index.html (visited on 04/13/2025).