ESCUELA POLITÉCNICA SUPERIOR
UNIVERSIDAD DE LAS ISLAS BALEARES

PROYECTO FINAL DE CARRERA

Estudios:

Ingeniería Informática

Título:

***Android Tracer for OpenGL ES* C Code Generation, Automatic Code Refactoring, and Replaying**

Documento: **Memoria**

Alumno:
Antonio Tejada Lacaci

Director:
Dr. Isaac Lera Castro

Fecha: 5 de Junio de 2017

Evaluación positiva del director

El director del proyecto Dr. Isaac Lera Castro hace constar que el proyecto "*Android Tracer for OpenGL ES* C Code Generation, Automatic Code Refactoring, and Replaying" está finalizado bajo su dirección y puede pasar a la fase de evaluación.

Firma:

Quiero dar las gracias a Carlos Guerrero por conseguir lo que veinte años no han conseguido, empujarme a terminar la carrera, y a Isaac Lera por ser tan flexible y tener la paciencia de dirigirme el proyecto a distancia.

Gracias a mi tío por los coscorrones, a mi tía por los pinchazos, y a mis padres por el resto de maltratos.

I Table of Contents

II List of Figures

III List of Tables

IV Abstract

Measuring raw *Graphics Processing Unit* (GPU) performance is an important task for *System on a Chip* (SoC) selection. Recording and replaying traces of graphics *Application Program Interface* (API) commands is a common method of measuring such raw GPU performance.

This work presents a method for generating C code from an *Android Tracer for OpenGL ES* trace, jointly with the corresponding APK file, so the trace can be replayed under different settings and devices for SoC GPU performance evaluation.

Additionally, in order to deal with big trace files, it applies a novel method of compressing C code by automatically finding subroutines in the C code generated from the trace, achieving upto 22x compression ratio.

Finally, it presents some of the results obtained with this tool across 6 Android architectures, giving insight into their performance at different framebuffer resolutions, and OpenGL ES synchronization methods.

V Keywords

Android, OpenGL, OpenGL ES, Android OpenGL ES Tracer, code generation, dictionary compression, automatic code refactoring, GPU performance, graphics profiling, SoC, SoC evaluation

1. Introduction

1.1. Document Structure

The Introduction chapter briefly defines the concepts and technologies necessary to understand the problematic, introduces the problematic itself and the motivation to solve it, and how the currently existing solutions fall short in some aspects.

The Proposal chapter describes a new solution to the problematic, the requirements to fulfill, the development planning, and the implementation details in depth.

The Results chapter presents a few of the results obtained with this tool.

The Limitations and Future Work chapter enumerates the limitations of this tool and possible improvements.

The Project History, Challenges and Curiosities chapter informally relates the origin of the different ideas that culminated in the implementation of this approach.

The Glossary succinctly defines a few of the most important terms used along this document.

1.2. Technological Landscape

This section presents brief introductions to the technologies involved in this work. For a more in depth explanation of those technologies, refer to the cited bibliography.

1.2.1. Android, AOSP, BSP, FireOS

Android is the most widely used mobile operating system with 2 billion monthly active users as of May 2017[1].
It was created inside Andy Rubin's startup Android Inc, later acquired by Google in 2005[2].

Android is composed of multiple projects, many of them open source under various licenses, but most of the source code to the Android operating system is available under an Apache 2.0 license at the *Android Open Source Project* or AOSP[3].

Not all code necessary to function a mobile device is freely available, though,

- several low-level proprietary packages are only distributed by the hardware vendor in binary form as part of a *Board Support Package* or BSP. These include security or communication firmware, userspace graphics libraries, etc.
- other additional packages (*Google Mobile Services* or GMS, including applications like Google Maps, Youtube, etc.) are only available in binary form and need to be licensed from Google[4].

Given the open source nature of Android, there have been several commercial forks developed independently from Google. One of those forks is FireOS from

Amazon[5], a version of AOSP extended with Amazon proprietary binaries for the range of Amazon mobile devices[6] (Kindle Fire, FireTV, FireTV Stick, etc.).

1.2.1.1. Android Application Development

Google provides freely downloadable cross-platform software kits for developing Android applications.

The Android development kits are split between the tools, compilers and libraries necessary to develop Java applications (*Android Software Development Kit* or Android SDK[7]), and the more reduced set of tools, compilers and libraries necessary to build native C applications (*Android Native Development Kit* or Android NDK[8]).

Android SDK applications are, for the most part, written in Java, while an NDK application consists of a Java layer and one or more shared library objects compiled from C or C++ that can be invoked from the Java layer via *Java Native Interface* or JNI calls.

The features supported by the NDK are only a very small subset of the features supported by the SDK[9], so it's frequent having to jump back and forth from native code to Java code in order to access functionality not supported by the NDK.
Google recommends using the NDK exclusively for those parts of an application that have performance requirements, or for code reuse with other platforms.

Once all the native and non-native pieces of an Android application have been built using Apache ANT builder[10], it's packaged into an *Android Application Package* or APK.
The APK can then be published to an application store like Google Play, or *sideloaded*, this is, installed on the device using the *Android Debug Bridge* or ADB (a usb connection between a host machine and an Android device previously configured in developer mode).

1.2.2. OpenGL

Open Graphics Library or OpenGL[11], is a cross-language cross-platform *Application Programming Interface* for displaying 2D and 3D computer graphics. It was designed by Silicon Graphics in 1992 as a substitute to their previous graphics library *IrisGL*[12].

Initially, the OpenGL evolution was governed by the *Architecture Review Board* or ARB, until Silicon Graphics' exit of the graphics industry on 2006, when it was grandfathered into the OpenGL Working Group of the Khronos Group[13], already responsible for several realtime multimedia APIs.

The OpenGL 1.0 specification was authored by Mark Segal and Kurt Akeley[14], later versions, including the current OpenGL 4.5[15], preserve the style and even part of the text of the original specification. The specification is freely downloadable at the Khronos OpenGL website[16].

OpenGL's specification purpose is double,

- unambiguously describes the API to application writers who wish to use OpenGL in their programs, and
- describes the API to OpenGL implementors or vendors who want to provide access to their products via OpenGL.

Implementors are also able to provide extended functionality not present in OpenGL by means of *extensions*, which are documented in the *OpenGL Extension Registry*[17] as addendums to the specification, and uniquely identified by a name prefixed by the vendor's identifier.
Many extensions are adopted by multiple vendors (in which case the extension name is prefixed with EXT) or have even a wider adoption and are condoned by OpenGL's ARB or Khronos (prefixed with ARB or KHR). Oftentimes those extensions make it into the next version of the OpenGL specification as core functionality.

There's no single implementation of OpenGL, instead, multiple parties can provide an OpenGL implementation. Normally the implementation is provided inside a driver for a hardware that accelerates totally or in part the OpenGL specification, but there are also cases where the implementation is purely in software (for example, the open source Mesa3D[18], Microsoft Windows' OpenGL 1.1[19] or Google Android's PixelFlinger[20] implementations).

The specification, while detailed, still allows implementors enough freedom to innovate while guaranteeing compatibility across implementations.

OpenGL doesn't require nor guarantee per-pixel accuracy across renders generated by different implementations, but it does guarantee a set of invariance rules inside the same implementation, geared towards supporting both techniques that can be implemented in a single pass, and techniques that require multiple passes over the framebuffer[21].

An OpenGL implementation must pass a set of required OpenGL conformance tests[22] to guarantee compatibility and in order to license the OpenGL certification logo.

Programming-wise, OpenGL is a state machine, where the *current state* of an OpenGL *context* is configured via different state-setting calls (current buffer of vertices, buffer contents, current texture, texture contents, etc.), and calls that trigger the rendering (drawing polygons, known as *primitives*)[23].

Once the rendering is triggered, operations in OpenGL mainly happen at two frequencies of execution,

- for each vertex that compose a given primitive, or
- for each *fragment* (color tuple plus and any ancillary data like depth or stencil value) that result of rasterizing the given primitive.

1.2.2.1. OpenGL ES

OpenGL for Embedded Systems or OpenGL ES[24] is a simplified subset of OpenGL tailored to work within the constraints of an embedded system.

As embedded systems have become more capable, OpenGL ES has been getting closer in functionality to OpenGL, from OpenGL ES 1.0, which didn't support programmable hardware, to the current OpenGL ES 3.2 which is close to parity with OpenGL 4.5.

OpenGL ES is the graphics API used by Android; all applications use it to display on the screen, either indirectly via the facilities offered by Android's view framework, or directly by making OpenGL ES calls.

1.2.2.2. EGL

The OpenGL specification is window-system independent and doesn't describe any of the necessary calls to display an OpenGL surface on a given window system or, more critically, even to create an OpenGL context (the necessary underlying structure holding the OpenGL state modified by OpenGL calls).

Each operating system normally provides one or more window-system dependent APIs that glues OpenGL to that operating system. For Windows, that is WGL, for X11 is GLX, AGL and CGL for OSX, and *Embedded-System Graphics Library* or EGL[25] for operating systems supporting OpenGL ES, Android amongst them.

The role of this window-system dependent library is to enumerate which framebuffer formats are supported by a given OpenGL implementation (also known as *pixelformats*), allow the creation of OpenGL contexts, framebuffer management and provide synchronization primitives.

1.2.3. Graphics Processing Units

Integrated circuits that offload graphics tasks from the main processing unit have existed for the longest time, but the name *Graphics Processing Unit* or GPU (as a counterpart to *Central Processing Unit* or CPU) wasn't widely used until the programmability of those graphics accelerators was exposed via a standard API.

The architectures behind those graphics accelerators, went through a series of evolutionary steps as circuit integration level increased:

1. Fixed function: The accelerator offloads from the CPU part of the graphics calculations (matrix transformations, vertex attribute interpolation, texture sampling and interpolation, etc.). That functionality is *fixed* and not modifiable by the application, even if internally may have been performed by programmable units
2. Configurability: Part of the programmability is exposed through APIs via a limited number of configurable stages[26] or parameters[27], each stage or parameter allowing to modify some computation for each vertex being transformed or pixel being rendered. There's no general-purpose programmability like control flow, unlimited memory accesses, etc.
3. Programmability: The graphics accelerator is now a fully featured parallel processing unit, capable of general purpose control flow, unlimited memory accesses, etc. Note that, even if general purpose, there are performance penalties to executing divergent code in those parallel units (i.e. code that

forces the parallel units inside the same cluster to take different execution paths).

Current GPUs are distant derivatives of the *Pixel Planes*[28] work done at the University of North Carolina at Chapel-Hill, which allowed parallelizing polygon rasterization by avoiding the serialized *Digital Differential Analyzer* or Bresenham algorithms, and switching to small parallel compute units that could perform per pixel evaluation of plane equations independently from each other.
Nowadays those parallel compute units are capable enough to perform the execution of generic program instructions for each vertex or pixel, and the overall architecture is also deeply pipelined to guarantee the required performance.

As mentioned, the hardware advances were hand-in-hand with evolution in graphics APIs so they could be exposed to applications via new revisions of the given graphics API. For an in-depth explanation on how current GPUs map to API constructs, refer to[29].

1.2.3.1. Mobile GPUs

Historically, GPUs used on mobile devices started significantly later, when the desktop GPU technology was well stablished, but followed a similar path to their desktop counterparts, trailing the features of the latter by a few generations, limited by the power consumption and size requirements of a mobile platform.

Nowadays, the biggest difference between desktop and mobile GPUs is not as much the features, which are almost at parity, as the emphasis in reducing power consumption.
This is achieved by reducing the peak performance rate of a mobile GPU (smaller amount of replicated compute units, lower execution frequencies, etc.) and, more importantly, reducing the amount of memory transactions.

Memory transactions are one of the main elements responsible for power consumption; the mobile GPU designers have focused in increasing memory locality and preventing unnecessary work to minimize those..

Non-mobile GPUs have mostly used *Immediate Mode Rendering* or IMR, where rendering commands are send to the hardware, and those are performed at that moment. This causes two main inefficiencies:

- Limited space locality benefits: Two polygons with spacial locality, i.e. that modify neighboring pixels, but sent to the hardware far apart in time, will need to bring the neighbor pixels from memory twice, without being able to exploit that locality, and
- Overdraw: Two opaque polygons that overlap, will result in unnecessary writes because only the topmost polygon will be finally shown.

In order to remove those inefficiencies, instead of executing rendering commands immediately, mobile GPUs use a *Tile-Based Rendering* or TBR[30] approach to execute them: the screen is *tiled* or divided in smaller regions, and the transformed polygons overlapping a given tile stored in a per tile *bin* or bucket.
At a later time, the bins are visited one by one and the rendering performed for each polygon contained in the bin, achieving space locality benefits.

In addition, the polygons inside a bin can be sorted by depth so only the pixel for the topmost polygon is ever written, in what it's called *Tile-Based Deferred Rendering* or TBDR[30], which eliminates overdraw.

1.2.4. System On a Chip

Contrary to desktop computers, where different tasks are carried by different chips (CPU, GPU, sound, network, etc.), on mobile architectures most of the tasks are carried by a single *System On a Chip* or SoC that integrates those multiple hardware blocks under a single package.

As is the case with mobile GPUs, the mobile SoC architects have focused on power savings, the most usual of which are:

- core offlining: blocks not being used (CPU cores, etc.) can be put in different sleep levels, upto the point of being completely shut down or *offlined*.
- frequency throttling: blocks function at low clock frequencies or *throttled* until there's a workload that requires higher performance, at which point the frequency is increased.

Note both cases trade off power for latency, since there needs to pass some time to detect that the new workload requires a higher performance level, and yet more time to perform the performance increase (be it by *onlining* a new core or by increasing the frequency).

All Android devices use some kind of SoC as the heart of the system, more than 95% of the smart phone devices use the ARM architecture[31].

1.3. Motivation

Given the huge Android market, but also the fierce competition, it's obvious that there's economic interest in designing devices that can address that market, and at the same time present advantages over the competition.

The design of such a device always starts with the evaluation of the possible SoCs to use. In that evaluation phase there are many factors that are taken into account like price, performance, availability, level of support from the vendor, to name a few.

1.3.1. Why measure raw GPU performance

The task of selecting a *System on a Chip* or SoC for a future mobile product involves evaluating alternatives by different hardware vendors, and even different configurations from the same hardware vendor.

In order to select the adequate SoC from a vendor and the proper mix and match of components inside the SoC, a detailed task of evaluation of each component is necessary to make sure it satisfies the functional and performance requirements of the product (if only to make sure that it outperforms previous generations and the competition).

In an ideal world, each vendor would provide a fully functioning system with all the hardware and software ready, and the evaluation would be just a matter of running the future product's final software on it and picking the best option.

Unfortunately, lead times for hardware are two years from design to fabrication, and software can trail one year or more behind that timeframe. By the time everything is ready, the technology is old and the competition is already out in the market.

It is therefore unrealistic to expect to be able to measure the product's final workload in the final system, and still fulfill time-to-market schedules, so different compromises have to be made.

One of such compromises is to apply a *divide-and-conquer* approach and measure selected components in isolation with regards to the rest of the system, in order to decouple the evaluation of that component from the rest.

One of such components is the *Graphics Processing Unit* or GPU, in charge of providing graphics hardware acceleration in order to display elements on the screen.

By measuring *raw* GPU performance (i.e. GPU loads as independent as possible from other loads like application's logic), the selection of the GPU component can be relatively decoupled from the rest of the components in the SoC.

1.3.2. Techniques to measure GPU performance

In order to measure GPU performance, several options exist, amongst them

- a dedicated benchmark (or microbenchmark) can be written or licensed,
- a real application can be instrumented and run, and
- a trace (a log of the graphics API calls performed) can be extracted from a real-life GPU workload and replayed on simulation or prototype hardware and software, without having to have all the software layers in place required for program where the trace was extracted from.

1.3.2.1. Microbenchmarks

Microbenchmarks give the peak rate of different stages of the architecture (texture rate, fillrate, *Arithmetic-Logic Unit* or ALU rate), normally irrespective of the CPU overhead.

They are an important factor for evaluating a GPU since they allow to understand the best-case scenario performance, but they don't represent real-life workloads.

1.3.2.2. Off-the-shelf Benchmarks

The reference mobile GPU benchmarks are GFXBench[32] and, to a lesser degree, 3DMark[33].

These are commercial benchmarks not backed by a standards committee. As such, they follow the design principles of the companies that produce them, are closed, the

source code is not available, and need to be licensed for a large amount of money for commercial use.

Their main advantage is that they are widely used in the industry (GPU vendors always provide estimations or scores for early hardware) and have a database of results which makes it easy to compare current or past architectures.

The black box nature makes it hard to perform unplanned experiments (e.g. unsupported configurations like different color depths, etc.) and it's also unknown how much they measure GPU performance and how much they measure CPU performance.

Another problem with off-the-shelf benchmarks is that GPU vendors optimize towards them, sometimes overzealously[34][35], resulting in an arms race between benchmark writers and GPU vendors.

Finally, by their own nature, they may not represent the real workload for our product, just the workload that was meaningful to the benchmark's company.

1.3.2.3. In-house Benchmarks

Unlike off-the-shelf benchmarks, in-house benchmarks are more meaningful as they are tailored to the necessities they were designed for, but require development and updating to track the real life workload they try to represent.

1.3.2.4. Real-life Applications

A real life application can also be used, but normally comes with software dependencies that may not run on the SoC's early software layers.

They also have repeatability problems unless they are specifically written with that in mind, in which case that code path needs to be tested and maintained to avoid code rot.

1.3.2.5. Trace Capture and Replay

A real-life application is run and its graphics calls are logged to some trace file (trace capture).
An application that is able to replay that trace file has to be written, and the trace can then be replayed, possibly under different settings (as fast as possible, using the original captured timing, on larger displays, etc.).

As juxtaposition to the limitations mentioned for other approaches, the advantages of using a trace replayer to measure GPU performance:

- Only require an initial investment to develop the trace capture and the trace replayer.
- Trivial to generate new traces as the real life workload changes.
- Repeatable execution
- Meaningful, match real life workloads.

- Allow experimenting with different configurations or settings.
- Even in the case where SoC vendors optimize for them, they would be optimizing for the real-life workload, which is beneficial.

Besides workload modeling, traces have also been useful as bug reproducers: instead of forcing software and quality assurance engineers to install cumbersome applications and their dependencies, a trace of the bug can be extracted, the trace replayer attached to a bug, the code inspected, triaged and fixed more easily.

By modifying a trace you can also quickly prototype how a modification to an application you don't have the source code for, would look like (rendering at smaller resolution, color depth, enabling dithering, etc.).

All these advantages don't mean that the other techniques and approaches are invalid, a trace replayer is just another tool on the SoC evaluator's belt to be used when appropriate.

1.4. Trace Replay Approaches

There are two main approaches to trace replaying

- Create a generic trace replayer that takes any binary trace and interprets it at runtime, and
- Convert the trace into source code and compile it as a standalone application.

The first has the advantage that a single program can run multiple traces without the need to recompile it.

The second, that the generated source code can be inspected and modified or provided to third parties for their perusal. This can be considered an important feature since it allows, without disclosing important IP, SoC vendors to provide recommendations over the generated code. Also, SoC vendors normally have pipelines that are able to take graphics API calls to run then on hardware simulators and obtain early performance numbers.

In addition, trace compilation allows any necessary translation overhead (e.g. texture format conversion from a proprietary format to another) to be applied at code generation time, while this is harder to do on binary traces. Code execution can also be faster than trace interpreting, but this may or may not be a leading factor to decide for the code generator alternative.

1.5. Existing OpenGL ES Tracing Solutions

There are already a few OpenGL tracing solutions, but none of them generate code and have varying capture and/or replay support:

- apitrace[36] seems to be the most featured one, since it can capture and replay OpenGL ES 1.0 to 3.0
- gDebugger[37] only supports GLES 1.1
- RenderDoc[38] only started supporting OpenGL ES on November 2016
- gapid[39] can capture OpenGL ES but not replay.

Finally, Android Tracer for OpenGL ES[40] allows inspecting and visualizing the contents of an Android OpenGL ES trace. Its underlying trace recording infrastructure was used in this project as explained in Android Tracer for OpenGL ES

2. Proposal

Once sufficiently motivated to perform GPU performance evaluation, justified the adequacy of a trace code generator as a suitable tool for that, and documented the lack of an already existing alternative, this chapter describes in detail the implementation of such a tool.

2.1. Requirements

The proposed solution is a tool that can generate code from an OpenGL ES trace, compile that code into an Android application, and execute the application under different conditions.

More specifically, the implemented solution must:

- Support Android OpenGL ES 2.0.
- Have minimal CPU overhead at replay time.
- Measure GPU performance, seconds per frame and total frames per second.
- Allow generated code inspection and modification by third parties.
- Have no dependencies, generate a self contained APK, run on vanilla Android (AOSP), non-rooted.
- Be expandable to future requirements (measuring graphics driver overhead, etc.)

Additionally, the tool will perform automatic refactoring of the generated C code, in order to reduce compile times and the APK size.

2.2. Task Planning

Figure 1 shows the break down of the tasks and the amount of days allocated to each.

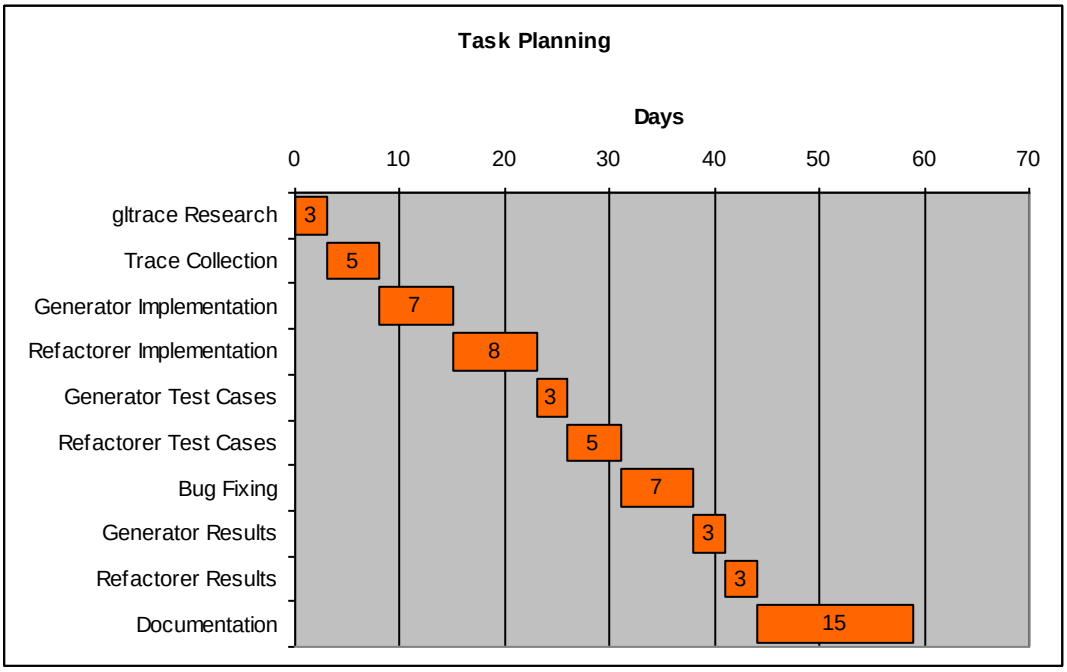

Figure 1: Task Planning

1. **gltrace Research**: Investigate how `.gltrace` binary files can be read. Validate approach on paper.
2. **Trace Collection**: Use the *Android Tracer for OpenGL ES* to record example traces, write scripts for automated recording.
3. **Generator Implementation**: Implement a script that generates C code from a binary `.gltrace`.
4. **Refactorer Implementation**: Implement a script that takes the generated C code and refactors it reducing its size.
5. **Generator Tests**: Write automated tests for the code generator.
6. **Refactorer Tests**: Write automated tests for the code refactorer.
7. **Bug Fixing**: Fix bugs in the code refactorer and generator.
8. **Generator Results**: Run the tool on several devices and settings, graph the performance results.
9. **Refactorer Results**: Analyze the refactorer compression efficiency.
10. **Documentation**: Write the documentation, analyze the results.

The tasks were not done in a serialized manner, but as it made sense. For example, tests were added as new functionality was implemented or bugs were found.
Sometimes, a test-driven development[41] approach was used where the test was written before the code, in order to understand better how the code should work.

Time-wise, the project was implemented as work allowed, between June 2014 and June 2017. The whole development timeline is listed under Git Repository.

2.2.1. Git Repository

As mentioned in Git, the version control system *git* was used to perform the development and can be cloned from the github repository `http://github.com/antoniotejada/glparse` with

```
1.      git clone https://github.com/antoniotejada/glparse
```

Table 1 shows the development timeline as per the git command

```
1.      git log --format="%ci%x09%s" --reverse
```

Date	Notes
6/15/2014	glparse.py OpenGL ES trace parser and player generator
6/15/2014	Added Android native activity that wraps traces generated by glparse.py, build scripts
6/15/2014	Added license to trace.c
6/18/2014	Fixed GL context creation,
6/21/2014	Added asset usage instead of inline constants, added EGL_BUFFER_PRESERVED, auto-detect trces with empty textures
6/23/2014	Split each frame in a different function, deinlining prototype
6/24/2014	Added parameter handling to deinline.py
6/28/2014	Added parameter handling to deinline.py
6/29/2014	glparse optimizations
7/1/2014	deinline suffix array implementation.
7/5/2014	deinline optimizations: switched suffix method from substring based dicts to length based dicts
7/6/2014	deinline optimizations: Added sliding window, refactored replace_code

10/26/2014	Made app fullscreen, load apk settings from command line, EGL info logging
10/28/2014	Added viewport/scissor scaling, frame capturing,
11/1/2014	Added per-trace packaging, EGL context config logging, frame & swap times
6/21/2016	Added Python build script, removed batch and shellscripts.
6/21/2016	Added test framework, some deinline tests
6/23/2016	Factored out code replacement routines, almost no functional changes.
6/29/2016	More refactoring of code replacement routines, added parameter coalescing, parameter tests
7/1/2016	Moved local declarations to the beginning of the function (47% more compression), removed empty lines
3/26/2017	Added glparse unit test, removed .out files, refactored common test code
3/27/2017	Created glcap.py utility to capture OpenGL ES traces without Android Studio
4/1/2017	Added glparse tests kipoprofile250, simple, triangle, added 64-bit trace support
4/7/2017	Proper context creation support, multicontext, glparse tests, build tests, 64-bit traces
4/8/2017	Deinline fixes for spaces or commas inside quotation marks, switches, enabled deinlining in build test.
4/9/2017	Delete arrays of resources support
4/14/2017	Added command line options, egl_swapbuffers_sync, fixed capture frame indexing
4/16/2017	Added asset file coalescing
4/18/2017	Code coverage pragmas, deleted dead code found by coverage
4/21/2017	deinline type casting
4/23/2017	Deinline aliased parameter coalescing
5/6/2017	Deinline mixed aliased/non aliased parameter support.
5/15/2017	Optimized gather_per_aliasing_instruction_information

Table 1: Development Timeline

Table 2 describes the files in the root directory of the aforementioned git repository.

File/Directory	Purpose
activity/	Activity template for Android NDK
build.bat	Invokes build.py
build.py	Invokes glparse.py, deinline.py, NDK build and ant build to generate an APK
deinline.py	Refactors a C file generated by glparse.py
external/	Khronos and protobuff files
generate_gltrace_from_inc.bat	Invokes several scripts to generate a .gltrace file from an existing trace.inc file
Glcap.bat	Invokes glcap.py
Glcap.py	Invokes an installed APK and generates an OpenGL ES trace.
glparse.py	Parses a .gltrace file and generates a trace.inc file
Parse_perf.py	Parse files in the perf directory to generate tables
Parse_window.py	Parse files in the perf directory to generate tables
perf/	Performance result log files
Profile.bat	Invokes cProfile and RunSnakeRun to find bottlenecks on glparse.py and deinline.py
run_perf.py	Runs generated APK on device under different command line parameters (resolutions, etc.)
run_perf_old.bat	Runs generated APK on device under different command line parameters (resolutions, etc.)
tests/	Tests for build.py, glparse.py, deinline.py
utils.py	Library of helper utilities

Table 2: Files and Directories

The full list of files and directories follows:

```
1.      |   build.bat
2.      |   build.py
3.      |   deinline.py
4.      |   generate_gltrace_from_inc.bat
5.      |   glcap.bat
6.      |   glcap.py
7.      |   glparse.py
8.      |   parse_perf.py
9.      |   parse_window.py
```

```
10.      |   profile.bat
11.      |   README.md
12.      |   run_perf.py
13.      |   run_perf_old.bat
14.      |   utils.py
15.      |
16.      ├──activity
17.      |   |   AndroidManifest.xml
18.      |   |   ant.properties
19.      |   |
20.      |   ├──jni
21.      |   |       Android.mk
22.      |   |       Application.mk
23.      |   |       common.h
24.      |   |       intent.c
25.      |   |       intent.h
26.      |   |       main.c
27.      |   |       trace.c
28.      |   |
29.      |   └──res
30.      |       └──values
31.      |               strings.xml
32.      |
33.      ├──external
34.      |   ├──google
35.      |   |       gltrace.proto
36.      |   |       gltrace_pb2.py
37.      |   |       protoc.exe
38.      |   |       readme.txt
39.      |   |
40.      |   └──khronos
41.      |           egl.xml
42.      |           gl.xml
43.      |           glx.xml
44.      |           readme.pdf
45.      |           README.txt
46.      |           wgl.xml
47.      |
48.      ├──perf
49.      |       allperf.log
50.      .       ...
51.      |       roth-sync_3_pb_true_0-720x1280-perf.log
52.      |
53.      └──tests
54.          |   build_test.py
55.          |   common.py
56.          |   deinline_test.py
57.          |   glparse_test.py
58.          |   make.bat
59.          |
60.          ├──build
61.          |       triangle.gltrace.gz
62.          |
63.          ├──deinline
64.          |       codeflow.c
65.          |       complex.c
66.          |       locals.c
67.          |       params.c
68.          |       params_bug_aliasing.c
69.          |       params_casting.c
70.          |       params_coalesced.c
71.          |       params_coalesced_different_names.c
72.          |       params_coalesced_partial.c
73.          |       params_common.c
74.          |       params_common_void.c
75.          |       params_types.c
76.          |       simple.c
77.          |
78.          └──glparse
79.              |   resources.gltrace.gz
80.              |   simple.gltrace.gz
81.              |   triangle.gltrace.gz
82.              |   twocontexts.gltrace.gz
83.              |   wars.gltrace.gz
84.              |
85.              └──includes
86.                      resources.inc
87.                      simple.inc
```

```
88.                     triangle.inc
89.                     twocontexts.inc
90.                     wars.inc
```

2.3. General Approach

The provided solution is written in Python and C and consists of:

- `build.py`: Python driver script that invokes the other scripts in order to generate an APK from a `.gltrace` file, as represented in Figure 2
- `glparse.py`: Python script that generates a `trace.inc` C include file and associated assets from a `.gltrace` file
- `deinline.py`: Python script that refactors a `trace.inc` C include file result of `glparse.py`, into another `trace.inc` C include file
- C glue code to include `trace.inc` and generate an APK using the NDK and the ANT build system.

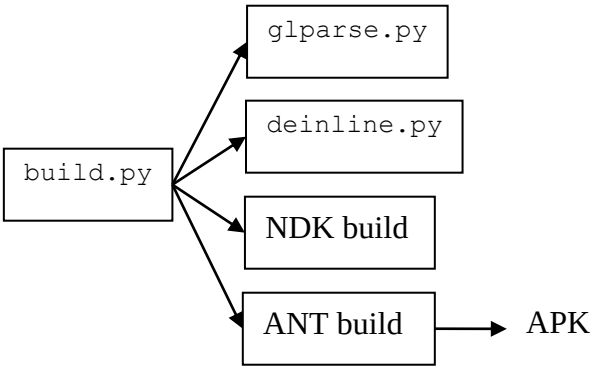

Figure 2: Build.py execution flow

Besides the above scripts, a set of small batch files were also written to simplify invoking the scripts above.

2.3.1. Design Principles

The number one design principle is to Keep It Simple:

- reuse existing components and data sources
- use an expressive language that minimizes code complexity
- make tools (compilers, etc.) work rather than introduce complex data structures.

Performance of the trace execution is important too, since it shouldn't introduce unnecessary CPU overhead that hinders the GPU performance, but optimizing the time and memory it takes to generate the trace is secondary.

2.4. Tools Used

It's worth mentioning a few existing tools that were used in the development of this project.

2.4.1. Android SDK and NDK

The Android SDK and NDK, described under Android Application Development, were used to compile the generated code and build an APK to run on an Android device.

2.4.2. Git

Git is a version control system written by Linus Torvalds in 2005[42]. It's focused on distributed development, without a centralized server.

2.4.3. Python

Python is an interpreted, interactive and object oriented language created by Guido van Rossum in 1989[43].
It has an extense standard library and multiple supplemental packages available on the *Python Package Index* PyPI[44].

2.4.3.1. Scriptine

Scriptine[45] is a Python package that simplifies using Python as a shell script. It uses introspection to expose Python functions as if they were command line options, function parameters as command line flags, and function documentation as command line help.

2.4.3.2. RunSnakeRun

As seen in Figure 3, RunSnakeRun[46] is a GUI viewer for Python's cProfile information using "box style" function timings that offers a visual way of finding hotspots in the Python code.

This kind of visualization is very useful to quickly understand what code to optimize, since the bigger the function's box, the more time is spent in that function.

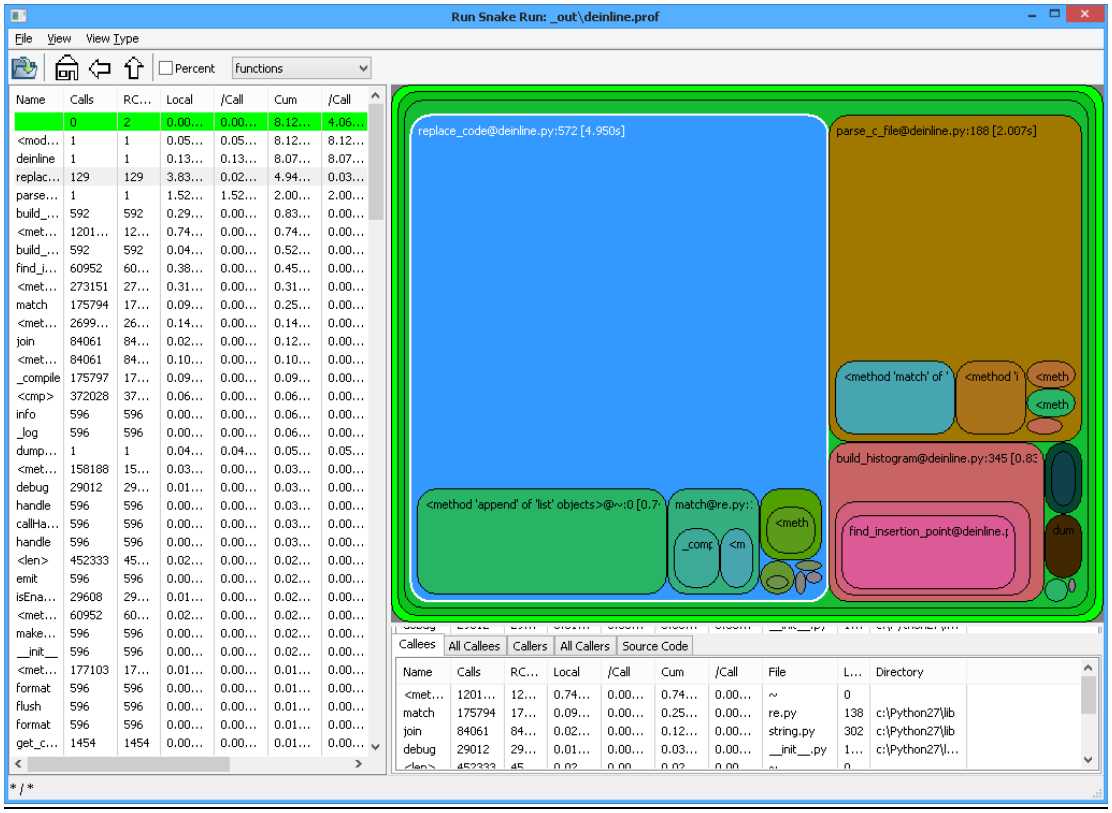

Figure 3: RunSnakeRun profiling deinline.py

2.4.3.3. Nose

Nose[47] is a testing framework for Python that supports test discovery, multiprocess parallel execution, code coverage and many other features.

2.4.3.4. Coverage

Coverage[48] is a tool to measure Python code coverage by test execution.

2.4.4. Word 2003

This document was written in Word 2003 and a *Visual Basic for Applications* script was also developed in order to add bibliographic references, since Word 2003 doesn't have that functionality.

The script is triggered by a keyboard shortcut and takes care of:

1. Displaying a dialog box requesting the text for the bibliographic entry,
2. Adding the new entry at the end of the document,
3. Walking the document's list of all the references, inside the document, collecting references and bibliographic entries,
4. Adding a reference to that entry in the current text position, and
5. Renumbering all later references in the document so bibliographic references appear monotonically numbered.

2.4.5. Excel 2003

The graphs in this document were done in Excel and are also available on a standalone Excel spreadsheet.

2.5. Trace Recording

Building a trace recorder is out of the scope of this work, so for this purpose, following the design principle of using existing tools, *Android's Tracer for OpenGL ES* is used.

2.5.1. Android Tracer for OpenGL ES

Android Tracer for OpenGL ES[40] is a tool part of the Android SDK that allows running applications in a special mode that traces all the OpenGL ES issued commands.

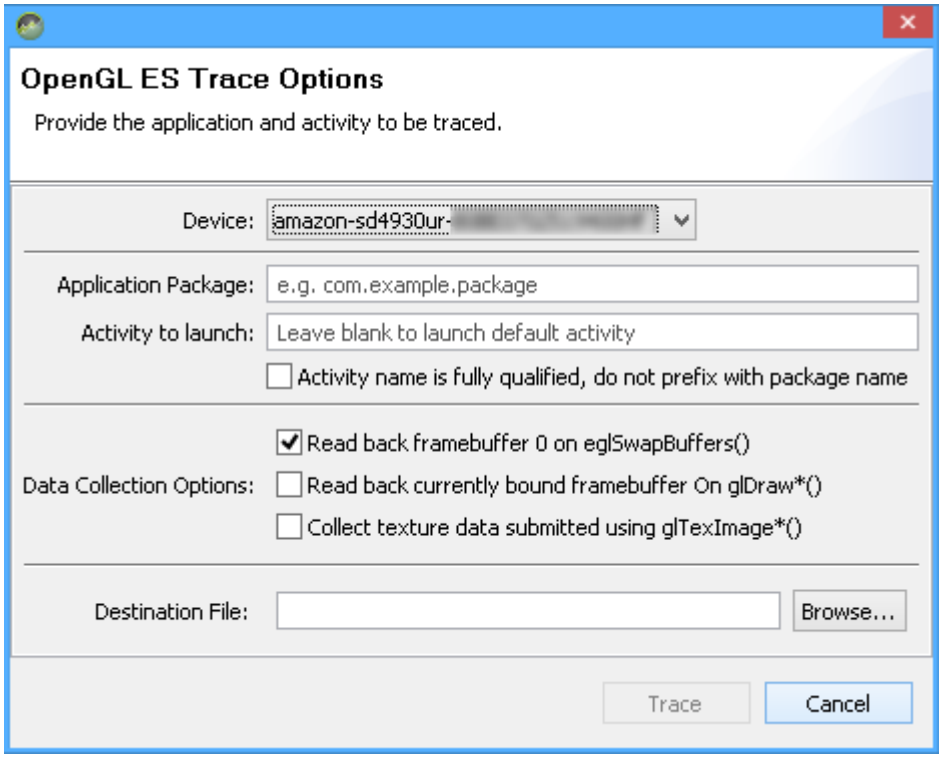

Figure 4: OpenGL ES Trace Options

The underlying architecture is that of a client-server[49], where Android's OpenGL ES library the application links against, connects via a Unix domain socket to a server and provides the OpenGL ES graphics commands and parameters as they are issued by the application. The traced commands can then be output to Android's log `logcat` or to a binary `.gltrace` file.

The tool allows selecting several capture options, in addition to capturing the OpenGL ES commands and parameters:

- capture the texture data
- capture the contents of the backbuffer
- capture the contents of the currently bound drawbuffer

The capture can be started from the GUI as shown in Figure 4 or from the command line, generating the aforementioned `.gltrace` file.

Android's Device Monitor is able to load `.gltrace` files and display the function calls, parameters, textures, captured buffers and current OpenGL ES state as can be seen in Figure 5

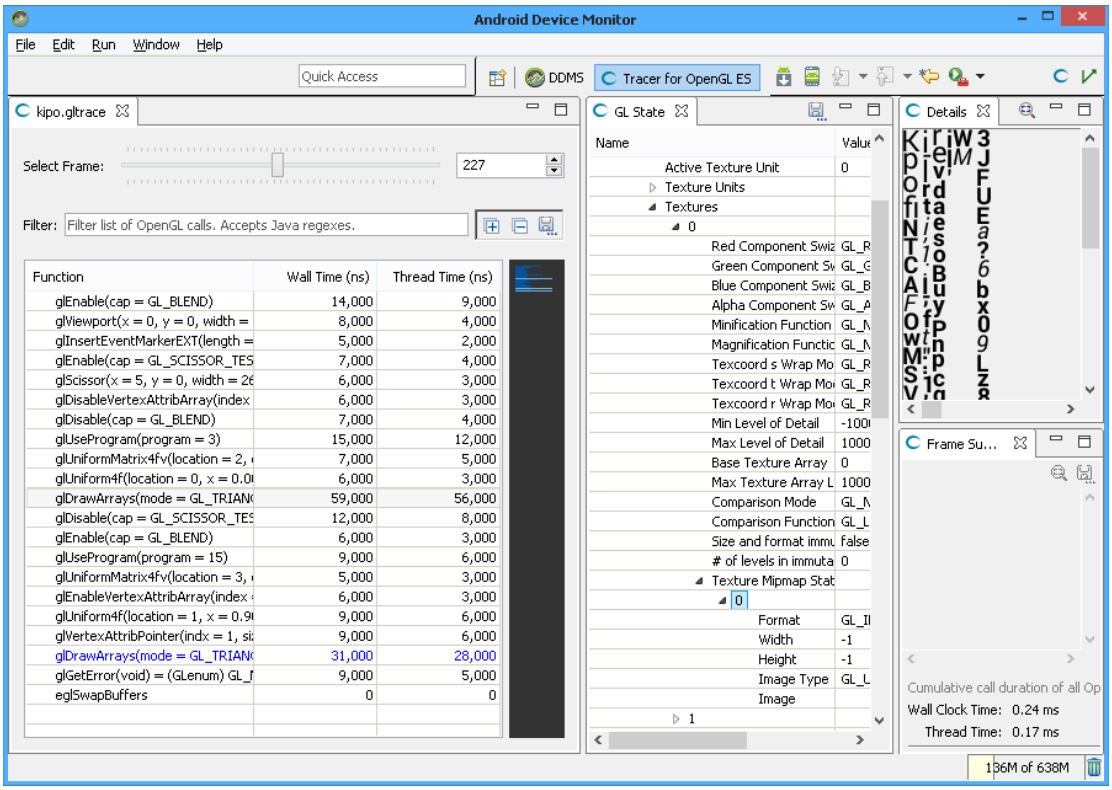

Figure 5: Android Device Monitor displaying an OpenGL ES trace

2.5.1.1. glcap.py

In order to automate trace capturing, instead of manually using the Android Device Monitor GUI tool, a Python script `glcap.py` was created.

The script connects via ADB forwarding to the Unix domain socket of an application started in OpenGL ES trace capture mode and captures the trace.

Besides extracting traces from existing OpenGL ES applications, this tool was also fundamental in order to automate the creation of tests:

1. A `trace.inc` file can be written by hand to exercise the desired OpenGL ES function calls,
2. An APK is generated from that `trace.inc` file using `build.py`,
3. Via `glcap.py`, the APK is run in OpenGL ES trace mode and a `.gltrace` file captured, and
4. The `.gltrace` file can now be used as part of the automated testing, since running `build.py` on that `.gltrace` should generate something close to the initial `trace.inc` file

This process is what the developed `generate_gltrace_from_inc.bat` script does.

2.5.1.2. Google's Protobuf

The format of an Android Tracer for OpenGL ES trace, or `.gltrace` file, is a protobuff[50] file that can be parsed using any protobuff library available.

For example, for the following OpenGL ES code

```
1.      GLfloat unif[3] = {1.0f, 2.0f, 3.0f };
2.      glUniform3fv(attr_location, 1, unif);
```

the `.gltrace` file contains the following protobuff entry

```
1.      start_time: 517898951235274
2.      duration: 13000
3.      function: glUniform3fv
4.      args {
5.        type: INT
6.        isArray: false
7.        intValue: 0
8.      }
9.      args {
10.       type: INT
11.       isArray: false
12.       intValue: 1
13.     }
14.     args {
15.       type: FLOAT
16.       isArray: true
17.       floatValue: 1.0
18.       floatValue: 2.0
19.       floatValue: 3.0
20.       int64Value: 1597765408
21.     }
22.     threadtime: 10000
```

2.6. Trace Code Generator

The below diagram shows the interactions between the C code generator script `glparse.py` and the different input and output files.

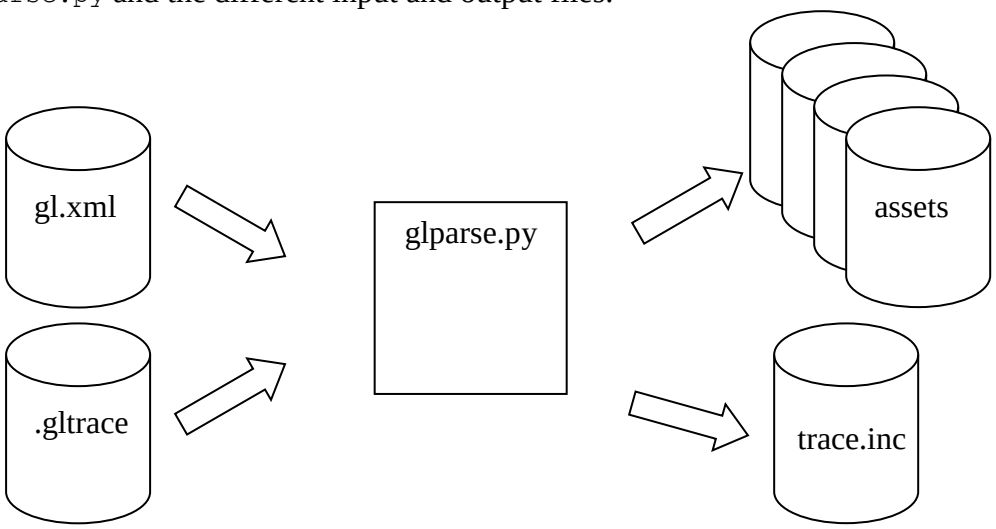

Figure 6: glparse.py data flow

Essentially, generating the code for a binary OpenGL ES trace is creating a C include file `trace.inc` by reading each entry of a `.gltrace` file trace, and generating C lines with the OpenGL ES function call and actual parameters for that entry.

The file `gl.xml` comes from the Khronos Group and is an optional input, as described in Numeric Literal to OpenGL ES Enumerant Translation.

Finally, `glparse.py` also generates a directory with `assets`, or binary data coming from the `.gltrace` file that is serialized to disk to be read at runtime, as explained in Non-scalar Parameters.

For convenience, the script bundles all the generated code in between `eglSwapBuffers` calls, into a `frameN` function.

Despite the simplicity of the general task, there are a few details that need to be considered when generating the C code. The following sections describe those subtleties.

2.6.1. Numeric Literal to OpenGL ES Enumerant Translation

The binary `.gltrace` file contains only numeric literals (e.g. `0x1908` instead of `GL_RGBA`), so a straight code generation, although functionally correct, is not suitable for human perusal:

```
1.      glBufferData(0x8892, 36, local_const_char_ptr_19, 0x88e4);
2.      glVertexAttribPointer(0, 3, 0x1406, 0, 0, (GLvoid*) 0x0);
3.      glEnableVertexAttribArray(0);
4.      glBindBuffer(0x8892, 0);
5.      glClear(16384);
6.      glUseProgram(global_unsigned_int_7);
7.      glDrawArrays(0x4, 0, 3);
```

It is therefore desirable to substitute the numeric literals with the OpenGL ES alphanumeric enumerants at code generation time:

```
1.      glBufferData(GL_ARRAY_BUFFER, 36, local_const_char_ptr_19, GL_STATIC_DRAW);
2.      glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, (GLvoid*) 0x0);
3.      glEnableVertexAttribArray(0);
4.      glBindBuffer(GL_ARRAY_BUFFER, 0);
5.      glClear(GL_COLOR_BUFFER_BIT);
6.      glUseProgram(global_unsigned_int_7);
7.      glDrawArrays(GL_TRIANGLES, 0, 3);
```

In order to generate those human-readable constants, the script allows translating them using Khronos' XML files[51] like `gl.xml`, which are preloaded at script initialization time.

2.6.2. OpenGL ES Resource Name Translation

Whenever OpenGL ES resource names are used as parameters (e.g. texture names, buffer names, etc.), those will need to be translated to the ones created at trace replay time, since the parameter stored in the trace is the resource name created at trace recording time.

For example, for the code

```
1.      glGenTextures(&textureName, 1)
2.      glBindTexture(textureName, GL_TEXTURE_2D);
```

the trace's protobuff entry for `glBindTexture` would contain the `GLuint` value for `textureName` that was generated at trace recording time

```
1.      glBindTexture(2048, GL_TEXTURE_2D);
```

which won't match the one generated by `glGenTextures` at trace replay time.

A translation table is necessary so when the `GLuint` value associated to `textureName` appears in the trace (2048 in the example), it's translated to the value that was generated at replay time.

This done by storing the result value in a variable and keeping track of which variable contains the replay-time `GLuint` value for the given capture-time `GLuint` value:

```
1.      static GLint global_int_ptr_0[1] = {2048};
2.      ...
3.      glGenTextures(1, global_int_ptr_0);
4.      glBindTexture(GL_TEXTURE_2D, global_int_ptr_0);
```

In order to detect when that parameter translation is necessary, two constant tables in the script store information about:

- what functions create resource names,
- what functions require parameter translation from the resource name stored in the trace to the resource name created at trace replay time.

One special case of this resource name translation is when an array of untranslated resource names are passed to an API call. This happens, e.g. with the resource deletion functions `glDeleteTextures`, `glDeleteBuffers`, etc.

```
1.      glGenTextures(&textures, 20);
2.      ...
3.      glDeleteTextures(&textures 20);
```

In that case, the generated code needs to create a local array and initialize to the translated values, then pass that local array to the desired function

```
1.      static GLuint global_unsigned_int_ptr_9181[1] = {59};
2.      ...
3.      glGenTextures(global_unsigned_int_ptr_9181, 1);
4.      ...
5.      GLuint local_GLuint_ptr_18548[] = { global_unsigned_int_ptr_9181[0] };
6.      glDeleteTextures(1, local_GLuint_ptr_18548);
```

2.6.3. Non-scalar Parameters

OpenGL ES functions that take non-scalar parameters (shader strings, texture data, vertex buffers...), require storing the trace data in a variable before being used, since those cannot be passed as immediate parameters to a C function.

Depending on the size of the parameter, the variable contents are inlined in the code with a variable initialization

```
1.      float local_float_ptr_33[] = { 1.0, 2.0, 3.0 };
2.      glUniform3fv(global_unsigned_int_32, 1, local_float_ptr_33);
```

or stored and then read from disk as assets,

```
1.      openAndGetAssetBuffer(param_DrawState_ptr_0, "int_asset_4",
&global_AAsset_ptr_I, (const void**) &global_const_int_ptr_I);
2.      glTexImage2D(GL_TEXTURE_2D, 0, 6408, 16, 16, 0, GL_RGBA, GL_UNSIGNED_BYTE,
global_const_int_ptr_I);
```

The threshold to perform one or the other action is dependent on the size of the data and is configurable in the Python script..

A study of the best values of that threshold is outside of the scope of this work, but it has to balance several factors:

- speed of fetching from disk vs. fetching from a local variable,
- scarcity of stack allocated memory vs. dynamically allocated memory,
- data cache thrashing impact of using long local variables.

The current default values are:

- float arrays with more than 64 elements are stored on disk,
- byte arrays with more than 256 elements are stored on disk

Finally, when storing into asset files, `glparse.py` coalesces multiple occurrences of the same binary data into a single file. In traces coming from real applications, this reduces the assets directory to around 75% of the original size.

2.6.4. Single Static Assignment

The code generator follows a *Single Static Assignment* or SSA[52] approach for naming temporary variables (i.e. never tries to recycle a previous temporary variable) and relies on the compiler recycling the variables when necessary.

This has some benefits like the one explained in Local Variable Definition Relocation.

2.6.5. Backbuffer Dimensions

One important piece of data that is not available in the trace is the dimensions of the standard backbuffer, since applications don't create it explicitly via OpenGL ES or EGL calls, but obtain it implicitly from the operating system.

In order to be able to render at the fastest speed, the trace replayer has the option of ignoring the OS-provided backbuffer[53] and creating an offscreen buffer instead (see Onscreen vs. Offscreen Rendering). A heuristic is used where the biggest scissor rectangle or viewport found in the trace is used as backbuffer dimensions[1].

[1] This heuristic is not very important, since the resolution at which to run the trace is normally passed from the command line, since it's usual to run a trace at different resolutions in order to find fill-rate vs. texture rate bottlenecks, etc

2.6.6. Viewport and Scissor Rectangles Override

As mentioned earlier, the trace can be replayed at an arbitrary resolution. For the rendering to be correct, `glViewport` and `glScissor` calls are intercepted at replay time in the glue code so they are properly scaled to the replayed dimensions.

These appear in the generated trace as the fake OpenGL ES calls `glScaledViewport` and `glScaledScissor`.

2.6.7. glVertexAttribPointerData

The Android OpenGL ES Tracer inserts vertex data via the fake OpenGL function function `glVertexAttribPointerData`[54]. The trace replayer implements this function in the glue code ultimately calling `glVertexAttribPointer`.

2.6.8. Example

The following code corresponds to the third frame of the `kipo.gltrace.gz` trace.
The trace performs the actions indicated previously, namely

- OpenGL ES function calls in between `eglSwapBuffer` calls are bundled inside a `FrameNN` function.
- The literal constants have been translated to OpenGL ES Enum names
- Some data is passed as assets via the function `openAndGetAssetBuffer` and the file name.
- Some data is embedded as local variables.
- References to OpenGL ES resource names have been exchanged with the variable they are stored into at trace replay time.
- Local variable names follow an strictly monotonically increasing naming convention.

```
1.      void frame3(AAssetManager* param_AAssetManager_ptr_0)
2.      {
3.          glEnable(GL_BLEND);
4.          glScaledViewport(0, 0, 800, 1205);
5.          GLchar local_char_ptr_241[] = "Flush";
6.          glInsertEventMarkerEXT(0, local_char_ptr_241);
7.          glEnable(GL_SCISSOR_TEST);
8.          glScaledScissor(0, 832, 795, 265);
9.          glDisableVertexAttribArray(1);
10.         glDisable(GL_BLEND);
11.         glUseProgram(global_unsigned_int_18);
12.         float local_float_ptr_242[] = { 800.0, 0.0, 0.0, 0.0, 0.0, 1205.0, 0.0,
0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0 };
13.         glUniformMatrix4fv(global_unsigned_int_38, 1, GL_FALSE,
local_float_ptr_242);
14.         glUniform4f(global_unsigned_int_44, 0.133333340287, 0.133333340287,
0.133333340287, 1.0);
15.         glBindBuffer(GL_ARRAY_BUFFER, global_int_ptr_3[0]);
16.         glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 16, (GLvoid*) 0x0);
17.         glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
18.         glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
19.         glDisable(GL_SCISSOR_TEST);
20.         float local_float_ptr_243[] = { 260.0, 0.0, 0.0, 0.0, 0.0, 260.0, 0.0, 0.0,
0.0, 0.0, 1.0, 0.0, 5.0, 113.0, 0.0, 1.0 };
21.         glUniformMatrix4fv(global_unsigned_int_38, 1, GL_FALSE,
local_float_ptr_243);
22.         glUniform4f(global_unsigned_int_44, 0.0, 0.0, 0.0, 1.0);
23.         glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
24.         glEnable(GL_BLEND);
25.         glUseProgram(global_unsigned_int_146);
```

```
26.         float local_float_ptr_244[] = { 260.0, 0.0, 0.0, 0.0, 0.0, 73.0, 0.0, 0.0,
0.0, 0.0, 1.0, 0.0, 5.0, 300.0, 0.0, 1.0 };
27.         glUniformMatrix4fv(global_unsigned_int_166, 1, GL_FALSE,
local_float_ptr_244);
28.         glUniform4f(global_unsigned_int_172, 0.266666680574, 0.266666680574,
0.266666680574, 0.666666686535);
29.         glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
30.         glDisable(GL_BLEND);
31.         glUseProgram(global_unsigned_int_18);
32.         float local_float_ptr_245[] = { 260.0, 0.0, 0.0, 0.0, 0.0, 260.0, 0.0, 0.0,
0.0, 0.0, 1.0, 0.0, 270.0, 113.0, 0.0, 1.0 };
33.         glUniformMatrix4fv(global_unsigned_int_38, 1, GL_FALSE,
local_float_ptr_245);
34.         glUniform4f(global_unsigned_int_44, 0.0, 0.0, 0.0, 1.0);
35.         glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
36.         glEnable(GL_BLEND);
37.         glUseProgram(global_unsigned_int_146);
38.         float local_float_ptr_246[] = { 260.0, 0.0, 0.0, 0.0, 0.0, 96.0, 0.0, 0.0,
0.0, 0.0, 1.0, 0.0, 270.0, 277.0, 0.0, 1.0 };
39.         glUniformMatrix4fv(global_unsigned_int_166, 1, GL_FALSE,
local_float_ptr_246);
40.         glUniform4f(global_unsigned_int_172, 0.266666680574, 0.266666680574,
0.266666680574, 0.666666686535);
41.         glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
42.         glDisable(GL_BLEND);
43.         glUseProgram(global_unsigned_int_18);
44.         float local_float_ptr_247[] = { 260.0, 0.0, 0.0, 0.0, 0.0, 260.0, 0.0, 0.0,
0.0, 0.0, 1.0, 0.0, 535.0, 113.0, 0.0, 1.0 };
45.         glUniformMatrix4fv(global_unsigned_int_38, 1, GL_FALSE,
local_float_ptr_247);
46.         glUniform4f(global_unsigned_int_44, 0.0, 0.0, 0.0, 1.0);
47.         glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
48.         glEnable(GL_BLEND);
49.         glUseProgram(global_unsigned_int_146);
50.         float local_float_ptr_248[] = { 260.0, 0.0, 0.0, 0.0, 0.0, 50.0, 0.0, 0.0,
0.0, 0.0, 1.0, 0.0, 535.0, 323.0, 0.0, 1.0 };
51.         glUniformMatrix4fv(global_unsigned_int_166, 1, GL_FALSE,
local_float_ptr_248);
52.         glUniform4f(global_unsigned_int_172, 0.266666680574, 0.266666680574,
0.266666680574, 0.666666686535);
53.         glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
54.         glUseProgram(global_unsigned_int_98);
55.         float local_float_ptr_249[] = { 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0,
0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0 };
56.         glUniformMatrix4fv(global_unsigned_int_125, 1, GL_FALSE,
local_float_ptr_249);
57.         glEnableVertexAttribArray(1);
58.         glUniform4f(global_unsigned_int_133, 1.0, 1.0, 1.0, 1.0);
59.         glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, global_int_ptr_83[0]);
60.         glBindBuffer(GL_ARRAY_BUFFER, 0);
61.         glBindTexture(GL_TEXTURE_2D, global_int_ptr_9[0]);
62.         glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 16, (GLvoid*) 0x685f1008);
63.         glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 16, (GLvoid*) 0x685f1010);
64.         closeAsset(global_AAsset_ptr_0);
65.         global_AAsset_ptr_0 = NULL;
66.         global_const_int_ptr_0 = NULL;
67.         openAndGetAssetBuffer(param_AAssetManager_ptr_0, "int_asset_250",
&global_AAsset_ptr_0, (const void**) &global_const_int_ptr_0);
68.         glVertexAttribPointerData(0, 2, GL_FLOAT, GL_FALSE, 16,
global_const_int_ptr_0, 0, 716);
69.         closeAsset(global_AAsset_ptr_1);
70.         global_AAsset_ptr_1 = NULL;
71.         global_const_int_ptr_1 = NULL;
72.         openAndGetAssetBuffer(param_AAssetManager_ptr_0, "int_asset_251",
&global_AAsset_ptr_1, (const void**) &global_const_int_ptr_1);
73.         glVertexAttribPointerData(1, 2, GL_FLOAT, GL_FALSE, 16,
global_const_int_ptr_1, 0, 716);
74.         glDrawElements(GL_TRIANGLES, 1074, GL_UNSIGNED_SHORT, (GLvoid*) 0x0);
75.         glEnable(GL_SCISSOR_TEST);
76.         glUseProgram(global_unsigned_int_52);
77.         float local_float_ptr_252[] = { 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0,
0.0, 0.0, 1.0, 0.0, 0.0, 108.0, 0.0, 1.0 };
78.         glUniformMatrix4fv(global_unsigned_int_76, 1, GL_FALSE,
local_float_ptr_252);
79.         glBindTexture(GL_TEXTURE_2D, global_int_ptr_8[0]);
80.         glBindBuffer(GL_ARRAY_BUFFER, global_int_ptr_5[0]);
81.         glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 16, (GLvoid*) 0x480);
82.         glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 16, (GLvoid*) 0x488);
```

```
83.         glDrawElements(GL_TRIANGLES, 18, GL_UNSIGNED_SHORT, (GLvoid*) 0x0);
84.         glGetError();
85.     }
```

2.7. Trace Code Auto-Refactoring

One of the drawbacks of a trace is the sheer size of it: hundreds of frames, each frame with hundreds of OpenGL ES function calls, each function call with the included parameter data (texture data, vertex data...).

Generating one or more lines of code per OpenGL ES function call in the trace can exceed compiler limits (code generators that output Java language instructions can exceed Dalvik's method limits[55]), increase the compile times and, more importantly for our profiling purposes, blow the instruction cache at execution time.

Additionally, generating one or more lines per OpenGL ES function call in the trace, makes modification of the generated code for experimentation more cumbersome, since now there are many places where the code needs to be modified, as opposed to the original code which may have the given feature refactored a single function.

Finally, reducing the generated code size also has other benefits like shorter compile times or, as mentioned before, even compiling at all, since long source files can cause compiler out of memory conditions.

It is, then, convenient to find away of *compressing* somehow the generated code.

The proposed way of performing that compression is by automatically *refactoring* or *deinlining* the code.

2.7.1. Algorithm

In order to refactor code, we present a novel algorithm that takes a simple approach where it tries to minimize the number of C lines in the final code, without caring how complex the C line is[2].
For simplicity, it also assumes that each C line is a simple instruction composed by a single function call.

Figure 7 shows the high level refactoring algorithm, where union $\cup$ denotes sequence concatenation and |N| denotes the cardinality or length of the sequence or set N.

$\text{Let } G = \{F_1, F_2, ..., F_n\} \text{ be a program composed by } n \text{ functions}$

$\text{Let } F_i = \{FP_i, L_i\} \text{ be a function composed by}$

- n formal parameters $FP_i = \{FP_{i,1}, ..., FP_{i,n}\}$
- m lines $L_i = \{L_{i,1}, ..., L_{i,m}\}$

[2] Note that this is not a big limitation, since the C lines come from the trace code generator, and the generator could be modified so all the generated C lines have similar complexity.
Additionally, if function call overhead is a concern, the compression factor formula can be easily modified so the compression factor of blocks of lines with less than a given number of lines is zero.

Let $L_{i,j} = \left\{ C_{i,j}, AP_{i,j,1}, AP_{i,j,2}, ..., AP_{i,j,n} \right\}$ be the j-th line from function F_i, containing

- a single function call $C_{i,j}$
- n actual parameters $AP_{i,j} = \left\{ AP_{i,j,1}, ..., AP_{i,j,n} \right\}$

Let $B_{i,j,k} = \left\{ C_{i,j}, .., C_{i,j+k-1} \right\}$ be the contiguous block of k function calls (ignoring the actual parameters) from lines $L_{i,j}$ to $L_{i,j+k-1}$ and function F_i

Let $S^v_{i,j,k} = \left\{ S^v_{i,j,k,1}, ..., S^v_{i,j,k,n} \right\}$ be

- the start line numbers of all non-overlapping occurrences of block $B_{i,j,k}$ in F_v, or
- empty if $B_{i,j,k}$ doesn't occur in F_v

Note that if $B_{i,j,k}$ occurs in F_v, then $B_{v,S^v_{i,j,k,n},k} = B_{i,j,k} \; \forall n \in \left[1, \left| S^v_{i,j,k} \right| \right]$

Build $\tilde{L}^v_{i,j,k} = \left\{ C^v, AP_{i,j,1}, ..., AP_{i,j,\left|AP_{i,j}\right|}, AP_{i,j+1,1}, ..., AP_{i,j+1,\left|AP_{i,j+1}\right|}, ..., AP_{i,j+k-1,1}, ..., AP_{i,j+k-1,\left|AP_{i,j+k}\right|} \right\}$ as the line resulting of making a function call C^v to F_v using all the actual parameters inside block $B_{i,j,k}$ as actual parameters to F_v

Build $\hat{L}^v_{i,j,k} = \left\{ C_{i,j}, FP_{v,k}, ..., FP_{v,k+\left|AP_{i,j}\right|-1} \right\}$ as the j-th line of function F_i, reparameterized to use F_v formal parameters as actual parameters, starting with formal parameter k

function Refactor(G)

forever

$$B_{p,q,r} = B_{p,q,r} / \text{CFactor}(p,q,r) \le \text{CFactor}(i,j,k) \forall i,j,k$$

if $\text{CFactor}(B_{p,q,r}) \le 0$ **then break**

$$FP'_i = \begin{cases} \left\{ FP_{|G|+1,1}, ..., FP_{|G|+1, \sum_{j=q}^{q+r-1} |AP_{p,j}|} \right\} & \textbf{if } i == |G|+1 \\ FP_i & \textbf{otherwise} \end{cases}$$

$$L'_i = \begin{cases} L_i & \textbf{if } S^i_{p,q,r} == \varnothing \\ \left\{ \bigcup_{j=q}^{q+r-1} \hat{L}^{|G|+1}_{p,j, \sum_{k=q}^{j-1} |AP_{p,k}|} \right\} & \textbf{if } i == |G|+1 \\ \left\{ L_{i,1}, .., L_{i,S^i_{p,q,r,1}-1} \right\} \cup \left\{ \bigcup_{j=1}^{|S^i_{p,q,r}|-1} \tilde{L}^{|G|+1}_{p,q,r}, L_{i,S^i_{p,q,r,j}+r} .., L_{i,S^i_{p,q,r,j+1}-1} \right\} \cup \left\{ \tilde{L}^{|G|+1}_{p,q,r}, L_{i,S^i_{p,q,r,|S^i_{p,q,r}|}+r}, .., L_{i,|L_i|} \right\} & \textbf{otherwise} \end{cases}$$

$$F'_i = \left\{ FP'_i, L'_i \right\}$$

$$G \leftarrow \left\{ F'_1, ..., F'_{|G|}, F'_{|G|+1} \right\}$$

return G

function CFactor(i, j, k)

return $\left(\left(\sum_{v=1}^{|G|} \left| S^v_{i,j,k} \right| \right) \cdot k - \left(\sum_{v=1}^{|G|} \left| S^v_{i,j,k} \right| \right) - k \right)$

Figure 7: Refactoring Algorithm

In essence, the algorithm, at every iteration:

1. finds the block of r lines $L_{p,q}$ to $L_{p,q+r-1}$ that currently provides the highest *CFactor* or *compression factor*,
2. creates a new function $F_{|G|+1}$ containing the r lines $\hat{L}^{|G|+1}_{i,j,k}$ to $\hat{L}^{|G|+1}_{i,j,k}$, i.e. the block of lines from $L_{p,q}$ to $L_{p,q+r-1}$ but reparameterized to use the new function's $F_{|G|+1}$ formal parameters, and
3. replaces every occurrence $S^v_{i,j,k}$ of that block of lines, with a single line $\tilde{L}^v_{i,j,k}$ that invokes the new function

Where *CFactor* or the *compression factor* of a block of k lines starting in line j and ending in line $j+k-1$ of function F_i, where the block is found in $\sum_{v=1}^{|G|} \left| S^v_{i,j,k} \right|$ non-overlapping occurrences in the whole program, is calculated as

$$\left(\sum_{v=1}^{|G|} \left| S^v_{i,j,k} \right| \right) \cdot k - \left(\sum_{v=1}^{|G|} \left| S^v_{i,j,k} \right| \right) - k$$

given that

- the refactoring saves as many lines as the block $B_{i,j,k}$ has, multiplied by the times it was found, $\left(\sum_{v=1}^{|G|}\left|S_{i,j,k}^{v}\right|\right)\cdot k$
- minus the lines necessary for the invocation of the function $F_{|G|+1}$ in all the occurrences, $\sum_{v=1}^{|G|}\left|S_{i,j,k}^{v}\right|$
- minus the lines necessary for the function's $F_{|G|+1}$ body, k

Note that finding the block with the highest compression factor is dependent on the functions being called inside the block, but independent on the specific actual parameters used when calling them.

In this sense, refactoring can be seen like a dictionary-based compression similar to LZW[56], where the dictionary is initialized with one identifier for each initial function F_i and a new entry is created in the dictionary for the new function created at each iteration of the algorithm.

By the previous definition of compression factor, a program that is refactored using the above algorithm will always contain less C lines after each iteration.

Once obvious that the algorithm finishes, there's the question of its optimality.

This algorithm follows a greedy approach (at each point it picks the block that currently has the highest compression factor), and it is known from data compression theory that choosing the optimum dictionary is an NP-hard problem[57]. Thus, a greedy approach may not offer the best solution in the long run[58][3].

One convenient advantage of this algorithm that can be easily overlooked, is that, by its cumulative nature, it will compress not only the original uncompressed code, but also any newly generated code result of compressing the original code.

Note that for our OpenGL ES trace use-case this refactoring scheme could also be applied to the original binary trace, rather than to the generated C code, with fake OpenGL ES function indices being created for the compressed blocks and some kind of tracking of the parameters used.

Applying the compression to the C source is more elegant since it does away with these fake OpenGL ES function indices and the parameter tracking, and with small changes would allow applying the scheme to other non-trace-related C sources.

[3] This can be intuitively understood by the fact that, if two options have the same compression factor, a greedy approach has no way of knowing which of the two will leave the program in a better state for the next refactoring.

2.7.2. Implementation

The following sections give insight into different implementation details of the refactoring algorithm.

2.7.2.1. Parsing Code

As an obligatory step previous to performing any refactoring, the code to be refactored needs to be read from the file generated by the trace parser.

Function definitions have to be found, and function calls and parameters to those function calls parsed, in order for the best block of instructions be extracted into a new function and its occurrences replaced by the appropriate function call.

For that purpose, the refactorer includes a simple C parsing algorithm, limited to parse code generated by the trace code generator.

Specifically, it doesn't handle complex code (e.g., member dereference operator, multiple statements in the same line, etc.), and requires all variables and function arguments to have decorated or mangled names so the type can be extracted from the variable name. Having type information for variables is necessary in case the variable later needs to be moved into a function and its value passed as a parameter.

The name mangling used by the trace code generator, follows a very simple scheme where variable names are composed piecewise by underscore-separated fields:

1. Zero or none `&` address-of operators
2. One of `global`, `local` or `param` scope identifiers
3. One or more type identifiers
4. Zero or more `ptr` pointer qualifiers
5. One numeric index
6. Zero or one `[n]` array dereference

The following lines show some examples of the allowed C constructs, using the decoration method defined above for the variable names:

```
1.      GLuint global_unsigned_int_ptr_1[1] = {32};
2.      void f(int param_int_ptr_0)
3.      {
4.          GLuint local_GLuint_0 = 0;
5.          f2(&local_GLuint_0, global_unsigned_int_ptr_1[0]);
6.          f3(0.1, GL_FALSE);
7.      }
```

By having the trace code generator use those decorations, the deinliner's parser can obtain the parameter scope (local, global, parameter, literal) and the type without having to do semantic analysis.

2.7.2.2. Finding the Block with the Largest Compression Factor

The literature for dictionary compression offers different techniques for finding the candidate to compress: McCreight[59] gives a linear time suffix trie construction,

other implementations use some adhoc searching strategy instead of a suffix trie, usually based on hashing[60].

In the current implementation, a *suffix array*[61] is created, with the following peculiarities:

- the symbols of the suffix array's alphabet are not really characters, but numeric identifiers assigned to each C function, and
- the location of the suffix inside the string is a tuple composed by the function where the suffix occurs and the offset into the string.

For example, in the following code

```
1.      void g1()
2.          f1(1,2,3);
3.          f2(4);
4.          f1(4,5,6);
5.          f2(5);
6.      void g2()
7.          f1(7,8,9);
8.          f2(7);
```

can be modeled as the following strings where `f1` is assigned the symbol `a` from the alphabet and `f2` is assigned the symbol `b` (note how parameters are ignored, since they are irrelevant to the problem of finding the largest compression factor):

```
1.      g1: abab
2.      g2: ab
```

In this case, the corresponding suffix array and 0-based indices to the occurrences of each suffix would be:

```
1.      a       g1:0, g1:1, g2:0
2.      ab      g1:0, g1:2, g2:0
3.      aba     g1:0
4.      abab    g1:0
5.      b       g1:1, g1:3
6.      ba      g1:1
7.      bab     g1:1
```

Linearly walking the array allows calculating the compression factor defined earlier for each suffix (suffix length * occurrences - suffix length - occurrences) :

```
1.      a       1 * 3 - 1 - 3 = -1
2.      ab      2 * 3 - 2 - 3 = 1
3.      aba     3 * 1 - 3 - 1 = -1
4.      abab    4 * 1 - 4 - 1 = -1
5.      b       1 * 2 - 1 - 2 = -1
6.      ba      2 * 1 - 2 - 1 = -1
7.      bab     3 * 1 - 3 - 1 = -1
```

Where only entry 2 in the suffix array is worth considering, since the other ones have negative compression factor (i.e. they don't compress).

Once the current iteration is done, the suffix array could be updated with the changes done, but in the current implementation is generated from scratch for simplicity's sake.

A simpler implementation using Python hashes over unicode strings is also included in the git repository. In that implementation each line of code is a represented as a

single unicode character (since ASCII characters would limit to 256 functions, which is too restrictive).

This simpler implementation is fast, but hashing all possible substrings consumes too much memory to be used with hundreds of complex frames.

2.7.2.3. Dealing with Overlapping Occurrences

Besides finding matching runs, we also need to ignore overlaps across occurrences when calculating the compression factor, since when occurrences overlap, only one of the occurrences can be refactored.

Given the following example:

```
1.      g1: ababa
2.      g2: aba
```

The corresponding suffix array would be

```
1.      a       g1:0, g1:1, g1:4, g2:0, g2:2
2.      ab      g1:0, g1:2, g2:0
3.      aba     g1:0, g1:2, g2:0
4.      abab    g1:0
5.      b       g1:1, g1:3
6.      ba      g1:1, g2:1
7.      bab     g1:1
8.      baba    g1:1
```

Note how the `g1:0` and `g1:2` occurrences for `aba` overlap. This is easily spotted because occurrences are sorted by the function where they appear, and then by the offset inside the function, so if two occurrences are in the same function and the offsets differ by less than the suffix length, then there's an overlap.

When walking linearly the array to calculate the compression factor, it's necessary to apply the above overlap detection to ignore the current occurrence if it overlaps the previous occurrence:

```
1.      a       1 * 3 - 1 - 3 = -1
2.      ab      2 * 3 - 2 - 3 = 1
3.      aba     3 * 2 - 3 - 2 = 1
4.      abab    4 * 1 - 4 - 1 = -1
5.      b       1 * 2 - 1 - 2 = -1
6.      ba      2 * 1 - 2 - 1 = -1
7.      bab     3 * 1 - 3 - 1 = -1
8.      baba    4 * 1 - 4 - 1 = -1
```

This shows another of the greedy features of the algorithm that prevent an optimal solution, not only it picks the currently best compression factor as explained earlier, but it also picks the first occurrence when there are overlapping ones, so there's no guarantee that the compression is optimal.

2.7.2.4. Sliding Window Suffix Array

The suffix array implementation in Python can become quite slow (around one hour for an 800-frame trace, see Deinline Performance), so a sliding window of size `window_size` approach is used where

- `window_start` is initialized to `1` and incremented on every iteration of the algorithm,
- `window_end` is initialized to `window_start + window_size` and incremented on every iteration of the algorithm,
- the suffix array is populated only with the functions F_i with i between `window_start` and `window_end`, and
- once the block to refactor is found, the block is replaced across all the program, not just between `window_start` and `window_end`.

For an analysis of different `window_size` values, refer to Deinline Performance.

Note that a C implementation of the suffix array would be fast enough for all purposes, since essentially the algorithm is doing gzip compression on a stream composed by the function indices (a tiny fraction of the trace file) and gzip's compression throughput has been benchmarked around 30MB/s on a current server CPU[62]

2.7.2.5. Code Replacing

Once the block with the largest compression factor has been found, the occurrence of each block has to be found and replaced.

This is just a matter of scanning the existing functions looking for the occurrences, replacing them, and adapting the actual parameters to invoke the new function (the details of which are explained in Parameter Handling and later subsections).

Instead of scanning for the occurrence, the suffix array information could be used to directly access the location of the occurrence to replace, but this ties the code replacing implementation to the specific method used to find the block with the best compression factor, which would prevent exploring other compression factor calculation algorithms (e.g. the aforementioned hash over unicode strings).

2.7.2.6. Parameter Handling

Once the block to factor out has been chosen, there remains the task of finding the formal parameters (for the new function definition) and the actual parameters (for the call sites into the new function).

An important observation already outlined in the algorithm is that, once the block B to refactor has been decided, there's always a combination of formal parameters that satisfies all the call sites, in particular,

- given a block B of N lines,
- with each line L_i being a function call with AP_i number of actual parameters,

then B can be refactored as a function that takes $\sum_{i=1}^{N} AP_i$ formal parameters.

For example, given the following code:

```
1.      void f1(param1_1, param1_2, param1_3)
2.      {
```

```
3.          f3(param1_1);
4.          f4(param1_2, param1_3);
5.          f5(param1_1);
6.          f6(param1_3, param1_2);
7.      }
8.
9.      void f2(param2_1, param2_2, param2_3, param2_4, param2_5)
10.     {
11.         f3(param2_1);
12.         f4(param2_2, param2_3);
13.         f5(param2_1);
14.         f6(param2_5, param2_4);
15.     }
```

it can be refactored by creating a function `g` that takes as many formal parameters as actual parameters (distinct or not) `f1` has in its body (or `f2`, since they are identical in that respect):

```
1.      void g(param3_1, param3_2, param3_3, param3_4, param3_5, param3_6)
2.      {
3.          f3(param3_1);
4.          f4(param3_2, param3_3);
5.          f5(param3_4);
6.          f6(param3_5, param3_6);
7.      }
8.
9.      void f1(param1_1, param1_2, param1_3)
10.     {
11.         g(param1_1, param1_2, param1_3, param1_1, param1_3, param1_2);
12.     }
13.
14.     void f2(param2_1, param2_2, param2_3, param2_4, param2_5)
15.     {
16.         g(param2_1, param2_2, param2_3, param2_1, param2_5, param2_4);
17.     }
```

Note that the refactoring was possible even if `f1` and `f2` themselves take different number of formal parameters, and even if they use different actual parameter layout in their bodies.

Note, as well, that this method is suboptimal with regards to minimizing the number of parameters, since for example both `f1` and `f2` pass the same actual parameter twice (`param_2_1` and `param1_1`) in two formal parameters (`param3_1` and `param3_4`), so `param3_4` could be removed from the parameter list and `param3_1` used solely instead:

```
1.      void g(param3_1, param3_2, param3_3, param3_5, param3_6)
2.      {
3.          f3(param3_1);
4.          f4(param3_2, param3_3);
5.          f5(param3_1);
6.          f6(param3_5, param3_6);
7.      }
8.
9.      void f1()
10.     {
11.         g(param1_1, param1_2, param1_3, param1_3, param1_2);
12.     }
13.
14.     void f2()
15.     {
16.         g(param2_1, param2_2, param2_3, param2_5, param2_4);
17.     }
```

This *parameter colaescing* technique and other techniques to reduce the parameter count are explained in subsequent sections.

2.7.2.7. Common Parameter Coalescing

A function with as many formal parameters as actual parameters has the refactored function calls in its body is a worst case scenario, and the current implementation actually does elimination of parameters that are the common across all call-sites, greatly reducing the number of necessary formal parameters.

In order to do that, as formal parameters for the new function are gathered from the block to be refactored, every call site is examined and, when all the call sites are found to use the same previously added formal parameter, that new parameter is ignored and the previously added used instead.

2.7.2.8. Identical Parameter Inlining

Many times, all the call-sites will use the same literal value in the same actual parameter, for example

```
1.      void f1(param_int_0, param_int_1)
2.      {
3.          a(param_int_0, param_int_1, param_int_0);
4.          b(1);
5.          c(param_int_0, param_int_1, param_int_0);
6.          d(3);
7.          e(4);
8.          f(5);
9.      }
10.
11.     void f2(param_int_0, param_int_1)
12.     {
13.         a(param_int_0, param_int_1, param_int_0);
14.         b(0);
15.         c(param_int_0, param_int_1, param_int_0);
16.         d(2);
17.         e(4);
18.         f(4);
19.     }
```

Note how `e(4)` is called in both call sites. This means that `4` doesn't need to be passed as a parameter an can be inlined across all invocations into the deinlined code:

```
1.      void f1(param_int_0, param_int_1)
2.      {
3.          subframe2(param_int_0, param_int_1, 1, 3, 5);
4.      }
5.
6.      void f2(param_int_0, param_int_1)
7.      {
8.          subframe2(param_int_0, param_int_1, 0, 2, 4);
9.      }
10.
11.     void subframe2(int param_int_0, int param_int_1, int param_int_2, int
param_int_3, int param_int_4)
12.     {
13.         a(param_int_0, param_int_1, param_int_0);
14.         b(param_int_2);
15.         c(param_int_0, param_int_1, param_int_0);
16.         d(param_int_3);
17.         e(4);
18.         f(param_int_4);
19.     }
```

This parameter inlining can be applied not only when literal values are used as parameters and are common across all call sites, but also when global variables are used in that way.

2.7.2.9. Aliased Parameter Coalescing

One non-obvious issue arises when refactoring code that modifies a pointer (aliasing variable) and dereferences it afterwards (aliased variable).

For example, in the code:

```
1.      glGenTextures(1, &id);
2.      glBindTexture(GL_TEXTURE_2D, id);
```

`glGenTexture` modifies `id` by reference and `glBindTexture` uses that modified value. Assuming that sequence of instructions is used in multiple places and worth refactoring, one naïve refactoring would be

```
1.      void subframe(pI, I)
2.      {
3.          glGenTextures(1, pI)
4.          glBindTexture(GL_TEXTURE_2D, I)
5.      }
```

Note how the refactoring created the function's formal parameters `pI` and `I` but, in the process, broke the aliasing that the original code had between `&id` and `id`, which results in incorrect code.

For resolving the case above, the deinliner coalesces `I` usages as `pI[0]`, and removes the `I` parameter altogether (i.e. coalesces the *aliased variable* into the *aliasing variable dereference*):

```
1.      void subframe(pI)
2.      {
3.          glGenTextures(1, pI)
4.          glBindTexture((...., ppI[0])
5.      }
```

Whenever possible, the deinliner will try to coalesce aliased variables into aliasing variable dereferences, but there are occasions where this is not possible, for example when some callers alias but some others do not:

```
1.      void f1()
2.      {
3.          glGenTextures(1, &id);
4.          glBindTexture(GL_TEXTURE_2D, id);
5.      }
6.
7.      void f2()
8.      {
9.          glGenTextures(1, &id):
10.         glBindTexture(GL_TEXTURE_2D, 0);
11.     }
```

In this case, `f2` is not suffering of any aliasing, but the deinliner will still deinline both functions into one, since the deinliner doesn't look at the function parameters, just at the sequence of functions.

For this case, a generic technique is used that chooses at runtime whether to alias or not, depending on a parameter provided by the caller, by inserting memory copy instruction between the aliasing and the aliased instructions:

```
1.      void subframe(pI, I, bytes)
2.      {
```

```
3.          glGenTextures(1, pI);
4.          memcpy(&I, pI, bytes);
5.          glBindTexture(GL_TEXTURE_2D, I);
6.      }
7.
8.      void f1()
9.      {
10.         subframe(&id, id, 4);
11.     }
12.
13.     void f2()
14.     {
15.         subframe(&id, 0, 0);
16.     }
```

The number of bytes to copy are passed as a parameter, with zero bytes to copy if the caller doesn't need aliasing resolution, or 4 if it needs it.

Note this technique gets more intricate when there are multiple aliasing and aliased relationships:

```
1.      void f1()
2.      {
3.          GLuint local_GLuint_0;
4.          GLuint local_GLuint_1;
5.
6.          glGenTextures(1, &local_GLuint_0);
7.          glGenTextures(1, &local_GLuint_1);
8.          glBindTexture(GL_TEXTURE_2D, local_GLuint_0);
9.      }
10.
11.     void f2()
12.     {
13.         GLuint local_GLuint_0;
14.         GLuint local_GLuint_1;
15.
16.         glGenTextures(1, &local_GLuint_0);
17.         glGenTextures(1, &local_GLuint_1);
18.         glBindTexture(GL_TEXTURE_2D, local_GLuint_1);
19.     }
```

In the example above, the aliasing instruction is criss-crossed across invocations:

- instruction 5 aliases instruction 7
- instruction 14 aliases instruction 15.

This is solved by inserting two memory copy instructions and passing the necessary extra parameters:

```
1.      void f1()
2.      {
3.          subframe18(&local_GLuint_0, &local_GLuint_1, local_GLuint_0, 0, 4);
4.      }
5.
6.      void f2()
7.      {
8.          subframe18(&local_GLuint_0, &local_GLuint_1, local_GLuint_1, 4, 0);
9.      }
10.
11.     void subframe18(GLuint * param_GLuint_ptr_0, GLuint * param_GLuint_ptr_1,
GLuint param_GLuint_2, int param_int_3, int param_int_4)
12.     {
13.         GLuint local_GLuint_0;
14.         GLuint local_GLuint_1;
15.         glGenTextures(1, param_GLuint_ptr_0);
16.         memcpy(&param_GLuint_2, param_GLuint_ptr_0, param_int_4);
17.         glGenTextures(1, param_GLuint_ptr_1);
18.         memcpy(&param_GLuint_2, param_GLuint_ptr_1, param_int_3);
19.         glBindTexture(GL_TEXTURE_2D, param_GLuint_2);
20.     }
```

The deinliner has an algorithm that exhaustively deals with all the possible cases by

- first, gathering information about the aliased/aliasing relationships. Due to the limitations of the deinliner's C parser and the lack of semantic analysis, all the aliasing resolution is done conservatively whenever the code finds that a variable's address and the variable itself are passed as parameters to a new deinlined function, and
- second, applying the generic technique inserting extra memory copies and parameters, or coalesces into pointer dereference depending on whether all the callers alias or not.

In addition to the above two aliasing resolution methods (coalescing and generic memory copy), there's a third method where the aliasing has previously been solved by global parameter inlining (as mentioned in Identical Parameter Inlining)

```
1.      void f1()
2.      {
3.          glGenTextures2(1, &global_GLuint_0);
4.          glBindTexture2(GL_TEXTURE_2D, global_GLuint_0);
5.      }
6.
7.      void f2()
8.      {
9.          glGenTextures2(1, &global_GLuint_0);
10.         glBindTexture2(GL_TEXTURE_2D, global_GLuint_0);
11.     }
```

In this case the global parameter inlining, converts the code into

```
1.      void f1()
2.      {
3.          subframe20();
4.      }
5.
6.      void f2()
7.      {
8.          subframe20();
9.      }
10.
11.     void subframe20()
12.     {
13.         glGenTextures2(1, &global_GLuint_0);
14.         glBindTexture2(GL_TEXTURE_2D, global_GLuint_0);
15.     }
```

The aliasing resolution algorithm, before applying any aliasing preserving technique, first verifies that the aliasing hasn't been already solved by global variable inlining.

2.7.2.10. Local Variable Definition Relocation

Asset inlining described in Non-scalar Parameters worsens the compression because variable definitions break sequences of instructions that could be otherwise factored out (or if assets were loaded from a file instead of inlined).

For example, given the following code

```
1.      void f1()
2.      {
3.          g1(0);
4.          int local_int_0 = 0;
```

```
5.          g2(local_int_0);
6.      }
7.
8.      void f2()
9.      {
10.         g1(0);
11.         g2(1);
12.     }
```

The inlined definition in line 4 unnecessarily prevents matching the bodies of `f1` and `f2`.

To help with that case, the simple parser described in Parsing Code detects variable definition and moves it to the beginning of the function

```
1.      void f1()
2.      {
3.          int local_int_0 = 0;
4.          g1(0);
5.          g2(local_int_0);
6.      }
7.
8.      void f2()
9.      {
10.         g1(0);
11.         g2(1);
12.     }
```

This no longer breaks the `g1`/`g2` sequence and allows for the following refactoring to be found:

```
1.      void f3(param_int_0)
2.      {
3.          g1(0);
4.          g1(param_int_0);
5.      }
6.
7.      void f1()
8.      {
9.          int local_int_0 = 0;
10.         f3(local_int_0);
11.     }
12.
13.     void f2()
14.     {
15.         f3(1);
16.     }
```

This is possible because of the Single Static Assignment approach used by the trace code generator, which prevents writes to a variable before the reader has been able to consume its value.
If SSA wasn't used, the following variable reuse case would prevent local variable relocation, since the initial value of `local_int_0` would be overwritten before it has been consumed by `g1`:

```
1.      void f1()
2.      {
3.          int local_int_0 = 0;
4.          g1(local_int_0);
5.          local_int_0 = 1;
6.          g2(local_int_1);
7.      }
```

2.7.2.11. Actual Parameter Type Casting

When a block of lines is refactored into a new function, the type of the formal parameters for the new function is obtained from the names of the actual parameters of the functions used inside the block (this is possible because the names are decorated with the parameter types, as explained in Parsing Code).

Because there's always more than one occurrence of the block in the code, one of the occurrences is picked, as seen in this example:

```
1.      void f()
2.      {
3.          glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 16, global_float_ptr_0);
4.      }
5.
6.      void g()
7.      {
8.          glVertexAttribPointer(1, 4, GL_UNSIGNED_BYTE, GL_TRUE, 36,
global_char_ptr_0);
9.      }
```

Which could be refactored as:

```
1.      void h(param_int_0, param_int_1, param_int_2, param_int_3, param_int_4,
param_float_ptr_0)
2.      {
3.          glVertexAttribPointer(param_int_0, param_int_1, param_int_2, param_int_3,
param_int_4, param_float_ptr_0);
4.      }
5.
6.      void f()
7.      {
8.          h(0, 2, GL_FLOAT, GL_FALSE, 16, global_float_ptr_0);
9.      }
10.
11.     void g()
12.     {
13.         h(1, 4, GL_UNSIGNED_BYTE, GL_TRUE, 36, global_char_ptr_0);
14.     }
```

Note how the formal parameter types for `h()` were taken from the occurrence of the block inside `f()`.
As highlighted in bold, this causes a type conflict because the newly created function uses `float*` as formal parameter type, but one of the invocations is passing `char*`.

When the code is compiled with strict warnings, this mixed pointer type causes compiler errors. The original unrefactored code compiles cleanly, because `glVertexAttribPointer` actually has `void*` as last formal parameter, which allows passing both `float*` and `char*` as actual parameters

In order to resolve this problem, when generating the code to call into the refactored function, the actual parameters have to be type casted to the formal parameters when those mismatch:

```
1.      void h(param_int_0, param_int_1, param_int_2, param_int_3, param_int_4,
param_float_ptr_0)
2.      {
3.          glVertexAttribPointer(param_int_0, param_int_1, param_int_2, param_int_3,
param_int_4, param_float_ptr_0);
4.      }
5.
6.      void f()
7.      {
8.          h(0, 2, GL_FLOAT, GL_FALSE, 16, global_float_ptr_0);
```

```
9.      }
10.
11.     void g()
12.     {
13.         h(1, 4, GL_UNSIGNED_BYTE, GL_TRUE, 36, (float*) global_char_ptr_0);
14.     }
```

Note the `char*` actual parameter is now coerced into a `float*`, which is not strictly correct, but this incorrectness is innocuous since the cast is only introduced to prevent the compiler error and it doesn't have any side effects.

2.7.3. Example

The following is an example from the third frame of kipo.gltrace.gz sample shown previously in this document.

- Refactored code is recursively refactored, not just the original code.
- Refactored functions are at least two C lines long (otherwise there's no win between the function body and the function call)
- Parameters are replaced properly and constant parameters removed from the refactored function and the call-sites.
- The code is reduced from a single 82-line function to 74 lines split across 17 functions[4] (in both cases not counting opening and closing braces or function prototypes).

```
1.      void subframe500(unsigned int param_unsigned_int_0, unsigned int
param_unsigned_int_1)
2.      {
3.          glDisable(param_unsigned_int_0);
4.          glUseProgram(param_unsigned_int_1);
5.      }
6.
7.      void subframe540(int param_int_0, int param_int_1, int param_int_2, int
param_int_3)
8.      {
9.          glEnable(GL_SCISSOR_TEST);
10.         glScaledScissor(param_int_0, param_int_1, param_int_2, param_int_3);
11.     }
12.
13.     void subframe480(char * param_char_ptr_0, int param_int_1, int param_int_2, int
param_int_3, int param_int_4)
14.     {
15.         glInsertEventMarkerEXT(0, param_char_ptr_0);
16.         subframe540(param_int_1, param_int_2, param_int_3, param_int_4);
17.         glDisableVertexAttribArray(1);
18.         subframe500(GL_BLEND, global_unsigned_int_18);
19.     }
20.
21.     void subframe587(unsigned int param_unsigned_int_0, float * param_float_ptr_1,
unsigned int param_unsigned_int_2, float param_float_3, float param_float_4, float
param_float_5, float param_float_6)
22.     {
23.         glUniformMatrix4fv(param_unsigned_int_0, 1, GL_FALSE, param_float_ptr_1);
24.         glUniform4f(param_unsigned_int_2, param_float_3, param_float_4,
param_float_5, param_float_6);
25.     }
26.
27.     void subframe515(unsigned int param_unsigned_int_0, float * param_float_ptr_1,
unsigned int param_unsigned_int_2, float param_float_3, float param_float_4, float
param_float_5, float param_float_6, unsigned int param_unsigned_int_7, int
param_int_8)
28.     {
29.         subframe587(param_unsigned_int_0, param_float_ptr_1, param_unsigned_int_2,
param_float_3, param_float_4, param_float_5, param_float_6);
```

[4] This is roughly a 15% compression for a single frame, but the final compression across the whole trace is more than 80%, because many of the functions factored out are common to most of the 461 frames.

```
30.         glBindBuffer(param_unsigned_int_7, param_int_8);
31.     }
32.
33.     void subframe530(unsigned int param_unsigned_int_0, float * param_float_ptr_1,
unsigned int param_unsigned_int_2, float param_float_3, float param_float_4, float
param_float_5, float param_float_6)
34.     {
35.         subframe515(param_unsigned_int_0, param_float_ptr_1, param_unsigned_int_2,
param_float_3, param_float_4, param_float_5, param_float_6, GL_ARRAY_BUFFER,
global_int_ptr_3[0]);
36.         glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 16, (GLvoid const*) 0x0);
37.         glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
38.         glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
39.         glDisable(GL_SCISSOR_TEST);
40.     }
41.
42.     void subframe587(unsigned int param_unsigned_int_0, float * param_float_ptr_1,
unsigned int param_unsigned_int_2, float param_float_3, float param_float_4, float
param_float_5, float param_float_6)
43.     {
44.         glUniformMatrix4fv(param_unsigned_int_0, 1, GL_FALSE, param_float_ptr_1);
45.         glUniform4f(param_unsigned_int_2, param_float_3, param_float_4,
param_float_5, param_float_6);
46.     }
47.
48.     void subframe463(unsigned int param_unsigned_int_0, float * param_float_ptr_1,
unsigned int param_unsigned_int_2, float param_float_3, float param_float_4, float
param_float_5, float param_float_6)
49.     {
50.         subframe587(param_unsigned_int_0, param_float_ptr_1, param_unsigned_int_2,
param_float_3, param_float_4, param_float_5, param_float_6);
51.         glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
52.     }
53.
54.     void subframe506(unsigned int param_unsigned_int_0)
55.     {
56.         glEnable(GL_BLEND);
57.         glUseProgram(param_unsigned_int_0);
58.     }
59.
60.     void subframe464(float * param_float_ptr_0, unsigned int param_unsigned_int_1)
61.     {
62.         subframe463(global_unsigned_int_38, param_float_ptr_0,
global_unsigned_int_44, 0.0, 0.0, 0.0, 1.0);
63.         subframe506(param_unsigned_int_1);
64.     }
65.
66.     void subframe465(unsigned int param_unsigned_int_0, float * param_float_ptr_1,
unsigned int param_unsigned_int_2, float param_float_3, float param_float_4, float
param_float_5, float param_float_6, unsigned int param_unsigned_int_7, unsigned int
param_unsigned_int_8)
67.     {
68.         subframe463(param_unsigned_int_0, param_float_ptr_1, param_unsigned_int_2,
param_float_3, param_float_4, param_float_5, param_float_6);
69.         subframe500(param_unsigned_int_7, param_unsigned_int_8);
70.     }
71.
72.     void subframe531(unsigned int param_unsigned_int_0, float * param_float_ptr_1,
unsigned int param_unsigned_int_2, float param_float_3, float param_float_4, float
param_float_5, float param_float_6, unsigned int param_unsigned_int_7)
73.     {
74.         subframe463(param_unsigned_int_0, param_float_ptr_1, param_unsigned_int_2,
param_float_3, param_float_4, param_float_5, param_float_6);
75.         glUseProgram(param_unsigned_int_7);
76.     }
77.
78.     void subframe484(float * param_float_ptr_0)
79.     {
80.         glUniformMatrix4fv(global_unsigned_int_125, 1, GL_FALSE,
param_float_ptr_0);
81.         glEnableVertexAttribArray(1);
82.         glUniform4f(global_unsigned_int_133, 1.0, 1.0, 1.0, 1.0);
83.         glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, global_int_ptr_83[0]);
84.         glBindBuffer(GL_ARRAY_BUFFER, 0);
85.     }
86.
87.     void subframe544(unsigned int param_unsigned_int_0, unsigned int
param_unsigned_int_1, AAssetManager * param_AAssetManager_ptr_2, char *
```

```
param_char_ptr_3, int param_int_4, AAssetManager * param_AAssetManager_ptr_5, char *
param_char_ptr_6)
88.     {
89.         glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 16, (GLvoid const*)
param_unsigned_int_0);
90.         glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 16, (GLvoid const*)
param_unsigned_int_1);
91.         closeAsset(global_AAsset_ptr_0);
92.         global_AAsset_ptr_0 = NULL;
93.         global_const_int_ptr_0 = NULL;
94.         openAndGetAssetBuffer(param_AAssetManager_ptr_2, param_char_ptr_3,
&global_AAsset_ptr_0, &global_const_int_ptr_0);
95.         glVertexAttribPointerData(0, 2, GL_FLOAT, GL_FALSE, 16,
global_const_int_ptr_0, 0, param_int_4);
96.         closeAsset(global_AAsset_ptr_1);
97.         global_AAsset_ptr_1 = NULL;
98.         global_const_int_ptr_1 = NULL;
99.         openAndGetAssetBuffer(param_AAssetManager_ptr_5, param_char_ptr_6,
&global_AAsset_ptr_1, &global_const_int_ptr_1);
100.    }
101.
102.    void subframe501(float * param_float_ptr_0, AAssetManager *
param_AAssetManager_ptr_1, char * param_char_ptr_2, int param_int_3, AAssetManager *
param_AAssetManager_ptr_4, char * param_char_ptr_5, int param_int_6, int param_int_7)
103.    {
104.        subframe484(param_float_ptr_0);
105.        glBindTexture(GL_TEXTURE_2D, global_int_ptr_9[0]);
106.        subframe544(0x685f1008, 0x685f1010, param_AAssetManager_ptr_1,
param_char_ptr_2, param_int_3, param_AAssetManager_ptr_4, param_char_ptr_5);
107.        subframe586(global_const_int_ptr_1, param_int_6, param_int_7);
108.    }
109.
110.    void subframe466(float * param_float_ptr_0, AAssetManager *
param_AAssetManager_ptr_1, char * param_char_ptr_2, int param_int_3, AAssetManager *
param_AAssetManager_ptr_4, char * param_char_ptr_5, int param_int_6, int param_int_7)
111.    {
112.        subframe501(param_float_ptr_0, param_AAssetManager_ptr_1, param_char_ptr_2,
param_int_3, param_AAssetManager_ptr_4, param_char_ptr_5, param_int_6, param_int_7);
113.        glEnable(GL_SCISSOR_TEST);
114.    }
115.
116.    void frame3(AAssetManager* param_AAssetManager_ptr_0)
117.    {
118.        glEnable(GL_BLEND);
119.        glScaledViewport(0, 0, 800, 1205);
120.        GLchar local_char_ptr_241[] = "Flush";
121.        subframe480(local_char_ptr_241, 0, 832, 795, 265);
122.        float local_float_ptr_242[] = { 800.0, 0.0, 0.0, 0.0, 0.0, 1205.0, 0.0,
0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0 };
123.        subframe530(global_unsigned_int_38, local_float_ptr_242,
global_unsigned_int_44, 0.133333340287, 0.133333340287, 0.133333340287, 1.0);
124.        float local_float_ptr_243[] = { 260.0, 0.0, 0.0, 0.0, 0.0, 260.0, 0.0, 0.0,
0.0, 0.0, 1.0, 0.0, 5.0, 113.0, 0.0, 1.0 };
125.        subframe464(local_float_ptr_243, global_unsigned_int_146);
126.        float local_float_ptr_244[] = { 260.0, 0.0, 0.0, 0.0, 0.0, 73.0, 0.0, 0.0,
0.0, 0.0, 1.0, 0.0, 5.0, 300.0, 0.0, 1.0 };
127.        subframe465(global_unsigned_int_166, local_float_ptr_244,
global_unsigned_int_172, 0.266666680574, 0.266666680574, 0.266666680574,
0.666666686535, GL_BLEND, global_unsigned_int_18);
128.        float local_float_ptr_245[] = { 260.0, 0.0, 0.0, 0.0, 0.0, 260.0, 0.0, 0.0,
0.0, 0.0, 1.0, 0.0, 270.0, 113.0, 0.0, 1.0 };
129.        subframe464(local_float_ptr_245, global_unsigned_int_146);
130.        float local_float_ptr_246[] = { 260.0, 0.0, 0.0, 0.0, 0.0, 96.0, 0.0, 0.0,
0.0, 0.0, 1.0, 0.0, 270.0, 277.0, 0.0, 1.0 };
131.        subframe465(global_unsigned_int_166, local_float_ptr_246,
global_unsigned_int_172, 0.266666680574, 0.266666680574, 0.266666680574,
0.666666686535, GL_BLEND, global_unsigned_int_18);
132.        float local_float_ptr_247[] = { 260.0, 0.0, 0.0, 0.0, 0.0, 260.0, 0.0, 0.0,
0.0, 0.0, 1.0, 0.0, 535.0, 113.0, 0.0, 1.0 };
133.        subframe464(local_float_ptr_247, global_unsigned_int_146);
134.        float local_float_ptr_248[] = { 260.0, 0.0, 0.0, 0.0, 0.0, 50.0, 0.0, 0.0,
0.0, 0.0, 1.0, 0.0, 535.0, 323.0, 0.0, 1.0 };
135.        subframe531(global_unsigned_int_166, local_float_ptr_248,
global_unsigned_int_172, 0.266666680574, 0.266666680574, 0.266666680574,
0.666666686535, global_unsigned_int_98);
136.        float local_float_ptr_249[] = { 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0,
0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0 };
```

```
137.        subframe466(local_float_ptr_249, param_AAssetManager_ptr_0,
"int_asset_250", 716, param_AAssetManager_ptr_0, "int_asset_251", 716, 1074);
138.        glUseProgram(global_unsigned_int_52);
139.        float local_float_ptr_252[] = { 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0,
0.0, 0.0, 1.0, 0.0, 0.0, 108.0, 0.0, 1.0 };
140.        subframe467(local_float_ptr_252);
141.    }
```

2.8. Android Native Glue Code

Once the `trace.inc` file has been generated, it's included via a C preprocessor `include` directive inside a `trace.c` file.

The `trace.c` file provides some minor utility functions that don't require being generated inside `trace.inc` (asset loading, `glViewport` overrides, etc.).

A `main.c` file performs the Android boiler-plate code of reading incoming command line options in the form of Android intents (framebuffer width and height, color depth, etc.), creating the OpenGL surface and contexts, input event loop handling, and invoking the rendering for each frame.

Finally, the `Android.mk` and `Application.mk` Android makefiles files are used to instruct the Android NDK how to perform the build.

2.9. Testing

2.9.1. Tests

The tests are written using the Python Nose[63] framework, although they can also be run standalone from the command line.

When inside Nose, the tests are run in multiprocess mode which runs them in parallel and accelerates the test execution, quite lengthy otherwise. The test execution time is currently at 25 seconds, 44 seconds in single process mode on the current development laptop. In the past, before more targeted *.gltrace* files and optimizations were introduced the test execution could be in the minutes, so the support for parallel test runs is an important feature worth considering when designing new tests.

The approach followed for testing is that of *golden master testing*[64] or, in other words:

1. run the test once generating the program's output,
2. manually and visually verify the output is correct, store that output as the *gold file* (this process is frequently called *gilding*),
3. at any later time the code is modified, run the test and compare the output against the *gold file*. Fail the test if the output doesn't match the *gold file*.

Before every commit to the git repository, the tests would be run in a *continuous integration* fashion.

One variation introduced over that scheme, is that the gold files are not really stored anywhere, given that they can be generated every time new code is pulled, before

performing any modifications to the code (this guarantees that the generated files match the gold files).

This is convenient since it prevents having to commit gold files to the git repository, since gold files coming from an OpenGL ES trace can become quite big (tens of megabytes).

Note that the aforementioned *gilding* process needs to happen not only the first time the gold files are generated, but at any time where modifications in the code cause a valid gold file modification.

A way of avoiding manual gilding would be to have self-verifying tests that check certain intrinsic property in the test results.

For example, given an OpenGL ES trace, it can be self-verified by storing the rendering the first time the trace was run, and checking that the new rendering matches the old one.

This wasn't implemented because it wasn't found desirable to force an Android device to be connected in order to run the tests. Also, note that not only this method requires a device to be connected, but it has to be the same device where the original rendering was generated, since rendering results of different devices may be different (the OpenGL ES specification doesn't require pixel-accurate results between OpenGL ES implementations).

Another interesting self-verifying alternative would be to:

1. generate an APK from a trace.inc file
2. run the APK in OpenGL ES trace capture mode, capture the trace
3. invoke `glparse.py` on that captured trace, generating a new trace.inc file
4. compare the original `trace.inc` and the new `trace.inc` files or the `.gltrace` files.

This wasn't implemented because it also needs an Android device available and has the complexity that the two generated `trace.inc` files may not match due to boilerplate OpenGL ES calls made by the startup code (i.e. `glparse.py` is not idempotent).
Similarly, the `.gltrace` files may not match due to runtime differences (timestamps that are stored in the `.gltrace` file, OpenGL ES generated names, etc.).

As a partial self-verification, the code does include asserts that test for invariants and will make the test fail if the assert is not satisfied.

2.9.1.1. glparse_test.py

Initially the gold files used for testing came from capturing `.gltrace` files from existing Android applications (all Android applications use OpenGL ES as rendering backend) and generating `trace.inc` files from them.

Those `trace.inc` files are not ideal since they are big, hard to verify manually and don't exercise the whole range of OpenGL features.

Eventually `glparse.py` was able to bootstrap itself, this is,

1. generate an APK from a `trace.inc` file
2. run the APK in OpenGL ES capture mode
3. capture the trace with `glcap.py`
4. generate a `.gltrace` file from it.
5. generate a new `trace.inc` from it as gold file.

This allowed, with just writing a few targeted `.inc` files, generate the corresponding `.gltrace` files and use those as tests.

As mentioned in glcap.py, `glcap.py` was used to automate this capturing process of targeted traces. The batch script `generate_gltrace_from_inc.bat` encapsulates this process.

Gold Filename	Purpose
Resources.inc	Verify resource creation/destruction works (shaders, framebuffers, uniforms, renderbuffers, textures...)
simple.inc	Simple OpenGL ES state initialization
triangle.inc	Simple OpenGL ES rendering a triangle
twocontexts.inc	Two OpenGL ES contexts
wars.inc	Workarounds implemented in the parser

Table 3: glparse_test.py Gold Files

2.9.1.2. deinline_test.py

Simple C files were written to exercise each of the deinlining features, including a new C file every time a bug was found during the development, and a more complex C file coming from an OpenGL ES trace.

The refactored C files resulting from running `deinline.py` on them were used as gold files.

Gold Filename	Purpose
codeflow.c	Check the supported code flow, conditionals, switch statements, function calls
complex.c	Real application trace
locals.c	Check that functions are deinlined even with local declarations and empty lines in between
params.c	Check for simple actual parameters
params_bug_aliasing.c	Check for the different techniques of aliasing preservation (coalescing to a global, coalescing to pointer dereference, explicit memcpy)
params_casting.c	Verify that parameters are casted properly to prevent compiler warnings.
params_coalesced.c	Verify that parameters with the same value are coalesced
params_coalesced_different_names.c	Verify that parameters with the same value are coalesced, even if the matching pairs in one invocation don't match one-to-one another invocation
params_coalesced_partial.c	Verify that parameters with the same value are coalesced, even if the matching pairs in one invocation don't match one-to-one another invocation and there are also non-coalescing usages of the parameter between invocations
params_common.c	Verify that common parameters in the call-sites are removed from the function parameters
params_common_void.c	Verify that functions that are left with no parameters work and are left with a void parameter
params_types.c	Test of the different parameter types
simple.c	Simple test

Table 4: deinline_test.py Gold Files

2.9.1.3. build_test.py

This is an end-to-end test, including parsing, refactoring, NDK and ANT builds (but excluding install and run), done by invoking `build.py`.

Currently the test exercises a single `.gltrace` file, to avoid lengthy modify-test-commit iterations, but multiple `.gltrace` files are also supported.

The gold files are the source files generated as result of those phases (`trace.inc` files, assets directory, NDK object files, etc.).

Some object files are excluded from the gold files because the NDK compiler toolchain generates files that are either time-dependent (e.g. stores the time of day) or random-number-generator-dependent (e.g. stores some random seed).

Gold Filename	Purpose
Triangle.gltrace.gz	Generate OpenGL trace corresponding to triangle.inc

Table 5: build_test..py Gold Files

2.9.2. Code Coverage

Nose's plugin for Python's code coverage package was used to measure the code coverage of packages and tests.

Running tests\make.bat in coverage mode

```
1.      c:\Python27\python.exe -c "import sys; sys.path.append('..'); import nose;
nose.main()" --detailed-errors --stop --logging-level=INFO --with-coverage --cover-
html --cover-html-dir=_out\cover --cover-tests --cover-erase --cover-branches
1>_out\make.log
```

creates the following report:

Module	statements	missing	excluded	branches	Partial	Coverage
build.py	110	11	25	42	17	80%
build_test.py	37	3	13	4	1	85%
common.py	40	4	22	10	1	86%
deinline.py	424	2	38	182	7	99%
deinline_test.py	51	0	12	12	1	98%
glparse.py	585	100	19	274	46	80%
glparse_test.py	39	3	13	6	1	87%
gltrace_pb2.py	31	0	0	0	0	100%
utils.py	14	1	13	6	1	90%

Table 6: Per Module Code Coverage

Line-by-line coverage can be seen in the full report, to summarize:

- Missing coverage for most of the files is due to exception and error handling code that is not exercised by the tests.
- Coverage for `glparse.py` is missing some OpenGL ES functions that don't appear in the test traces, and some hard-coded configuration options that hide some code paths.

3. Results

3.1. Python Performance

At various steps during the development, the performance of the Python code was examined using the *RunSnakeRun* profiler, and the code optimized as necessary.

The following section presents one of those optimization sessions in which a Python function is changed from taking 96% of the deinlining time, to 25%, and finally 7%.

In general, it was found that the less work is done in Python (as opposed to calling Python Standard Library functions implemented in C), the better. This is true even for cases where the non-Python alternative algorithm is asymptotically worse than the Python alternative.

One of the limitations of *RunSnakeRun* worth noting, is that it doesn't show per-line-number information, so it's often needed to move code into functions if a break-down of the execution times is desired.
This is a *cProfile* limitation (Python's standard profiler), which is where *RunSnakeRun* displays the information from, but other profilers[65] do allow for line-per-line information (although at the expense of having to tag the code to be profiled, presumably because of increased overhead).

3.1.1. Finding Aliasing/Aliased Relationships Optimization Case Study

As explained in Aliased Parameter Coalescing, when deinlining it's necessary to find which instructions alias another instruction in order to maintain the aliasing relationship in the new function.

The function `gather_per_aliasing_instruction_information` is in charge of collecting that information, in what essentially is an $O(n^5)$ algorithm[5]:

```
1.      Let S be the set of blocks of instructions that have been found to be worth
refactoring into a single block,
2.      for each block of instructions B in S:
3.          for each instruction I in this block B:
4.              for each parameter P in this instruction I:
5.                  for each previous instruction PI:
6.                      for each previous parameter PP in the instruction PI:
7.                          if PP aliases P
8.                              store I, P, PI, PP in list
9.      return list
```

The check in line `7` is simply a string comparison and a regular expression that checks if `P` dereferences `PP`, i.e. `P == PP[n]` or `&P == PP`. In Python this is expressed as:

```
1.      if ((prev_param == ("&" + param)) or
2.          (re.match(prev_param + r"\[\d+\]", param) is not None)):
```

Figure 8 shows the *RunSnakeRun* profile for that implementation.

[5] Strictly speaking it's O(count(B) * count(I)^2 * count(P)^2)

Figure 8: Baseline aliasing instruction profiling

In that case `gather_per_aliasing_instruction_information` is taking 96.10% of the execution time of the whole deinlining.

From that percentage, matching the regular expression (compiling the regular expression plus doing the matching) is taking 93.84% of the total time. Note how most of the regular expression code in the Python Standard Library is itself written in Python.

The obvious optimization is to simplify the matching operation. An easy simplification is to use a string reverse find `rfind` instead:

```
1.      at_index = param.rfind("[")
2.      if ((prev_param == ("&" + param)) or
3.          ((at_index > 0) and (prev_param == param[:at_index]))):
```

Figure 9 shows the *RunSnakeRun* profile for that implementation.

Figure 9: rfind optimization of aliasing instruction profiling

With that change, `gather_per_aliasing_instruction_information` became 25.40% of the total time, with `rfind` taking 7.10%

Still, that algorithm is taking a quarter of the execution time, just for dealing with the unlikely case of finding aliasing/aliased instructions, so a better solution has to be found.

At this point the only solution likely to have enough execution time reduction is to change the algorithm so it's no longer $O(n^5)$.

Given that the original algorithm is basically two nested searches, applying Andrew Koenig's *Fundamental Theorem of Software Engineering*[66], we can introduce some indirection in the form of a hash lookup in order to accelerate the second search.

A good way of reducing the execution order is by using a hash indexed by the variable name (i.e. the parameter without any array indexing or address-of reference), and that stores the list of instructions where that variable is used as a parameter. This prevents parameter information having to be linearly searched a second time:

```
1.      for each block of instructions B in S:
2.          hash = empty
3.          for each instruction I in this block B:
4.              for each parameter P in this instruction I:
5.                  V = varname(P)
6.                  hash[V] = hash[V] + {I, P}
7.          for each V in hash:
8.              for each {I, P} in hash[V]:
9.                  for each {PI, PP} in hash[V]:
10.                     if (PI < I) and (PP aliases P):
11.                         store I, P, PI, PP in list
12.     return list
```

Which transforms the algorithm in $O(n^3) + O(n \cdot m^3)$, which not only is faster than $O(n^5)$ but also reduces the cardinality of *m* with regards to the previous *n*, since only parameters that are variables (as opposed to literals, etc.) are stored in the hash.

Figure 10 shows the *RunSnakeRun* profile for the hash implementation.

Figure 10: Hash optimization aliasing instruction profiling

With the hash optimization, the time for gathering aliasing instruction information gets finally reduced to 7.47% of the overall deinlining time.

3.2. Trace Analysis

The following section describes the use of the trace replayer for analyzing different aspects of the GPU of the following architectures:

Device	SoC	CPU	GPU	RAM	Display	Year
Fire Phone[67]	MSM8974	Quad Krait 2.2GHz	Adreno 320	2GB	1280x720	2014
Nexus 7 2012	Tegra3[68]	Quad Cortex-A9 1.2GHz	GeForce ULP MP12	1GB	1280x800	2013
Fire HD7[69]	MTK8135	2 Cortex-A15 1.5GHz 2 Cortex-A7 1.2GHz	PowerVR G6200	1GB	1280x800	2014
NVIDIA Shield Portable	Tegra4[70]	Quad Cortex-A15 1.9GHz	GeForce ULP MP72	2GB	1280x720	2013
Fire[71]	MTK8127	Quad Cortex-A7 1.3GHz	Mali 450	1GB	1024x600	2015
Samsung S7 (US) [72]	MSM8996	2 Kryo 1.6GHz 2 Kryo 2.2 GHz	Adreno 530	4GB	1440x2560	2017

Table 7: Analyzed Architectures

Figure 11: Fire (left), Fire Phone, NVIDIA Shield Portable, Nexus 7 2012, Samsung S7 US, Fire HD7 (right)

For the analysis, 370 frames from a trace corresponding to the first stage of the Android game Sonic Dash[73] were used. Three frames rendered by the tool are shown in Figure 12.

Figure 12: Frames of the trace replayer for Sonic Dash

The time between `eglSwapBuffers` calls was then graphed, and the average included in parenthesis in the graphs' legend.

3.2.1. Onscreen vs. Offscreen Rendering

Normally, OpenGL ES applications render to a backbuffer, followed by an `eglSwapBuffers` call to swap the backbuffer to the front buffer and make the rendering visible on the display. These series of steps is often called *onscreen* rendering

There are several drawbacks of using onscreen rendering to analyze and compare GPU performance:

1. Limits the rendering speed to the frame rate of the display. On Android, in order to avoid *tearing* (simultaneously showing the contents of two different

backbuffers on the display) `eglSwapBuffers` will only swap at the display's vsync boundaries (when the pixel scan hits the end of the display and the display is not being refreshed)[6]
2. Performs an extra scaling step when the resolution of the backbuffer is different from the display's resolution. This can cause significant overhead when the backbuffer or the frontbuffer have a high resolution.
3. Swapping can incur in an extra copy depending on the window compositor settings. When a buffer is swapped, Android combines it with the rest of the elements being displayed (navigation bar, button bar, etc.). This combination is done by the *SurfaceFlinger* component and can, depending on whether an overlay engine is available or not, incur in an extra copy.

One non-obvious problem of limiting the frame swap to vsync boundaries, is that applications will stall waiting for the backbuffer to be available once they have swapped. This makes applications to either refresh faster than the display or will refresh at a multiple of the refresh rate, since a new backbuffer won't be made available until the display is done displaying the previous one.

This problem is normally less noticeable because applications normally use *doublebuffering*, where the application alternates rendering to two different backbuffers, so it can render to one backbuffer while the previous one is being displayed, at the expense of increasing the time between the application issues the OpenGL ES calls and those calls make it to the display in the way of pixels.

Nevertheless, even with double buffering, a fast enough application will eventually starve for backbuffers and will have to stall until a new buffer is released, at a vsync boundary.

To overcome the onscreen rendering limitations, when measuring GPU performance *offscreen* rendering is used instead, where the backbuffer is never swapped to the frontbuffer, thus it's never made visible and the rendering speed is not limited to the display's refresh rate or unnecessarily scaled or copied.

For example, Figure 13 shows how Tegra4 renders slightly faster offscreen (16.37ms per frame) than onscreen (18.46ms per frame), due to a combination of the factors mentioned above.

[6] This is a design decision of Android's implementation of EGL, other operating systems allow decoupling the backbuffer swap frequency from the display frequency, at the expense of causing tearing (e.g. on Windows the extension WGL_swap_control[32] or WGL_swap_control_tear[33] which the author contributed to while employed by NVIDIA Corporation).

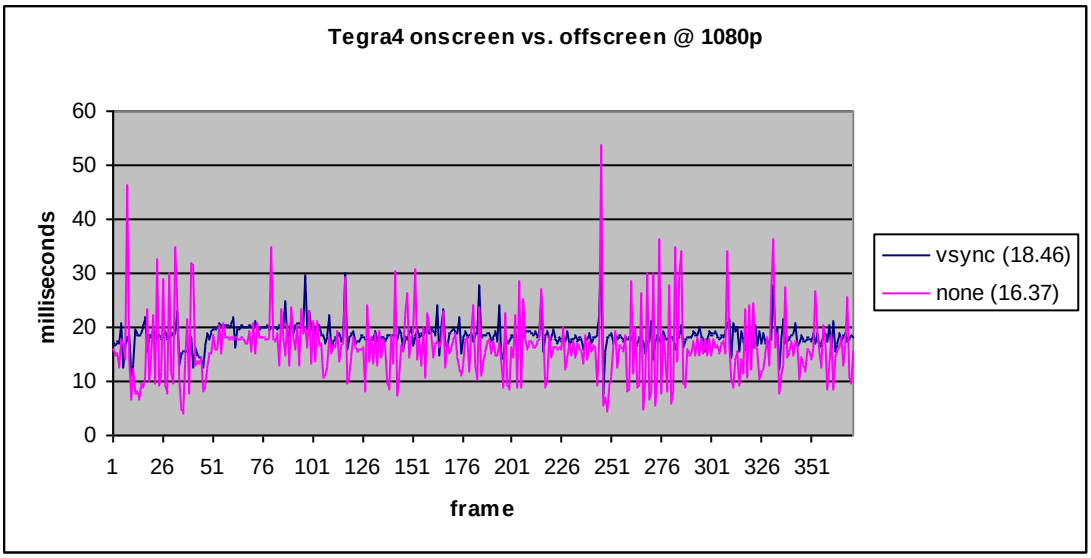

Figure 13: Tegra4 onscreen vs. offscreen @ 1080p

3.2.2. Offscreen Rendering Synchronization Methods

One of the problems of offscreen rendering is that, since the backbuffer is never displayed, it can be argued that all rendering can be optimized away and discarded.

Many GPU drivers utilize this leeway to its full extent and completely discard all rendering to a backbuffer surface that is never otherwise used.

In order to prevent the driver discarding all the accumulated rendering calls, there are several methods that can force the synchronization of the graphics pipeline:

1. Inserting an `EGLSync` object
2. Doing a `glFinish`
3. Doing a `glReadPixels` of one pixel from the of the backbuffer

The `EGLSync` object method is the most light weight, since it only inserts an `EGLSync` object (it doesn't even need to wait for it) and overhead-wise it should be comparable to onscreen rendering, but it's only available on some EGL implementations.

The `glFinish` method is more heavy weight since it has to wait for all the rendering to finish, locking the CPU thread until the GPU is done, serializing CPU and GPU.

The `glReadPixels` method is the most heavy weight since not only it has to do all the work on serialization that the `glFinish` method does, but also a roundtrip to the GPU memory reading the pixel.

Note that some drivers may ignore `glFinish` calls with the argument that it had been misused by developers in the past.

Figure 14 shows how Mali 450 behaves for the different synchronization methods:

- ignores the rendering to an offscreen surface (10.62ms),
- when an `EGLSync` is used, the rendering time increases to 27.10ms.,

- similar overhead of the `glFinish` and the `glReadPixels` methods (42.96 and 43.73ms, respectively).

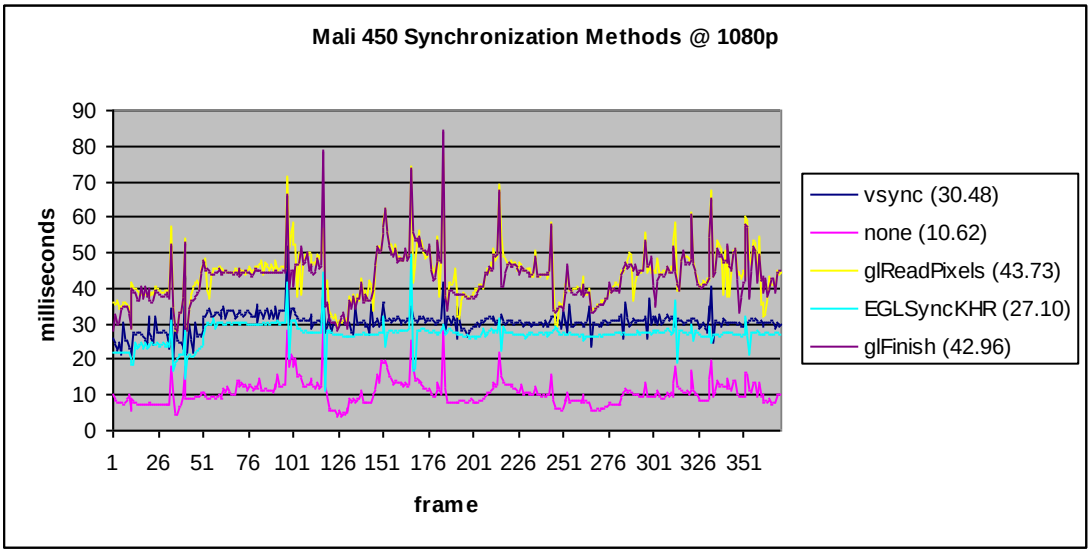

Figure 14: Mali 450 Synchronization Methods @ 1080p

Figure 15 shows how much `glReadPixels` and `glFinish` can penalize performance on Adreno 530:

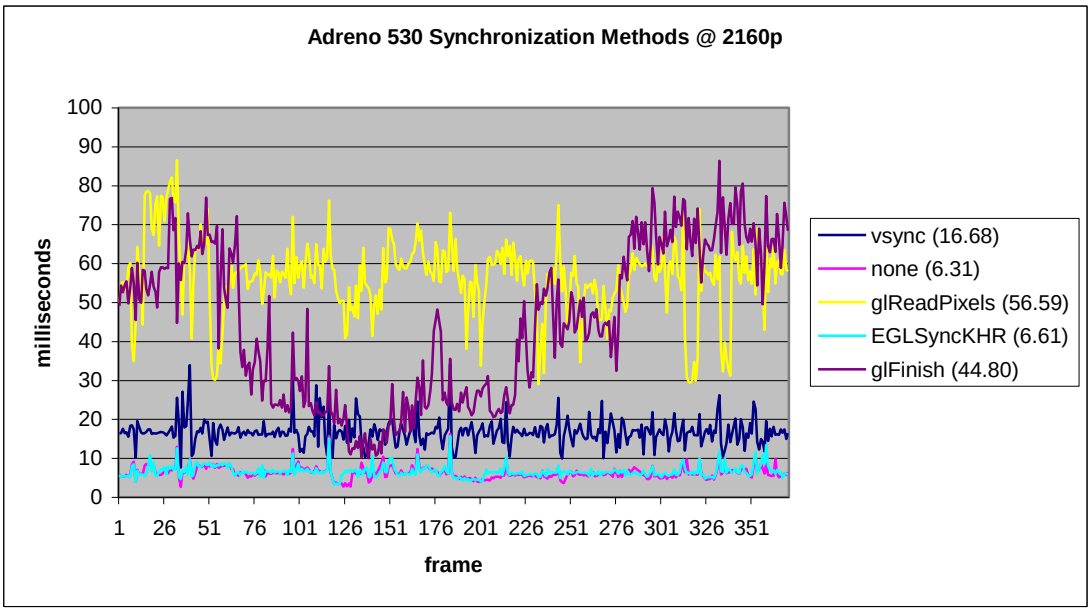

Figure 15: Adreno 530 Synchronization Methods @ 2160p

3.2.3. Performance With Different Framebuffer Sizes

One interesting usage of a trace replayer is to forecast the behavior of a given hardware in future market conditions.

One such condition is the current migration of the market from HD (1080p, 720p) to Ultra HD (4K).

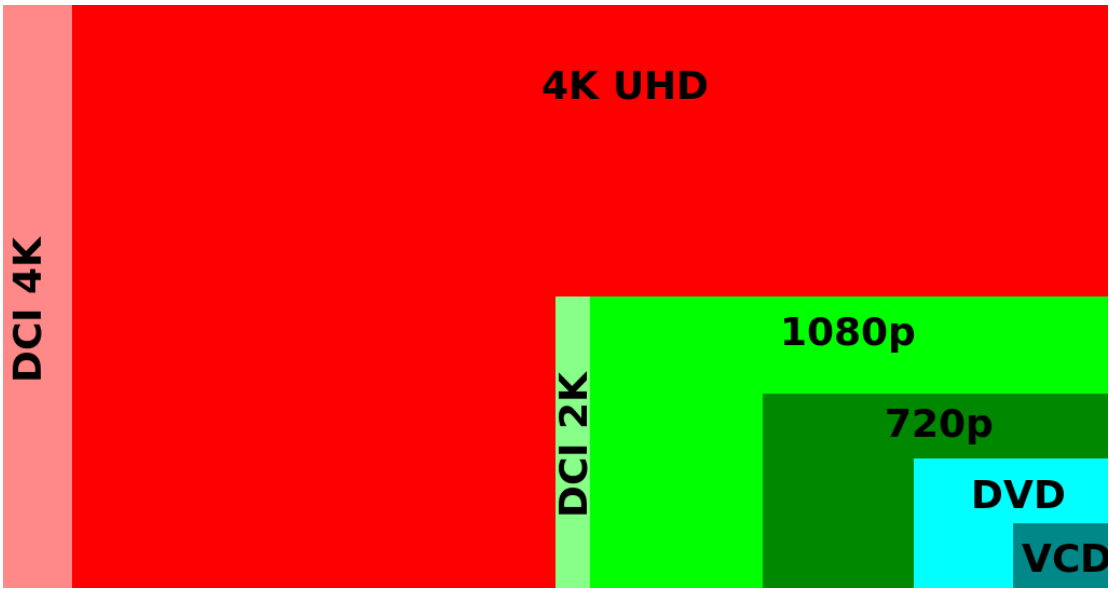

Figure 16: Comparison of common broadcast resolutions[74]

Despite the unavailability of 4K mobile displays, the GPU to select can be future proven by measuring the performance of offscreen rendering at different resolutions.

A similar scheme can also be used to evaluate the suitability of the GPU in future VR products that require higher resolution and stereoscopic rendering (which requires to replicate the rendering for each eye).

On a second axis to resolution, there's the refresh rate achievable. It's industry standard to consider adequate a refresh rate of 60 frames per second (60Hz or 16.6ms of frame time) for interactive content.
Some non-interactive content can be displayed at lower refresh rates and still be adequate (cinema is well-known to require just 24Hz, since directors have learnt and developed techniques to live with the limitations of the format, like reduced panning[75]).

Yet on a third axis not analyzed here, there's the power consumption, since performance on mobile devices without regarding to power consumption is of limited usefulness, but that requires of disassembled devices connected to dedicated power measurement equipment[76].

The following subsections will show how the different architectures perform at 720p, 1080p, and 4K resolutions.

One limitation to take into account is that, while the trace replay approach allows to easily change the resolution of the offscreen framebuffer, it doesn't change the resolution of the assets (for example, textures sizes, density of the polygonal meshes).

It's likely that future high resolution applications will also use high resolution assets. In this sense, the current trace replay that only enlarges the framebuffer will give optimistic results.

Note that the trace approach is flexible enough to modify the textures to use higher dimensions, but using higher resolution meshes is more complicated (although it

could also be approximated by tessellating the ones existing in the trace, but this also has corner cases).

3.2.3.1. 720p

Figure 17 shows how the frame times at 1280x720 for all the architectures but for Tegra3 are below the required 16.6ms.

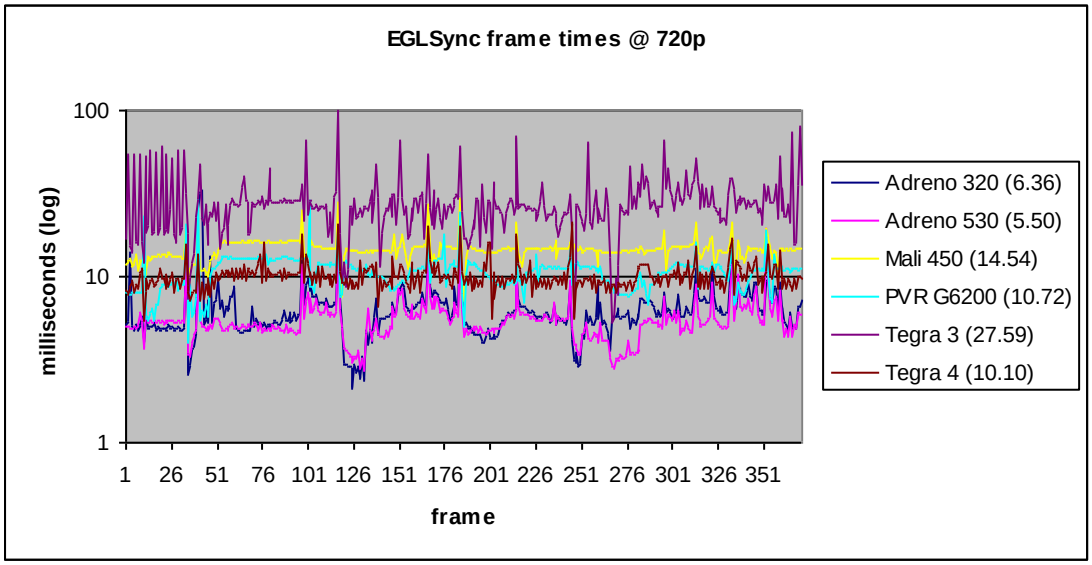

Figure 17: EGLSync frame times @ 720p

3.2.3.2. 1080p

On Figure 18, at 1920x1080 most GPUs start to be too slow for sustained 60fps, but for the Adrenos and PowerVR.

It's interesting to note how the best of the GPUs, Adreno 530, runs at the same speed at 720p than at 1080p. This means that at these resolutions the bottleneck for this GPU is not the GPU itself, but the CPU not being able to feed it fast enough.

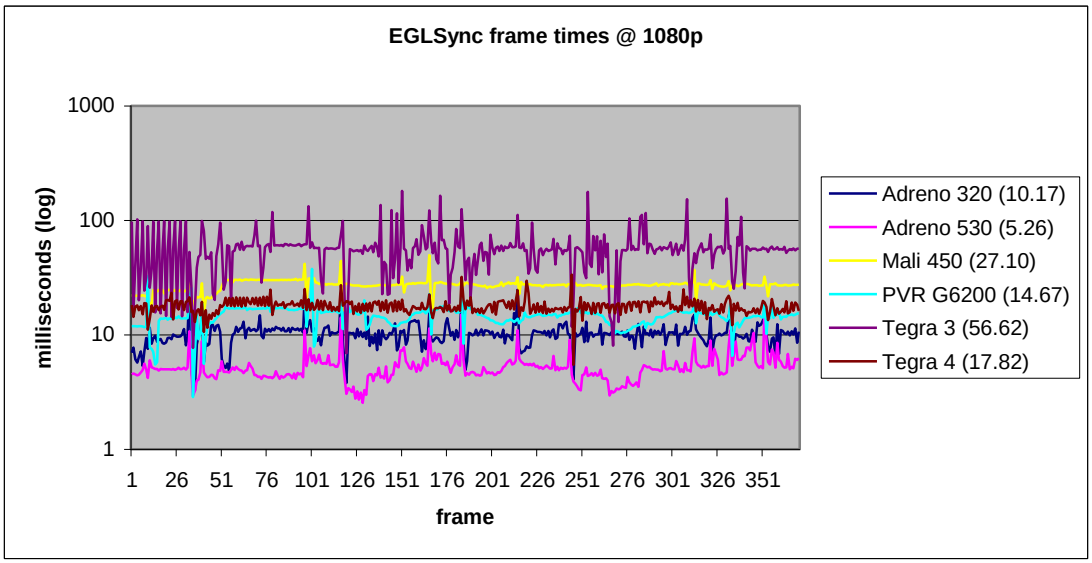

Figure 18: EGLSync frame times @ 1080p

3.2.3.3. 2160p (4K)

Figure 19 shows how at 2160p, only Adrenos and PowerVR can even run the test at all.

The missing GPU architectures couldn't run the test for different reasons:

- Tegra3 failed to create a 2160p backbuffer.
- Tegra4 was only able to create a 2160p backbuffer in onscreen mode.
- Mali 450 crashed half way through the demo when running in `EGLSync` mode (killed by Android's low memory killer).

Adreno 320 and PowerVR have fallen off the interactive graphics bandwagon and are in the 30fps territory.
Nevertheless, despite 4x increase in pixels from 1080p to 2160p, the frame times increase well below linearly with the amount of pixels (Adreno 320 ~3x and PowerVR ~2x), which gives an idea on how finely tuned the memory subsystem and locality policies of each architecture are.

Again the best of the crop, Adreno 530, is slowly starting to move the bottleneck from the CPU to the GPU, but its frame times are still well below the other GPUs and in refreshing faster than 60fps,

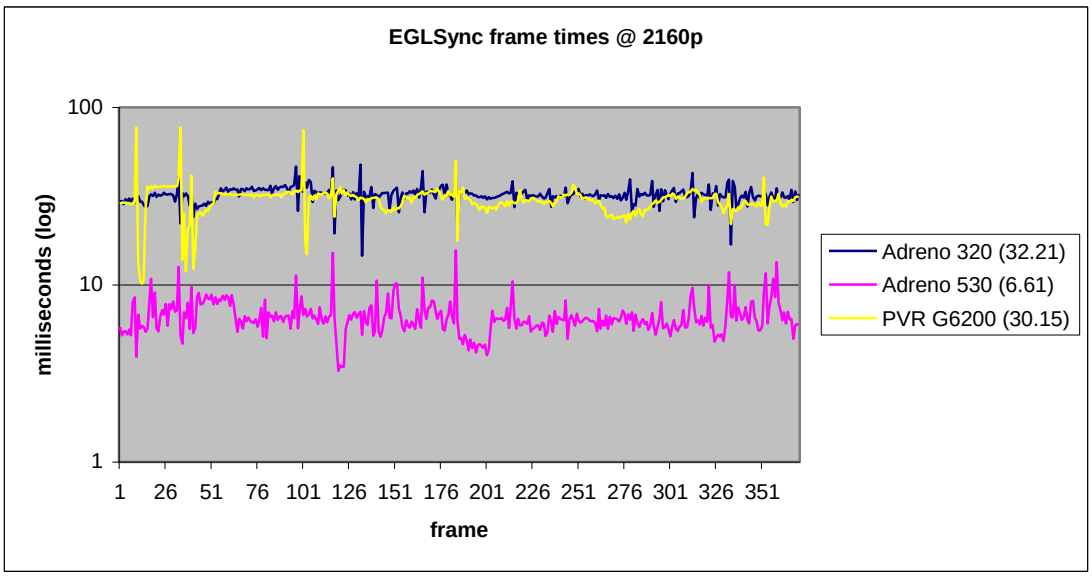

Figure 19: EGLSync frame times @ 2160p

3.3. Deinline Performance

As mentioned previously, the Python implementation of the deinline algorithm is not optimal in speed, both because of

- ease of implementation decisions (the suffix array is regenerated on every iteration, rather than updated), and
- speed limitations of an interpreted language like Python.

In order to overcome those, as detailed in Sliding Window Suffix Array, a sliding window approach is used where, instead of adding all the frames and subframes to the suffix array, only N frames and subframes are added on each iteration.

The sliding window approach increases the execution speed, at the risk of losing compression opportunities, so it's important to understand how the window size affects speed and compression.

To analyze that, the `sonicdash_stage1` trace with 886 frames was deinlined for different window sizes.

Figure 20 shows how the window size affects the compression ratio for that trace, calculated using the classic data compression definition[77] as the original size divided by the compressed size (in this case, number of C lines in the original source code, divided by the number of C lines in the refactored source code).

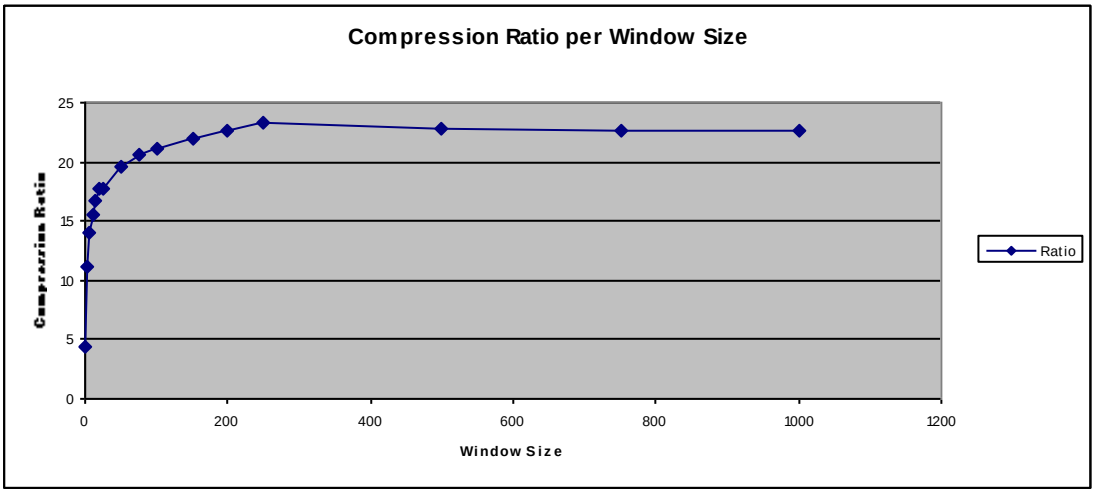

Figure 20: Compression Ratio per Window Size

It's interesting to note that, not only there are diminishing returns for window sizes greater than 50 (compression ratio hovers between 20 and 23 for window sizes between 50 and 1000), but that the compression ratio actually peaks at 23.327 with a window size of 250, and then it decreases to slightly above 22 for greater window sizes.

The decrease of compression ratio for greater window sizes is not unexpected since, as previously mentioned, the greedy search for the best compression is best-effort and doesn't guarantee the optimum results. Changes in the initial conditions like the window size result in better or worse compressions being found.

On the other hand, the compression time, as shown in Figure 21, increases dramatically for window sizes above 250, from 203 seconds (~3 minutes) to 4120 seconds (~1 hour) for a window size of 1000.

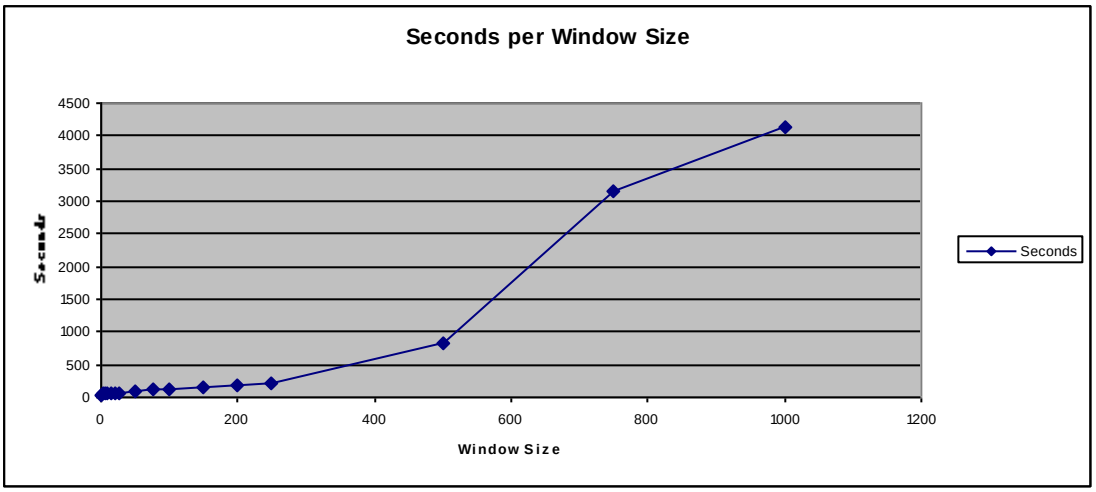

Figure 21: Seconds per Window Size

From the empirical results for this specific trace, it follows that a window size of 50 offers a good trade-off between compression ratio at 19.56 (from a minimum ratio of 4.314 and a maximum of 23.327) and compression time at 89 seconds (from a minimum time of 37 seconds and a maximum of 4120).

4. Limitations and Future Work

The following sections describe the limitations of the tool and areas for future improvement.

4.1. Vendor-specific Extensions

Oftentimes applications execute different code depending on the platform they run in.

This is usually the case with compressed texture formats[78], since until recent versions of OpenGL ES there wasn't a cross-vendor compressed texture format. This means that traces recorded in one platform cannot be replayed on other platforms, limiting the usefulness of a replayer approach.

Several avenues are possible to work around this issue:

- If you have control over the platform code, you can hide vendor-specific extensions at trace recording time in order to force the traced application to take a generic codepath. This has approach has been used successfully in the past.
- Otherwise, either at trace build time or run time, the vendor-specific format can be converted into vendor agnostic or into another vendor-specific format.

4.2. Data Modification Behind OpenGL ES' Back

Another limitation of the trace approach comes when any of the data is updated behind OpenGL ES' back. This can happen because the data has a CPU pointer (e.g. DirectTexture[79]) or because another API (e.g. OpenCL) performed the modification.
The solution to this issue is to have the routine in charge of modifying the data informing the trace recorder that the data has changed (some sort of synchronization with the graphics API is needed anyway). This can be done by adding new tracing entries or overloading existing ones.

4.3. Multithreading

Multithreading is not supported since the OpenGL ES trace doesn't store thread information.

4.4. Multiple Contexts

The multiple context support is limited to shared contexts since from the following `eglCreateContext` prototype:

```
1.      EGLContext eglCreateContext(EGLDisplay display,
2.                                  EGLConfig config,
3.                                  EGLContext share,
4.                                  EGLint const* attrib_list);
```

only the return value and the EGL version are stored in the trace. There's no information on whether this context is shared with another or what configuration was used to create the context.

The approach taken is to always share across all created contexts, and use the `EGLConfig` supplied in the command line for all the contexts.

4.5. Trace Recording Speed Affecting Trace Replay

Some applications exhibit time-dependent behavior, this is, the time it takes to submit the graphics commands may affect what the application does next (e.g. wall-clock timed physics simulations in games, etc.).

When expensive trace recording options are used (e.g. storing texture data), it's possible that the recording will affect the application's execution timing.

For the specific case of texture data, since the texture dimensions are always available, it's possible to not store the texture data in the trace, and use impostor data at trace replay time. This will speed up the trace recording, but may also produce mismatches in the replay with regards to the original execution since it's possible that the texture data (transparency, translucency, normal map values, etc.) affects the execution time of the trace.

Note most of the time enabling texture data storage in the trace is not an issue since update of texture data is already an expensive operation so applications tend to do it sparingly or at the beginning of the application.

4.6. Inlined Asset Coalescing

As explained in Non-scalar Parameters, the current code generator does asset file coalescing, but it doesn't coalesce assets that are inlined via variable initialization. It's expected that similar size savings and code simplification can be achieved via coalescing inlined assets.

In addition, this coalescing can prevent unnecessary duplication that can lead to memory thrashing not present in the original program the trace was obtained from.

4.7. Bugs in Android's Trace Layer

During the development of this tool, several bugs were found in the recording of the trace, where e.g., trace parameters were missing or incorrect. Some of them were fixed in Amazon's FireOS, others were worked around in the trace itself.

4.8. CPU load

As described above, the trace replaying approach obtains raw GPU performance and CPU driver overhead, but it doesn't model at all other application CPU overhead. One possible way of modeling that workload would be to use the timestamp included in

the trace entries to model that CPU workload and normalize that workload across CPUs.

4.9. Statistics Collection

While the trace collects basic statistics at execution time like frame duration, one interesting avenue to explore is collecting other statistics.

These statistics could be of:

- static nature, by analyzing the trace offline, like polygons submitted per frame, textures used per frame and size of each, or
- dynamic nature by replaying the trace in special modes that measure overdraw, texture mipmap level used, etc.

4.10. Other OpenGL ES Versions

The implemented solution only supports OpenGL ES 2.0 and a few extensions. It should be easy to extend to OpenGL ES 3.0 or later.

4.11. Automatic Code Refactoring Improvements

The current compression factor calculation is simplistic since it only tries to minimize the number of C lines and parameter pushing into the stack is not accounted for.

Two different C lines can translate to a very different number of machine code instructions. If minimizing the number of machine code instructions is desired, more intelligence could be added to the compression factor calculation. One way of doing this would be to compile the given code and use C line number to machine code symbolic information to extract the number of machine code instructions of a given line of C code.
Simpler heuristics that don't require producing machine code are also possible (count number of C operations, C operands, C function calls, etc.).

The parser used by the code refactorer is also very limited, specifically it doesn't support field dereferences (via arrow or dot operator). A better parser would make the refactorer more robust and applicable to more C constructs (conditionals, loops, etc.).

4.12. Automatic Code Refactoring Speed

As mentioned earlier, unconstrained automatic code refactoring can take several minutes to an hour. This would be fixed by implementing the refactoring in C or C++ instead of Python.

Alternatively, the suffix array information could be used when replacing code (right now a linear scan is used), and updated instead of thrown away and recalculated from scratch on the next iteration.

Finally, the windowed approach could be combined with a multithreaded version that compresses different window ranges in different threads.

4.13. Other Automatic Code Refactoring Uses

The developed refactoring algorithm can be enhanced to work with not only generic C code, but also other languages where instruction count is important (JIT, trace caches[80][81][82], GPU shaders, etc.).

5. Project History, Challenges and Curiosities

This section details the origin of the project, personal view of it and challenges ad curiosities faced during the development.

5.1. Trace Replayer Origin

OpenGL call recording and replaying is not a new concept, all graphics hardware companies and many software ones have that kind of functionality in one way or another. In 2014 Amazon wasn't one of those.

During my employment at Lab126 (the Amazon subsidiary that builds the Kindle Fire family of devices), I worked in the group responsible for evaluating SoCs for future products, specifically the GPUs component.

A coworker was tasked with writing a benchmark that mimicked the FireTV home UI, which runs on FireOS, with the purpose of evaluating the performance of different SoC alternatives. He couldn't use the real FireTV home application because the software layers that the application depended on, were not ready on the evaluation platforms, which are only provided with AOSP (not the Amazon-specific service layers that FireOS adds, ported at a later time and only on the final platform).

Coincidentally, I had been looking into *Android Tracer for OpenGL ES* protobuf format with the idea of writing some kind of statistics analyzer, and maybe a replayer. I suggested him to record a trace of the home application and write a replayer for that format instead of a benchmark, since a benchmark would need to be written from scratch, carefully matching the existing UI code, and then maintained as the home UI changed.
One of nice to have feature of the benchmark, was being able to provide the source code to hardware vendors for suggestions on performance improvements, so he suggested going the code generation route rather than a replay of a binary trace.

He went on his way for several weeks, busily writing tens of Java classes, abstractions, and interfaces.

At some point, I suggested that it should be easy to write an automatic refactorer for the code generator. Alas, he didn't seem to think that way, so he dismissively rejected it could be done and challenged me to write it.

Putting my money where my mouth was, and without ever having seen his code, I wrote the trace code generator in one weekend (using a lonely Python script), and in the next weekend I wrote the fully functioning refactorer's first version.
At that time, my script wasn't industrial strength (didn't have test cases or good OpenGL ES coverage), but allowed me to quickly prototype new ideas and add support to features his Java-based infrastructure would take more time to support, so I ended up using it for a few experiments while his version wasn't ready yet.

Eventually his tool was mature enough, and I ended up joining forces and adding DirectTexture[79] functionality to it. I'm proud to say that it is still being actively used by the team currently doing GPU evaluation.

5.2. Deinliner Origin

In 2001, before the GPU word had been coined, I was working on the Windows 95 and Windows NT OpenGL drivers for the first consumer-grade fully programmable graphics chip (3Dlabs' P10, later sold as workstation board and known as Wildcat VP[83], and the chip for which the *OpenGL Shading Language* or GLSL[84] was designed).

It had to support this recent API coming from Redmond, called DirectX 9, and other features that Microsoft wanted for a new operating system codenamed Longhorn[85] (which I would work on 2003 when I later joined Microsoft's *Windows Graphics and Gaming Technology Group*).

The chip was the most capable chip for its time, it had fully programmable vertex, texture and fragment engines, including loops and function calls, something that would still take close to five years to be available from the competition.

Unfortunately the chip architect had to made trade-offs, and one of them was limiting the maximum instruction length of pixel shaders it would accept, to the point that naively compiled DirectX 9 pixel shaders would not fit. It was going to require some creative thinking to support all possible DirectX 9 shaders that could be constructed.

My idea at the time was to compile the pixel shaders without caring for the instruction count limitation, and then, using the function call capability of the chip, automatically find subroutines inside the program in order to compress the shaders and fit them in the required number of instruction slots.

At that time I lacked the necessary knowledge to implement that idea, especially problematic finding the best compression factor in the presence of overlapping occurrences, so I had to park it.

10 years later, at Lab126, I was busy implementing another invention I had, which would allow for multithreaded PNG[86] image decompression, providing performance increase and power savings to Amazon's mobile platforms.
The PNG image format uses a dictionary compression similar LZW, but employing the patent-free DEFLATE algorithm[87]. Reading the data compression literature, got me re-acquainted with the suffix array data structure, which elegantly dealt with overlapped occurrences, and brought back to my memory the old plan of applying dictionary compression to source code.

5.3. Deinliner Aliasing Preservation

The biggest challenge during the development of the tool was the aliasing preservation problem described in Aliased Parameter Coalescing.

I first encountered the problem when the deinliner refactored the sequence of functions `openAsset` followed by `getBuffer`, since `openAsset` was modifying a pointer that is later passed on to `getBuffer`.
When replaying, corruption would appear on the screen or the application would crash the device because of not preserving that aliasing.

After meticulous debugging, I realized what the problem was, and worked around it by merging those functions into a single `openAndGetAssetBuffer`. I thought that would be sufficient, since I didn't realize that OpenGL ES call sequences also suffered from this problem.

Intermittently, I would notice texture corruption or crashes. This was only happening to refactored traces and not to the original one, so one possibility was that the aliasing problem was back. It was then when I realized that some sequences of OpenGL ES functions also required aliasing preservation. I fixed the problem again by implementing coalescing of aliased variable access into aliasing variable pointer dereference. I also put an assert to detect that there were no mixed aliasing/non-aliased sequences, and the assert didn't fire in my test cases, so life was good again.

It wasn't until close to the deadline for finishing, when doing the performance analysis of the refactorer using multiple window sizes, that the assert I put fired. At this point I determined to fix it once and for all, arriving to the final implementation that preserves aliasing when it has to and it doesn't if it's not needed.

I'm especially proud of having implemented something I envisioned a decade ago and making it work as I imagined. In addition, I'm also greatly satisfied with the solution I arrived in order to implement the different aliasing-preserving techniques, in an elegant way that has a robust fall-back, but that most of the time gets transparently fixed by coalescing pointers into dereferences.

5.4. Python Peculiarities

Python is a good language for prototyping, especially when dealing with strings, dictionaries and lists. It's productive and expressive, has an exhaustive standard library and support packages, and the *JetBrains PyCharm* IDE makes a joy the *read eval print loop* or REPL iteration cycle.

I probably wouldn't use anything else for non-performance critical tasks, but it has a few hidden traps that it's important to be aware of.

The most important is the fact that internally uses references, so when dealing with nested lists, many times explicit element-by-element copies are needed if you don't want to end up modifying the original list.

The following Python console log illustrates the point

```
1.      >>> L = [[0]] * 5
2.      >>> L
3.      [[0], [0], [0], [0], [0]]
4.      >>> L[1]
5.      [0]
6.      >>> L[1].append(1)
7.      >>> L[1]
8.      [0, 1]
9.      >>> L
10.     [[0, 1], [0, 1], [0, 1], [0, 1], [0, 1]]]
```

A list `L` is created as 5 sublists containing a single element 0, then a 1 is added to the first sublist. Counter-intuitively, all the sublists are actually modified because they are actually references to the same sublist.

It also has substandard design quirks like the only way of passing a list by reference to a function (so the function can recreate it if necessary) is by embedding it inside another list, or the fact that forgetting to qualify a variable as `global` in a local scope will silently create a local variable, or not being able to modify variables local to the parent function in nested functions (again, they silently create a variable local to the child function instead).

6. Glossary

1080p: 1080 progressive vertical pixel resolution
4K: Four thousand-pixel resolution
720p: 720 progressive vertical pixel resolution.
ADB: Android Debug Bridge
Adreno: Qualcomm's brand of mobile GPUs, initially acquired from ATI mobile division.
ALU: Arithmetic-Logic Unit
Android: Google's mobile operating system.
AOSP: Android Open Source Project
API: Application Programming Interface
APK: Android Package Kit.
ARB: Architecture Review Board
ARM: Advanced RISC Machines.
BSP: Board Support Package
FireOS: Amazon's fork of Android.
GLSL: OpenGL Shading Language
GPU: Graphics Processing Unit
HD: High Definition
Imagination: Hardware intellectual property vendor for the PowerVR family of GPUs.
Khronos Group: Realtime and multimedia not-for-profit consortium for open standards.
Mali: ARM's brand of mobile GPUs
NDK: Native Development Kit
NVIDIA: Hardware vendor, designer of the Tegra SoC and GPUs.
OpenCL: Open Compute Language
OpenGL ES: Open Graphics Library for Embedded Systems
OpenGL: Open Graphics Library
PowerVR: Family of GPUs designed by Imagination.
Qualcomm: Hardware vendor designer of the Adreno GPUs and SoCs.
SDK: Software Development Kit
SoC: System on a Chip
Tegra: NVIDIA's brand of mobile GPUs.
UHD: Ultra High Definition
XML: eXtensible Markup Language

7. Bibliography

[1] Burke, Dave (2017). Android:Celebrating a big milestone together with you. https://blog.google/products/android/2bn-milestone/
[2] Ben Elgin. Business Week (2005). Google Buys Android for its Mobile Arsenal. http://tech-insider.org/mobile/research/2005/0817.html
[3] Google (2017). Android Open Source Project. https://source.android.com
[4] Google (2017). Google Mobile Services. https://www.android.com/gms/
[5] Amazon (2017). Publish to FireOS 5. https://developer.amazon.com/android-fireos
[6] Amazon (2017). Amazon Devices. https://developer.amazon.com/devices
[7] Google (2017). Android Application Development. https://developer.android.com/index.html
[8] Google (2017). Android NDK. https://developer.android.com/ndk/index.html
[9] Google (2017). Android.mk. https://developer.android.com/ndk/guides/android_mk.html
[10] Apache (2017). Apache Ant Java Library. http://ant.apache.org/
[11] Khronos Group (2017). OpenGL Overview. https://www.opengl.org/about/#1
[12] OpenGL wiki. (2017). History of OpenGL. https://www.khronos.org/opengl/wiki/History_of_OpenGL
[13] Khronos Group (2006). OpenGL ARB to Pass Control of OpenGL Specification to Khronos Group. https://www.khronos.org/news/press/opengl_arb_to_pass_control_of_opengl_specification_to_khronos_group
[14] Akeley, K., Segal, M. (1994). OpenGL 1.0 Specification. https://www.khronos.org/registry/OpenGL/specs/gl/glspec10.pdf
[15] Khronos Group (2017). What's New in OpenGL 4.5. https://www.opengl.org/documentation/current_version/
[16] Khronos Group (2017). Khronos OpenGL Website. https://www.khronos.org/opengl/
[17] Khronos Group (2017). OpenGL Registry. https://www.khronos.org/registry/OpenGL/index_gl.php
[18] Mesa 3D (2017). The Mesa 3D Graphics Library. https://www.mesa3d.org/intro.html
[19] Microsoft (2017). OpenGL. https://msdn.microsoft.com/en-us/library/windows/desktop/dd374278(v=vs.85).aspx
[20] MIPS Developer Team. April 11th 2012. Learning about the Android graphics subsystem. https://www.imgtec.com/blog/learning-about-the-android-graphics-subsystem/
[21] Woo, Mason. Neider, Jackie. Davis, Tom. 1997. OpenGL Programming Guide 2nd Edition, Appendix H.
[22] https://www.khronos.org/opengl/adopters/
[23] Kilgard, Mark. Jan 18 2012. CS 354 Introduction. https://www.slideshare.net/Mark_Kilgard/cs-354-introduction
[24] Khronos. 2017. OpenGL ES. https://www.khronos.org/opengles/
[25] Khronos Group. 2017. EGL https://www.khronos.org/egl
[26] Kilgard, Mark J. 2003. NV_register_combiners Specification. http://developer.download.nvidia.com/opengl/specs/GL_NV_register_combiners.txt

[27] Kilgard, Mark J. 200. EXT_vertex_weighting Specification. https://www.khronos.org/registry/OpenGL/extensions/EXT/EXT_vertex_weighting.txt
[28] Eyles, John and Austin, John and Fuchs, Henry and Greer, Trey and Poulton, John. 1988. Pixel-Planes 4: A Summary. Advances in Computer Graphics Hardware II (Eurographics'87 Workshop)
[29] Giesen, Fabian. 2011. A trip through the Graphics Pipeline 2011. https://fgiesen.wordpress.com/2011/07/09/a-trip-through-the-graphics-pipeline-2011-index/
[30] Kristof Beets. June 6th 2013. Understanding PowerVR Series5XT: PowerVR, TBDR and architecture efficiency. https://www.imgtec.com/blog/understanding-powervr-series5xt-powervr-tbdr-and-architecture-efficiency-part-4/
[31] ARM Ltd (2017). Mobile. https://www.arm.com/markets/mobile
[32] Kishonti Informatics (2017). GFXBench 4.0. https://gfxbench.com/benchmark.jspg
[33] FutureMark (2017). 3DMark The Gamer's benchmark for Android. https://www.futuremark.com/benchmarks/3dmark/android
[34] Klug, B., Lal Shimpi, A. (2013). They're (Almost) All Dirty: The State of Cheating in Android Benchmarks. http://www.anandtech.com/show/7384/state-of-cheating-in-android-benchmarks
[35] Klug, B., Lal Shimpi, A. (2013). Looking at CPU/GPU Benchmark Optimizations in Galaxy S 4. http://www.anandtech.com/show/7187/looking-at-cpugpu-benchmark-optimizations-galaxy-s-4
[36] apitrace (2017). http://apitrace.github.io/
[37] http://www.gremedy.com/products.php
[38] Karlsson, Baldur (2016).RenderDoc - a graphics debugger. https://github.com/baldurk/renderdoc
[39] Google (2017). GAPID: Graphics API Debugger. https://github.com/google/gapid
[40] Google (2017). Tracer for OpenGL ES. https://developer.android.com/studio/profile/gltracer.html
[41] Beck, K. (2003). Test Driven Development: By Example.
[42] Torvalds, L. (2005). http://lkml.iu.edu/hypermail/linux/kernel/0504.0/2022.html
[43] Python Software Foundation (2017). General Python FAQ. https://docs.python.org/3/faq/general.html
[44] Python Software Foundation (2017). PyPI - the Python Package Index. https://pypi.python.org/pypi
[45] Oliver Tonnhofer (2015). Scriptine. https://pypi.python.org/pypi/scriptine
[46] Fletcher M.C. (2013). RunSnakeRun. https://pypi.python.org/pypi/RunSnakeRun
[47] Pellerin J. (2015). Nose. https://pypi.python.org/pypi/nose/1.3.7
[48] Batchelder, N. (2016). Code coverage measurement for Python. https://pypi.python.org/pypi/coverage
[49] Design of the GLES tracing library. Googlesource. https://android.googlesource.com/platform/frameworks/native/+/master/opengl/libs/GLES_trace/DESIGN.txt
[50] https://android.googlesource.com/platform/frameworks/native/+/jb-dev/opengl/libs/GLES_trace/gltrace.proto
[51] XML API Registry of Reserved Enumerants and Functions. Khronos OpenGL ES Registry, 2016. https://www.khronos.org/registry/gles/#specfiles
[52] Cytron, Ron; Ferrante, Jeanne; Rosen, Barry K.; Wegman, Mark N. & Zadeck, F. Kenneth (1991). "Efficiently computing static single assignment form and the

control dependence graph" (PDF). ACM Transactions on Programming Languages and Systems 13 (4): 451–4
[53] Android Open Source Project (2017). Implementing graphics. https://source.android.com/devices/graphics/implement.html#vsync
[54] Siva Velusamy (2012). Android Open Source Project. https://android.googlesource.com/platform/sdk/+/52ecd2f08ee936df425e4c53f9bd8310061f3f57
[55] Android Studio (2017). Configure Apps with Over 64K Methods. https://developer.android.com/studio/build/multidex.html
[56] Ziv, J. and Lempel, A. (1977) A universal algorithm for sequential data compression. IEEE Trans. Inform. Theory, IT-23, 337–343.
[57] Michael Mitzenmacher (2004). On the Hardness of Finding Optimal Multiple Preset Dictionaries. Harvard University. IEEE Transactions on Information Theory 2004.
[58] Storer, J. A. (1988) Data Compression: Methods and Theory. Computer Science Press, Rockville, MD
[59] McCreight, E. M. (1976) A space-economical suffix tree construction algorithm. J. ACM, 23, 262–272.
[60] Storer, J. A. and Helfgott, H. (1997) Lossless Image Compression by Block Matching. The Computer Journal, Vol. 40, No. 2/3, 1997 pp 137
[61] Manber, Udi; Myers, Gene (1990). Suffix arrays: a new method for on-line string searches. First Annual ACM-SIAM Symposium on Discrete Algorithms. pp. 319–327.
[62] http://java-performance.info/performance-general-compression/
[63] Pellerin J. (2015). Nose. https://pypi.python.org/pypi/nose/1.3.7
[64] Code Climate. Gold Master Testing (2014). http://blog.codeclimate.com/blog/2014/02/20/gold-master-testing/
[65] https://github.com/rkern/line_profiler
[66] https://en.wikipedia.org/wiki/Fundamental_theorem_of_software_engineering
[67] https://developer.amazon.com/public/solutions/devices/fire-phone/docs/fire-phone-specifications
[68] http://www.nvidia.com/object/tegra-3-processor.html
[69] https://developer.amazon.com/public/solutions/devices/kindle-fire/specifications/01-device-and-feature-specifications
[70] http://www.nvidia.com/object/tegra-4-processor.html
[71] https://developer.amazon.com/public/solutions/devices/kindle-fire/specifications/01-device-and-feature-specifications
[72] http://www.phonearena.com/phones/Samsung-Galaxy-S7_id9817
[73] https://play.google.com/store/apps/details?id=com.sega.sonicdash&hl=en
[74] TRauMa (2012). Comparison of Common Broadcast Resolutions. https://en.wikipedia.org/wiki/4K_resolution#/media/File:Digital_video_resolutions_(VCD_to_4K).svg
[75] RED.COM, Inc. Panning Best Practices. http://www.red.com/learn/red-101/camera-panning-speed
[76] Monsoon Solutions Inc. Power Monitor. https://www.msoon.com/LabEquipment/PowerMonitor/
[77] Wikipedia (2017). Data Compression Ratio. https://en.wikipedia.org/wiki/Data_compression_ratio
[78] Android Developers Blog (2015). Efficient Game Textures with Hardware Compression. http://android-developers.blogspot.com/2015/01/efficient-game-textures-with-hardware.html

[79] Willcox, James (2011). Using direct textures on Android. http://snorp.net/2011/12/16/android-direct-texture.html
[80] S. Liao (1996). Code Generation and Optimization for Embedded Digital Signal Processors. Ph D. Dissertation, Massachusetts Institute of Technology, June 1996
[81] Lefurgy C, Piccininni E., Mudge T. (1999). Evaluation of a high Performance Code Compression Method. Proceedings of Micro-32, November 16-18
[82] Fraser C. W. (2002). An Instruction for Direct Interpretation of LZ77-compressed Programs. Microsoft Research. Technical Report MSR-TR-2002-90
[83] Baumann, Dave (2002). 3dlabs Wildcat VP Preview. https://www.beyond3d.com/content/reviews/30/
[84] Khronos Group (2015). OpenGL Shading Language. https://www.khronos.org/opengl/wiki/OpenGL_Shading_Language
[85] Experience Longhorn (2017). Experience Longhorn. http://longhorn.ms/
[86] W3C (2003). Portable Network Graphics Specification. https://www.w3.org/TR/PNG/
[87] Deutsch, P. (1996). DEFLATE Compressed Data Format Specification version 1.3, Internet RFC 1951 (May 1996). http://www.ietf.org/rfc/rfc1951.txt