Software Improvements to Assist Development of Tegra System Software

Silicon Valley Project Center

March 12, 2020

Submitted by:
Haozhe (Percy) Jiang, hjiang2@wpi.edu
Matthew Kaminski, mkaminski@wpi.edu
Matthew McMillan, mmcmillan@wpi.edu

Submitted to:
Project Advisor: Mark Claypool, claypool@cs.wpi.edu
NVIDIA Advisor: Allen Martin, AMartin@nvidia.com

Worcester Polytechnic Institute

Abstract

NVIDIA produces a system on a chip (SoC) series called Tegra for mobile devices. The NVIDIA Tegra Systems Software Team is responsible for developing and maintaining software on the Tegra chips including code tracing, code scanning, and latency measurement. Our project assists the developers on the NVIDIA Tegra Software System Team by making tools that make their work more efficient. This includes developing a parser tool to parse the output of Tegra instruction tracing, improving and simplifying the code of a commonly used code analysis script, measuring the latencies in CPU optimization code, and evaluating the performance of atomic functions. We completed and submitted code to NVIDIA repositories for each project.

Acknowledgments

We would like to thank our sponsor, NVIDIA, for the opportunity to contribute to many of their internal technologies. We would like to thank our entire team, the NVIDIA Tegra Software System Team. Our mentors, Allen Martin, Richard Wiley, Vignesh Radhakrishnan, Adeel Raza, and Mark Young have been incredibly helpful in guiding us throughout our project. We are also helpful for all of the guidance provided to us by our MQP advisor, Mark Claypool.

Table of Contents

1. Introduction

NVIDIA has developed a series of System on a Chip (SoC) called Tegra, which target low-power mobile devices that might not need a more powerful GPU. These systems have a wide variety of applications, which include gaming devices such as the Nintendo Switch as well as autonomous vehicles. The Tegra chips use an ARM architecture which is ideal for smaller devices as it is more power efficient than x86 due to ARM's reduced instruction set.

The Tegra Systems Software Team is responsible for developing much of the software that powers the system as a whole, such as the kernel and various operating system components. Both performance and robustness are critical in this software because all other software will run on top of it. Our group worked with the Systems Software Team to improve development and workflow. We created solutions designed to give the developers more insight about their code and hardware, and to streamline the debugging process. We have completed four separate projects related to Tegra internal tooling. Those projects are 1) The STP Parser Project, 2) The Coverity Scan Project, 3) The Latency Measurement Project, and 4) The Atomics Performance Project. Both the STP Parser Project and the Coverity Scan Project were completed and submitted as a patch to NVIDIA's relevant repositories after going through NVIDIA's code review process. The code for the Latency Measurement Project was implemented, but did not yet go through the necessary code review processes in order to be merged into the *master* branch of NVIDIA's repository. The tool is available to any developer who wants to use it. The data for the Atomics Project was collected and passed on to Richard Wiley, who was in charge of the project.

However, we did not have the time to collect more extensive data that could have given us more insight.

This report describes the background information and an in-depth explanation of the methodology, implementation, and results of each of our four projects. Chapter 2 presents the background information relevant to our project. Chapter 3 discusses our first project, the STP Parser. Chapter 4 details our work on the second project, the Coverity Scan project. Chapter 5 writes about the Latency Measurement Project, our third project. Chapter 6 details our last and final project, the Atomics Performance Project. Chapter 7 presents the conclusion of our project and Chapter 8 lists our references.

2. Background

Tegra is a system on a chip (SoC) that incorporates a CPU and GPU onto one unit and Tegra is being used for gaming, machine learning simulations, and self-driving cars.

The Tegra chip runs an operating system called Linux for Tegra (L4T). Currently, the latest version is based on Ubuntu 18.04. However, the bootloader is custom-built for the Tegra system and differs from the standard Ubuntu bootloader GRUB. The startup process of the OS uses systemd, which is an initialization system and a service manager. It utilizes multithreading and serves as the glue between the user space and the kernel. The Tegra boot process depends on the systemd init process to load and configure the device, including starting system drivers.

The MIPI System Trace Protocol (STP) is a generic base protocol used by the CoreSight STM. It is specifically designed to merge multiple different standards of system traces into a uniform standard. It allows the merging of trace streams originating from anywhere in the Tegra chip.

3. The STP Parser Project

The following section describes our first project, a parsing tool for Tegra. Whenever code for a new Tegra chip version is modified, mistakes may be found after the new chip is manufactured and given to the team. NVIDIA is exploring better utilizing a hardware debugging module on the Tegra chip that may allow problems to be identified earlier in the development cycle as it can be used in the simulator. Unfortunately, the current parser tool used for hardware-based debugging provides output in a form that is difficult to read and which cannot be easily used in other programs like Excel. Our project augments the existing parsing tool with another tool that provides output in a format that can be quickly read and which can be passed into other programs.

3.1 Background

The following section describes the tracing tool for Tegra whose output needs to be parsed, the drawbacks of NVIDIA's current parsing tool, and the motivation of developing our project.

3.1.1 CoreSight System Trace Macrocell (STM)

The Jetson AGX Xavier Tegra chip has multiple processors that share the same bus. However, these processors all run at different clock speeds. As a result, it is critical to be able to understand where in execution each processor is at a given time. This allows the developers to ensure not only that the processor is functioning correctly, but also to measure its performance and find bottlenecks that must be addressed.

Since Tegra chips have multiple processors sharing the same bus, the tracing tools must run efficiently to avoid slowing down the Tegra chip. The typical tracing tool to use in this case would be the *ftrace* tool, which is used by the Linux kernel to trace code execution across different parts of the kernel. However, the *ftrace* tool has an unacceptable amount of overhead. Even a millisecond can be too long and interfere in the performance measurements.

NVIDIA currently uses the CoreSight STM to trace the behaviors of the various parts of the Tegra chip. The Arm CoreSight System Trace Macrocell (STM) is a trace source that enables real-time instrumentation of software with no impact on system behavior or performance. The CoreSight System Trace Macrocell (STM) [2] has significantly less overhead than standard tracing solutions such as the *ftrace* tool. CoreSight is the set of tools that are used to debug and trace software that runs on Arm-based SoCs. Debugging features are used to observe or modify the state of parts of the design, while trace features allow for continuous collection of system information for later offline analysis. The System Trace Macrocell (STM) [3] is a component of CoreSight Debug and Trace Tools.

3.1.2 MIPI System Trace Protocol (STP)

The CoreSight STM implements the MIPI System Trace Protocol (STP), a generic base protocol specifically designed to merge multiple different standards of system traces into a uniform standard. It allows the merging of trace streams originating from anywhere in the Tegra chip to a trace stream of 4-bit frames [4]. It uses a channel/master topology, which means that each trace source is assigned a pair of channel and master.

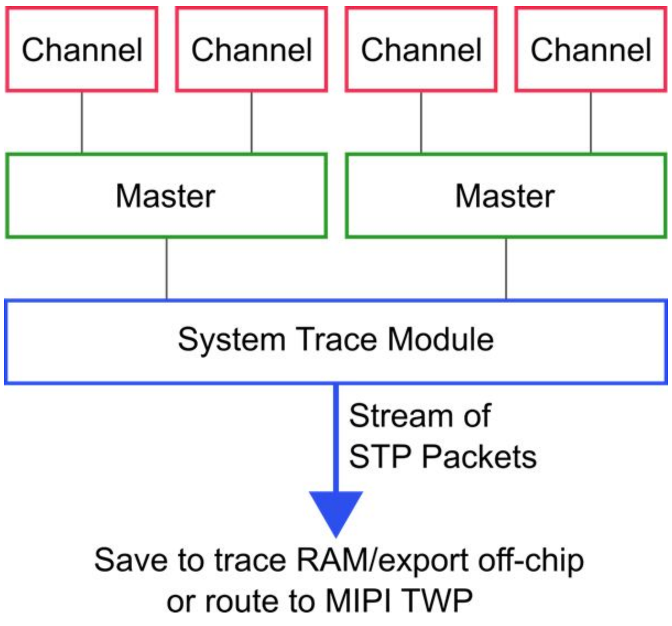

Figure 1: Topology of the System Trace Module

As shown in Figure 1, there could be multiple channels in a master and multiple masters in the System Trace Module. In our specific case, there are 128 masters, each supporting 65,536 channels on Arm CoreSight STM [4]. Therefore, a pair of master and channel represents the trace source of an output message. For each trace output message, the developers on the Tegra Software System Team would like to see the source of the message, the timestamp of the message, and the content of the message.

3.1.3 The *mem_parser* Tool

NVIDIA had been using the *mem_parser* tool to convert the raw CoreSight STM trace files into an intermediate format in the MIPI STP v2.2 standard [4]. However, the output emitted by this tool was considered primitive and difficult to analyze.

```
C8:
        CHANNEL ID = 3
FLAG_TS:    TIMESTAMP: 584791825408
D32: 0x5348454c
   SHEL
D32: 0x4c545241
   LTRA
D32: 0x43450000
   CE<0x00><0x00>
FLAG_TS:    TIMESTAMP: 584791876352
```

Figure 2: Output of the *mem_parser* Tool

As shown in Figure 2, the channel ID of the message is displayed at the top while the master ID of the message is never displayed. The *mem_parser* tool only displays a master or channel ID when they are changed, which means the master ID of a message would not be displayed if this message belongs to the same master as the last message. Although it is possible to locate the source of a message in the entire output, it is inconvenient for reading sources of multiple messages at a glance. Additionally, the mem_parser tool is not able to display any channel or master names, and names are significantly more descriptive than numbers.

The mem_parser tool displays a timestamp at the beginning of a message and sometimes also at the end of the message. It normally displays the timestamp with a flag like shown in Figure 2, but it also occasionally prints the timestamp along with the first line of message content. Finding the corresponding timestamp for a message can be confusing because of their inconsistent locations. Other than that, the *mem_parser* tool displays timestamps in either decimals or hex without units. This could be confusing for the users because they would not be able to see the time elapsed between each message without doing some calculations. It would be significantly easier for the users of this tool to be able to see the timestamps in a uniformed format.

The most inconvenient part of the *mem_parser* tool is that it displays the content of a message in multiple lines, generally four lines with four bytes in each line. As shown above in Figure 2, the message "SHELLTRACE" is divided into three lines with four bytes in each line. Therefore, the user would have to visually concatenate the separate lines into one line in order to read the content of the message. It would be much more convenient for the users if the content was simply displayed in one line.

3.2 Methodology

The following section describes the planning process in completing this project. This includes our motivations and approaches to the project requirements, the ideal output format, and some suggested extra features.

3.2.1 The *stp_parser* Tool

Our goal was to develop another parsing tool, the *stp_parser*, to parse the output of the *mem_parser* tool into a more human-readable format. Our *stp_parser* tool was designed to resolve many of the drawbacks of the *mem_parser* tool on displaying the source, timestamp, and content of each message. In order to achieve our goal, we consulted with numerous developers on the Tegra Software System Team to understand their perspectives and determine what would be a better output format.

3.2.2 Ideal Output Format

We spoke at length with the main developer who would be using our parser. He mentioned that he frequently uses Excel to process debug output, so it would be convenient to get output in an Excel-compatible format. We chose to support CSVs due to their widespread use and ease of implementation. Based on the developer's input, the output of our parser would display the source, timestamp, and content of a message all in one line. The timestamp of a message was required to be displayed in a uniform format with seconds as units. We wanted the output of our *stp_parser* tool to be significantly shorter than the output of the *mem_parser* tool so that the users would be able to see more information in each line and see more lines without scrolling. This data would also be easy to manipulate in Excel.

3.2.3 Naming Conventions

Although the masters and channels have unique IDs, they also have names. A few developers on the team suggested that we display the master and channel IDs along with their names since names are significantly more descriptive and recognizable than numbers. For example, the master ID 98 actually corresponds to the BPMP module, which stands for “Tegra Boot and Power Management Processor”. It would be more descriptive and easy to skim if the output displays a message from “the main channel of the Boot and Power Management Processor” rather than “channel 0, master 98”. Therefore, we planned to map certain IDs to certain names and display both of them.

Sometimes different Tegra chips have a different set of ID to name mappings and new versions of Tegra chips may have new mappings. Because of this, users could disable the mapping feature or customize the mapping feature by providing their own mappings.

3.2.4 Filters

The output of a trace file can be large since the Tegra chip has multiple processors that share the same bus. However, the users are often only interested in the output messages from a few certain masters or channels. Therefore, it would be convenient if our parser tool has a filtering feature.

We provide both inclusive and exclusive filters in our tool. The inclusive filter would be used when the user only wants to see the messages from a few certain channels from a certain master. The exclusive filter would be used when the user wants to see all the output messages other than those from a certain master. The filters are mutually exclusive because they do not make sense when used together.

3.2.5 Typo Handling

Given the fact that not every developer is familiar with the output message source names and people can sometimes make typos, a few developers on the team suggested we make a typo handling feature on the filters. We planned to compare the master names in the input filter against the master names in the mappings to decide if there is a typo. Then, we would apply the filter on the intended master name and display a warning message to inform the user about how we dealt with the potential typo.

3.3 Implementation

The following section describes the details of how we parsed the output of *mem_parser* into the ideal format, how we implemented the extra features, and how we handled different possible use cases.

3.3.1 Parsing Messages

We parsed the output of the *mem_parser* tool line by line using a for loop and stored the information in a named tuple called *Message*. The fields of the named tuple are *master_id*, *master_name*, *channel_id*, *channel_name*, *time*, and *body*. For each block of message content in the *mem_parser* output, shown in Figure 2, we concatenate them into one string and store it into the *body* field.

Since the master and channel IDs are only displayed when they are changed, we stored the current IDs in local variables, updated them along the for loop, and appended them to the master ID and channel ID of each message. If a master ID was changed but the channel ID is not specified, we set the local channel ID variable to be 0 by default. We also converted the IDs to names and appended them to the master name and channel name of each message. If an ID could not be mapped to a known name, we displayed the ID with the name *unknown* by default. The details of the implementation of naming conventions are described in Section 3.3.3 below.

We converted all forms of timestamps into seconds and stored them in the *time* field for each message. After converting them into seconds, we noticed that some timestamps only show the changed bits compared to the previous timestamp. This could be confusing and misleading because there was almost no way to tell what a timestamp describes. Therefore, we stored the timestamp of the previous message while looping through the *mem_parser* output and displayed every timestamp to indicate the time passed since the beginning of the trace.

```
98,BPMP,3,secondary,73.98978176,"SHELLTRACE"
```

Figure 3: Output of the *stp_parser* Tool

As a result, we were able to display the source, the timestamp in seconds, and the content of the message in one line. Figure 3 shows a sample output of our *stp_parser* tool on the same message as the *mem_parser* tool parsed in Figure 2. The content of the message, "SHELLTRACE", is displayed in one line so it is more readable. The timestamp indicates that this message was printed about 74 seconds after the beginning of the tracing process. The source of the message shows that it came from master 98 "Boost and Power Management Processor" and channel 3 "secondary".

3.3.2 Arguments

In total, we support seven command line arguments implemented with Python's argparse module, a parser for command-line options and sub-commands. The only required argument of the tool is the *input* argument. The user would have to specify the location of the binary input file, which is the output generated by CoreSight STM. Without other optional arguments, our tool simply calls the *mem_parser* tool to parse the binary input file, parses the output of the *mem_parser* tool into CSV format, and displays the CSV format output. The command `python3 stp_parser.py sample_inputs/sample.bin` would run both parsers and display the output of the binary file specified in the location.

We also provided the user with an option to see output similan to syslog format instead of the CSV format by the optional argument *-p* or *--pretty*.

We have provided the users with both inclusive and exclusive filtering features with arguments *-i* or *--include* and *-e* or *--exclude*. The user would have to specify a filter range and apply either the include or exclude flag to see the filtered output.

If the user chooses to use a different name mapping instead of the default one, they can specify a mapping file location using the *-m* or *--map* flag. However, the mapping has to be in a specific format that we support. The format and functionality are further described in Section 3.3.3 below.

If the user chooses to input an output file of the *mem_parser* tool instead of a raw binary output file of the CoreSight STM tool, they can indicate that the input has been preprocessed by the *mem_parser* tool using the *-t* or *--processed* flag. For example, the command `python3` `stp_parser.py sample_inputs/sample.out -t` would only run the *stp_parser* on the file specified in the location.

The user is also allowed to use the *mem_parser* tool in a different location using the --*mem_parser* flag and passing in the location of the tool.

3.3.2 Input File

Before parsing, the script ensures our *stp_parser* runs on the output of the *mem_parser* tool. The script first checks if the specified path exists and exit with an error message if not. Then the script would look for the -*t* or --*processed* flag. If the flag was present, we would simply run the *stp_parser* tool on the file. Otherwise, the script would look for the --*mem_parser* flag and run the *mem_parser* tool before running the *stp_parser* tool. The script would display an error message and exit if the user passed a preprocessed file without using the --*processed* flag or vice versa.

3.3.3 Mappings

As mentioned earlier, each channel in a master has a unique ID and name and there are multiple channels in a master. However, two channels may have the same ID but belong to different masters, which results in them having different names. Therefore, our goal was to map a master ID to a master name and numerous pairs of channel IDs and channel names.

```
mappings= {
    98: {
        'name': 'BPMP',
        'channels': {
            0: 'main',
            3: 'secondary',
            6: 'bootup'
        }
    },
    34: {
        'name': 'CCPLEX',
        'channels': {
        }
    }
}
```

Figure 4: Mapping Format

We developed a format of name mapping as shown in Figure 4. There are three kinds of Python dictionaries in total, the *mappings* dictionary, the *master_channel* dictionary, and the *channel_dict* dictionary. The *mappings* dictionary is the main dictionary that is used by our main script. Inside the *mappings* dictionary, each master ID is mapped to a *master_channel* dictionary. As shown in Figure 4, the *master_channel* dictionary consists of a name string and a *channel_dict* dictionary. The name string represents the master name corresponding to the master ID. The *channel_dict* dictionary contains the channels of that master and maps each channel ID to a channel name.

In order to convert the master IDs to master names, the script looks for the ID in the *mappings* dictionary and finds the name string mapped to that ID. In order to convert the channel IDs to channel names, the script first goes into the *master_channel* dictionary corresponding to the channel's master ID. Then the script goes into its *channel_dict* dictionary to search for the name string mapped to that channel ID. If the ID was not found, the script sets the name to "unknown" by default.

Since we also allow users to use their own mappings, we also implemented a check on the format of the input mappings. The input mapping has to have the exact same format specified above. However, the *channel_dict* is allowed to be empty. Otherwise, the *stp_parser* displays an error message and exits.

3.3.4 Filters

We implemented the filter to support both names and IDs. We developed the filter in a format that master names or IDs are separated by commas and channel names or IDs are surrounded by square brackets right next to the corresponding master. For example, *--include BPMP[main,3],34* means include every output message from channel *main* and channel 3 of master *BPMP* and include every output message from master 34. If the same filter range is applied with the *--exclude* flag, it shows every output message in the file except for the ones in the filter range. We also support exclusive filters on channels. For example, *98-[0]* means include every message from master 98 except for the ones from channel 0. An error would be caused if the input filter did not conform to this format.

Assuming the user passed a valid filter, the script parses the filter into a dictionary to decide whether a message should be printed. The dictionary maps a master ID or master name to a list of channel ID or channel names.

3.3.5 Typo Handling

We handled typos by comparing the input master names against the master names inside the *mappings* dictionary. We have implemented a function to calculate the minimum editing distance between two strings. We calculated the minimum editing distance between the input master name and every master name in the *mappings* dictionary. If the user has entered a correct master name in the filter, the function finds the matching master name, which would have a minimum editing distance of zero. Otherwise, the script finds a master name in the dictionary that has the least minimum editing distance with the input master name, which would most likely be what the user intended to type but made a typo. When a typo was found, the script fixed the master name in the filter dictionary to be the intended master name and applied the filter. For example, if the user accidentally typed “BPML” but intended to type “BPMP”, our tool would display a warning message “WARNING: Master name ‘BPML’ does not exist. We assume you meant ‘BPMP’”. If the user-intended master name did not exist in the *mappings* dictionary, they would have to either check if the master name was valid in CoreSight STM or use their own *mapping*, which is supported by our tool.

3.3.6 Unit Testing

We have achieved 100% code coverage with unit testing. It includes error handlings and warning messages. This helps ensure that every feature described above works exactly as intended.

3.4 Conclusion

Our resulting code was submitted on GitLab and was reviewed and approved by the developers on the Tegra Software System Team. We were able to include all desired features as of now without necessary future work.

4. The Coverity Scan Project

The following sections will detail our work on the Coverity Scan project, which involved the translation of a static code analysis script from Bash to Python. This script was initially written in Bash when it consisted of a few commands but later grew to include configurations for several codebases. We separated this configuration from the core logic so that updating configurations could happen without changing the core logic. To achieve this, we rewrote the script in Python for its clearer syntax and greater access to libraries over Bash.

4.1 Background

In addition to graphics card design, NVIDIA also works on autonomous vehicle systems. Because of the speed with which vehicles must respond to events, these are real-time systems, and are largely written in C. While C provides performance advantages over almost any other language, it is not without its downsides. Manual memory management in particular provides a vector for many serious errors to occur that are difficult to later detect. Additionally, the pattern of passing structs to functions that can mutate them is also prone to errors. An incorrectly implemented function could corrupt data unrelated to it without immediate detection.

To address these issues, the Motor Industry Software Reliability Association was created by the government of the UK. This body put forth standards that code would have to comply with in order to be allowed to run on vehicle systems. MISRA reduces the allowed complexity of functions to make them easier to test and reduce the likelihood of bugs, and also mandates immutability to be used when possible to prevent accidental mutation, among many other rules.

NVIDIA previously scanned codebases that must comply with the MISRA standard using a Bash script. This is a highly configurable script which can take over a dozen flags to change the location of the code being scanned, the type and mode of the scan, and what happens to the output. This code can optionally be uploaded to a local server in order to perform additional checks.

While this script worked well initially, there are a number of issues with Bash that make it difficult to write large programs with it. Specifically, the data structures it supports are fairly primitive and it has no standard library. Another issue with the initial script is the lack of separation of concerns. Almost all variables were defined at the top of the file and then later overridden when needed. Additionally, the configuration for different code bases was done as a series of piecemeal if statements and variable overrides, so updating any configuration or the script as a whole was error-prone.

We were given the task of converting this script from Bash into Python. Python has many advantages as a scripting language for larger projects. For example, it includes a standard library and is able to perform tasks such as reading and writing to and from files without relying on calling external programs as is necessary in Bash. Additionally, its syntax is considered much more legible as it does not rely on flags that need to be memorized for many of the parts of the language.

4.2 Methodology

To begin our conversion of the program into Python, we met with one of the product owners of the script, Adeel Raza. He requested that we rewrite the program in Python, and also introduce a simpler configuration system that was separate from the program as a whole.

Our group converted most of the program line-by-line into Bash, making further optimizations once the core logic was tested to work. We also planned to use the *argparse* library in Python in order to parse parameters. *Argparse* is a library that makes it simple to specify allowed options for parameters, create short and long flags for the parameters, and add help text. The resulting object can then be used throughout the program, and error handling and the display of help text is handled automatically by the program.

A large part of one file was dedicated solely to checking parameters and setting defaults, so we hypothesized that the readability of the code could increase substantially while the chance of error would go down.

To work on this project, our group created a separate repository on NVIDIA's internal GitLab instance. That allowed us to commit code frequently without polluting the repository in which the original Bash code resided. We later copied the completed script to the original repository and submitted a patch as requested.

For the development itself, we used Visual Studio Code's LiveShare extension to allow the entire group to work on the software. Because so much of the initial translation would be line-by-line, we parallelized the process initially.

4.3 Implementation

The implementation went mostly as planned. We began, as we had anticipated, by going through the Bash code line-by-line and performing a mostly direct translation. We did encounter issues that we did not anticipate as the initial code was complicated.

In order to separate the configuration from the code itself, we devised a system where standalone files would be stored in a special directory containing configuration information. Each file creates an instance of the Configuration class we designed. Based on feedback from the users, we updated this class to have default values so that the configuration files were much shorter and showed only the ways in which they differ from the default. Because these configuration files were still class instantiations, it would be easy to catch any misspelled or missing configuration options and prevent bugs that would be hard to catch.

One issue with the configuration files is that sometimes they needed to use variables that were not available until runtime. For example, many locations would be specified relative to the working directory of the program, the installation location of a tool, or the location of a repository. To resolve this problem, we added support for using Bash-like variables. These variables were preceded with a dollar sign and surrounded by curly braces, and their names were always capitalized (as is the convention in Bash). For example, the variable for the working directory was ${WORKING_DIRECTORY}. These variables could be used in any of the strings that were configuration files and would be replaced at runtime with their intended values.

The Bash variable solution was not without issues. Namely, typos in the variable names were not caught, since they were treated as strings. However, a more complicated scheme would involve passing variables between files at runtime and would result in non-idiomatic import patterns. Because of the relatively small number of variables, we did not feel that it was worth the additional code complexity in this case.

We did use the *argparse* library for parsing parameters as planned. This was straightforward to implement and was done quickly. It also allowed us to easily add long flag support to the program, something that had been desired by the team but was difficult to do previously due to the limitations of Bash. However, the help text for each option became long, so we moved all help text to a different file. This text was imported and passed to the *argparse* library, but significantly reduced the amount of visual space taken up by the parameter parsing which in turn made the code more readable.

Our main program scanned the configuration folder on startup and located the relevant configuration file for the desired scan type (this was specified as a parameter). The file would then be imported and the variable replacements would occur. This is a flexible pattern as the external files can be added, removed, or modified without having to change the main logic. Because the configuration files only contain values and not logic, in our experience any issues could be easily traced back to an incorrect value.

4.4 Conclusion

Our resulting code was submitted as a patch to NVIDIA’s code review system, and reviewed and approved by users. We were able to fully replicate the original functionality of the Bash script with cleaner syntax. In particular, the argument parsing was improved in clarity and code complexity.

We were also able to find a few minor bugs in the original script that likely went unnoticed because of the unclear syntax in certain places. We refactored the code in order to make it more readable, but in some places, Bash commands with pipes are still run as the functionality would be difficult to replicate in Python.

Our code is also more portable than the Bash script since Python is standardized language while the available features of Bash can vary by system. More importantly, this code is logically split into modules, making it easy to extend when necessary. This was a key objective of the project because of the difficulty encountered by teams when making modifications in the past.

4.5 Future Work

While we were able to achieve feature parity with the existing script, many new features were proposed by the users that could not be implemented due to time constraints. Even though we did not implement these features, many of them could be more easily implemented now that the script is easier to modify.

One suggested feature was the automated retrieval of results from the Coverity server. Scan results are committed to a server optionally upon the completion of the scan so that further analysis can be done. However, the results of this analysis must be retrieved manually by visiting a webpage and using a GUI to find them. It would be more simple for the developers for this analysis to be retrieved automatically to make the whole process more automated. Because the script is now written in Python, we anticipate that it will not be difficult to make a GET request to the local server and parse the results. This would, however, take a significant amount of work in Bash as there is no built-in way to make requests to an address.

Another requested feature was to include a command for the pre-scan environment configuration. Some scans required certain commands to be run before the scan itself could be run. This can include deleting certain files from old scans or setting certain environment variables. While these are documented in the NVIDIA wiki, it would be faster to ask the script to automatically run the necessary commands. We do not anticipate that this would be a time-consuming task.

Lastly, another feature that would be good to include in the future is a feature to compute the difference between the results between the last scan and the most recent scan. This would allow developers to more easily track how much progress has been made in the process of MISRA compliance. A developer on the team has already created a script to do this in Python, so it can be integrated with our script.

5. The Latency Measurement Project

The following sections describe our latency measurement project, in which we developed a Linux kernel module for Tegra. Many Tegra chips contain a module that optimizes hot spots in code in real-time called Dynamic Code Optimization (DCO). Currently, a tool called *cyclictest* is being used to analyze the latency that DCO introduces. A drawback to using *cyclictest* is that it runs in user space, which means that it is prone to additional latency unrelated to DCO. This project replaces the existing tool with one that can run in kernel space. This new module collects more accurate data on DCO latency that can be used to diagnose anomalies so that they can be fixed.

5.1 Background

Some Tegra chips contain a module called Dynamic Code Optimization (DCO). DCO is able to analyze frequently used sections in the code of an application and performs various optimizations. DCO takes time to optimize the ARM instructions, and at times certain events can slow the DCO down. For example, if the code itself changes then all of the optimized code needs to be invalidated. Because DCO is used in real-time scenarios, such as self-driving cars, it is important to keep latency to a minimum.

Measuring DCO latency is more complicated than just measuring the runtime of a function call; DCO triggers sporadically and at any point in program execution, and so measuring its latency is non-trivial. However, there are certain events that invoke DCO to run, such as bring an application back into context. The current approach for measuring DCO latency is to run a tool called *cyclictest*. *cyclictest* is a standard tool used in Linux to measure the time between when an event is called and when it is executed, and this scenario triggers the DCO to run. All that *cyclictest* does is it gets a high-resolution timestamp, calls sleep for a specific amount of time, and then measures the amount of time it takes for the thread to wake up following the sleep time. For example, if a process sleeps for 1 second and it takes 1.2 seconds for the thread to wake up, DCO ran for 0.2 seconds.

The process for measuring latency on the board can be improved to better distinguish between DCO interference and other sources of latency. A disadvantage of using *cyclictest* for measuring DCO latency is that the test runs in userspace. Userspace applications are not well suited for precise latency measurement as there is overhead and variability in how the kernel decides to schedule processes and handle interrupts. Common sources of unrelated latency are preemption and disabled critical sections in the kernel. Overall, this setup does not accurately measure DCO latency, as additional overhead and occasional unrelated latency spikes cause the results to be unreliable.

A better approach is to test it in kernel space, which allows more control over the environment and less kernel overhead caused by managing user space applications. Additionally, the user could isolate the test to a specific CPU core and remove the CPU core from task scheduling, thus preventing other applications on the same core from causing context switches. The kernel contains a module for scheduling periodic interrupts, which can also be disabled for the code. Overall, moving the test into kernel space allows DCO to be measured more directly and with higher accuracy.

5.2 Methodology

To improve the latency measuring process, we created a Linux kernel driver to perform latency measurement. We first did preliminary research on how to interact with a Linux kernel module and learned that using the *sysfs* virtual file system is the idiomatic approach. However, we quickly encountered a few peculiarities working in kernel space. For one, *malloc* and *free* were not accessible to allocate memory in kernel space, so instead *vmalloc* and *vfree* were used. Additionally, the process of spawning and managing the lifetime of a kernel thread was much different than the user space equivalent *pthread*, and involved manually waking up the kthread process. Finally, only a subset of high granularity time functions were accessible in kernel space, and so we used the *getnstimeofday* function.

The approach for measuring DCO latency in kernel space is different than in user space. In the kernel module, latency is measured by rapidly gathering timestamps in a while loop, so that whenever DCO periodically runs, the latency is higher than the baseline. DCO latency timings are stored in buckets that cover different ranges of timestamps, due to the limited memory space available in kernel modules. For example, if the tool is configured with 10 buckets a maximum range of 100 nanoseconds, then each bucket will cover a range of 10 nanoseconds. If the tool measured the latency of 55 nanoseconds, the sixth bucket count would be incremented, which covers the range of 50-60 nanoseconds.

5.3 Implementation

The first step to ensure the latency measurement kernel module was as accurate as possible was to first isolate the kernel thread from any background tasks that can cause additional latency. The environment for the test was isolated to a CPU core using the kernel variable *isolcpus*, which only allows processes to run on isolated CPU cores if their thread affinity is set to run on it. Additionally, using the Linux *procfs* interface, the default smp_affinity mask was modified so that no interrupts would be received on the test CPU core.

The test parameters are configured using the *sysfs* interface. The user can set the maximum latency measured using the 'max_range' file and the number of buckets to store latency ranges in using the 'num_buckets' file. To run the test a specific number of times, the user writes the number of test iterations to a *sysfs* file called 'run_test'. The user can see if a test is currently in progress by reading from a *sysfs* file called 'status', and if need be, a test can be cut short by writing to the file called 'stop_test'. Once the test is done running, the timing output can be read from a file called 'timings'.

A companion Bash script was made to set up the environment before running the test, including turning off thermal throttling. The Bash script included a segment where the user can set test parameters, including the file where results are stored. An additional Python script was made that takes the data from the test and plots a histogram. The Python script takes in the timing data as a file, removes and all trailing zeros in the timing data, and generates a plot using the *matplotlib* library. The histogram allows developers to better visualize the DCO latency, any anomalies, and measure changes between different DCO code versions. An example histogram generated is shown in Figure 5. It was generated by calling `collect_timings.sh && python3 histogram.py`.

Before running the Bash script, the user has to insert the DCO latency kernel module into the kernel using *insmod* and configure the test parameters in the Bash script. The Bash script runs the tests and displays a message once the test has completed. Then, the Bash script writes the timings to the specified file. The last step is to run the Python script with the timings file as an argument, which will create an interactive histogram containing the timing data.

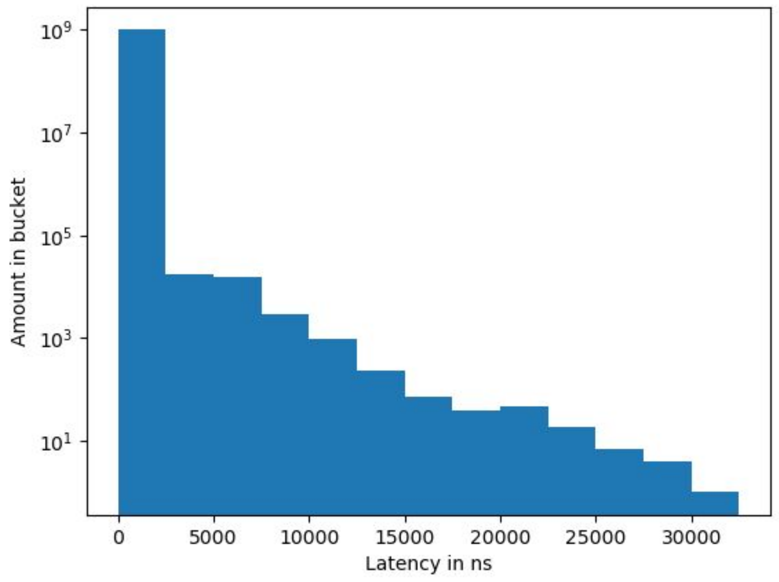

Figure 5: DCO Latency results

5.4 Conclusion

Overall, we developed a Linux kernel module that allows developers to more accurately measure DCO latency, test latency, compare latency, and see anomalies in DCO latency behavior. We added all of the required features for the project. However, due to time constraints, this project was not fully code reviewed by a developer on the Tegra Software Systems team. Future improvements may include cleaning up the code and tighter integration between the Python script, Bash script and kernel module, such that the test can be run seamlessly.

6. Atomics Performance Project

The following section describes the Atomics Performance Project, in which we measured the relative performance differences between a version of *libc* with atomic instructions and one without. Many teams at NVIDIA do not use binaries compiled with atomic instructions, and may be unaware of the potential benefits of changing how the binaries are compiled. The goal of our research was to examine the benefits of compilation with atomic instructions and find situations where they perform better. This way, teams can decide whether any possible benefits are worth the effort of changing the compilation process.

6.1 Background

The CPU architecture used by NVIDIA's Tegra SoCs is called ARM. As new versions of the architecture are released, new instructions are sometimes added. In order to take full advantage of these newly added instructions, the compiler must be specifically targeting the new architecture as it must not include these instructions for older chips that do not support them.

As of ARMv8.2, a group of instructions was added. These instructions allow atomic operations to happen in a single instruction. This includes common tasks such as *compare-and-swap* where a value needs to be examined and then set without any context switches between the two steps. Before the introduction of these new instructions, structures such as locks and mutexes had to be utilized. These structures could result in particularly slow code if there was a long critical section that was interrupted and had to be rolled back and restarted. With a single instruction, this cannot happen.

In order to take advantage of these new instructions, *libc* has to be recompiled. In order to do this on a large scale throughout the many teams that use *libc*, this effort would need to be justified. While it logically follows that the single instruction is faster, there is no concrete data that can be presented to teams to convince them to perform this recompilation.

Our task was to run benchmarks that would showcase the purported performance benefits of the new instruction set by running them both with a *libc* that had the new instructions and another *libc* binary that was not. We could then format the data in a presentable way and use it to encourage teams to prioritize the recompilation of *libc* targeted towards the new instruction set.

6.2 Methodology

We investigated a benchmark called GFXBench on NVIDIA's internal wiki to learn more about how it must be run. We wrote a Python script that would automatically run the benchmark a number of times both with atomic-support *libc* and the older version. We could then compare the results to see if one performed better than the other and if so, whether this difference was statistically significant based on the variance between them.

6.3 Implementation

We examined the libraries that were linked to the benchmark at runtime in order to determine the exact version of *libc* in use. Once we determined this, we downloaded the source code from GNU and recompiled *libc*. In order to compile *libc* with support for the atomic instructions, we had to pass a special flag to *gcc* as we built *libc*. Several times we did not enter the syntax correctly and we accidentally compiled *libc* without the support for instructions that we needed. To ensure that the instructions were present, we examined the assembly of the compiled library using *objdump* and ensured that these new instructions were present.

We also researched how to use different versions of *libc* in the same system without a complete reinstall every time. We learned about an environment variable called LD_LIBRARY_PATH which forces the running program to look in a specified directory for its shared objects rather than checking the default location. Once we set the environment variable to point to our newly built *libc*, we examined the benchmark at runtime again to ensure that it was loading the correct libraries.

We wrote a simple script in Python which simply set the LD_LIBRARY_PATH environment variable accordingly and then ran the benchmarks. Rudimentary parsing was then performed on the benchmark results by the Python script, and these results were stored in a file. Specifically, we called the grep utility on the output to find the number of frames per second (FPS), which is the metric that mattered to us. We then used echo to append the FPS information to the end of the results file.

We initially found that the FPS count was extremely low both with and without atomic instructions. Upon further investigation, we discovered that our board was flashed with a debug version of the kernel and we were advised to reflash it with a release version of the kernel. After the reflash, our FPS rates were similar to those documented in the internal NVIDIA wiki.

Many of the benchmarks included in GFXBench were GPU-limited tasks, so we narrowed down the benchmarks to four specific tests: gl_driver, gl_driver_off, gl_driver2, and gl_driver2_off. We proceeded to run these benchmarks 50 times. Once the data was collected, we used Python’s *matplotlib* library in order to graph the averages as well as the standard deviations.

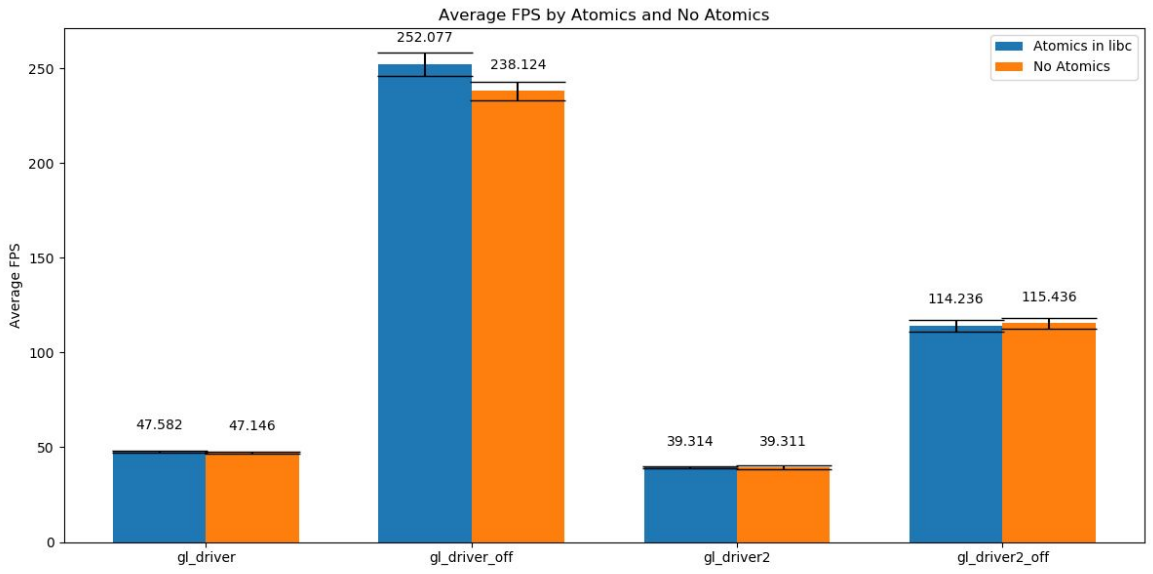

Figure 6: GFXBench results with and without atomics in libc, averaged over 50 runs

Figure 6 depicts the average number of frames per second for the version of *libc* with atomics compared to the version without. The only notable difference in frames per second is in the gl_driver_off test, which shows modest increase in performance with a roughly 5% increase. All of the other tests show the performance being roughly the same, with the atomic and non-atomic libraries performing within 1% of each other.

6.4 Conclusion

The atomics helped performance in the gl_driver_off test. We performed a t-test to verify this, and found that this difference was over 99.8% statistically significant. While this is certainly better than no improvement, the change in FPS was only about 5%. The teams will need to determine whether they believe that the performance increase gained from this change is worth the effort of a recompilation of *libc* and potentially maintaining multiple branches of the code (one branch for earlier processors and one for processors that support the atomic instructions).

6.5 Future Work

One potential test that could be run is testing the variation of performance between boots. The *libc* binary is optimized by the processor and then typically cached for the entirety of the boot because it is used so often. As a result, the quality of the optimization may affect the performance of the benchmark regardless of the atomic instruction set being present. To better understand the results we obtained, it would be helpful to see how much they vary across boots. We were able to write a Python script that we believe will execute these tests, but due to limited time, we have not been able to verify that the program works, nor have we been able to collect any data for this.

An additional task would be to help other teams use a newly compiled version of *libc* in their codebases. Teams may be more willing to adopt the newer instruction set as a compilation target if they are given help with the initial work rather than having to divert resources.

7. Conclusion

Our tools have made numerous internal processes easier for the developers on the NVIDIA Tegra System Software Team. We were able to work on both high-level and low-level languages while directly interacting with the members on who would be using our tools in the future. Our *stp_parser* tool was able to improve their debugging by providing them with more readable and more descriptive debug output. The Coverity Scan project has simplified the code used to run the scans and separated the configuration from the core logic. The DCO Latency project provides tools for developers to more accurately measure the latency of the DCO module of the Tegra chip and create a histogram to find outliers with ease. The atomics project provides NVIDIA developers with data detailing the implications of updating user space libraries with the new ARMv8.2 atomic instructions. Overall, our contributions to the toolset at NVIDIA have added functionality that was previously unavailable, and automated tasks that previously had to be done manually.

8. References

[1] "The AI Computing Company." NVIDIA,

https://www.nvidia.com/en-us/about-nvidia/ai-computing/

[2] Arm Ltd. "CoreSight Debug and Trace." ARM Developer,

https://developer.arm.com/ip-products/system-ip/coresight-debug-and-trace

[3] Arm Ltd. "System Trace Macrocell." ARM Developer,

https://developer.arm.com/ip-products/system-ip/coresight-debug-and-trace/coresight-component
s/system-trace-macrocell

[4] Zhang, Chunyan. "System Trace Module (STM) and its usage." Linaro,

https://www.linaro.org/blog/stm-and-its-usage/