An Evaluation of Valgrind Performance on Commercially Available Maker Devices

Greg Cusack Kunal Patel Nathan Pilbrough Tzu-Wei Chuang Zhengshuang Ren

University of California, Los Angeles

gregcusack@ucla.edu kunalpatel1793@gmail.com nathanpilbrough@gmail.com
twc5@ucla.edu zhengshuangren@gmail.com

I. PROBLEM DEFINITION

Valgrind is a widely used tool utilized in optimizing C/C++ code. Tools within Valgrind, such as `memcheck`, allow for the detection of memory leaks within source code. However, one of the common criticisms of Valgrind is that it is highly memory intensive at runtime. Therefore, we aim to understand how well Valgrind performs on a variety of maker devices compared to a standard PC by designing a compact script that creates memory leaks on purpose to find baseline performance metrics over all platforms. With this standard performance profile for each platform, we want to then test Valgrind performance on each platform with increasing size and complexity. Ideally, we want to use open source code that is readily available. The research question we are ultimately trying to answer is "Is Valgrind scalable on maker devices?" It is important to understand the scalability of Valgrind on these systems because they can enlighten future developers on performance limitations in common maker devices.

II. RELATED WORKS

Clause et al. implemented a prototype tool called *Leakpoint* to address memory leakage issues [1]. The system was built on top of Valgrind such that it was able to track pointers to dynamically allocated memory and thereby detect memory leaks. As an extension to the Valgrind functionality, the system also directed the developer to the locations where the leakage could be fixed. The authors found that the system was able to detect at least as many errors as the Valgrind system as well as provide an effective means of fixing the leaks. After evaluation it was found that the cost of the added functionality was a runtime overhead of 100-300 times normal operation. This is in comparison to the reported 30 times overhead as a result of running Valgrind independently. *Sleigh* is another memory leak detection tool created by Bond et al [4]. The system tracks the time since last use to find memory leaks. It is also more lightweight than Valgrind and was designed for use during performance runs as opposed to test runs. As a result, *Sleigh* does not pick up as many errors as Valgrind, but only adds 11 - 29% overhead execution time. Sui et al implemented a static memory leakage detector, Saber [5]. Saber differs from the above mentioned detectors through its use of sparse value flow graphs which enable the system to detect memory leaks. The system was able to detect leaks with a false positive rate of 18.7% however runtime performance tests were not conducted.

One related work on benchmarking is BugBench, a benchmark suite that systematically evaluate software bug detection tools. It provides a good general guideline on the criteria for selecting representative bug benchmarks as well as the metrics in evaluating bug detection tools such as Valgrind [3]. The authors found that Valgrind misses stack buffer overflow in their test cases. Therefore, this might be a good direction for us to consider in choosing our benchmarks. Finally, while BugBench did validate the selection of their benchmarks on Valgrind, it was solely on one machine.

Another related work is included in the Valgrind source code. Valgrind developers provide a few performance test cases. These test cases are meant for developers who add new tools to the Valgrind framework and want to evaluate their tool's performance. Two of the notable performance test cases include running `memcheck` on code which compresses and decompresses data and on a small and fast C compiler. However, the output of these performance tests is solely execution time.

III. MOTIVATION

With the current and impending influx of internet-connected, embedded devices, individual device resources need to be allocated properly in order to perform efficiently with limited resources. While competent programmers aim to develop the best possible code, bugs are inherently introduced into the system. Many of these bugs result from memory mismanagement and utilizing more memory than required. Due to the limited resources and speed requirements of many IoT devices, their code bases are largely developed in C/C++. Unfortunately, unlike Java,

C/C++ do not have garbage collectors, so resources and objects allocated and not freed eat up valuable memory. As a result, it is common for software developers to forget to free up program memory and return it back to the operating system. Luckily, developers from all over the world designed Valgrind, an open-source framework for building code analysis tools. Valgrind consists of tools such as `memcheck`, a widely used tool which identifies memory management issues such as memory leaks and other memory related issues. While Valgrind has been used to debug many different projects including topics in graphics, IoT operating systems, and statistical computing, Valgrind is a relatively memory and resource intensive tool; however, to the best of our knowledge, there are no detailed and quantitative analyses on Valgrind performance across multiple platforms with varying resources. The EEclipse team looks to port Valgrind to various common maker devices in order to quantitatively analyze the performance of Valgrind. Our goal is to identify the performance of Valgrind on limited resource devices, in order to prove a popular, complex memory leak checker can be used in the rapidly growing IoT domain.

IV. APPROACH DESCRIPTION

In this paper, performance of Valgrind refers to runtime, scalability, complexity, memory usage, and memory leak identification success rate. Due to the resources currently available to the team, Valgrind performance will be measured on three different maker devices: Raspberry Pi 3, BeagleBone Black, and the Intel Edison. Note that Valgrind requires an OS to run; therefore, Valgrind will not run on Arduino, which is why it is left out of this project. Valgrind performance benchmarks will also run on an Ubuntu VirtualBox VM running on a laptop containing a 2.5GHz Intel Core i7 in order to provide a performance reference frame. For each device, Valgrind performance will be measured on programs of various length and complexity. One of the main challenges of measuring performance of Valgrind is designing proper test cases. The EEclipse team will source and develop test cases which will accurately evaluate Valgrind performance across multiple embedded devices.

The Valgrind tool used in this evaluation is `memcheck`. `memcheck` looks for memory leaks throughout a user's code. When referring to memory leaks, we are referring to memory in a system which has been allocated to a specific program, but is never properly freed and returned back to the operating system. There are two broad categories of memory leaks, lost memory and forgotten memory[1]. A lost memory leak occurs when a pointer to an allocated object is either deleted or set to point to another object. This memory taken up by the object can no longer be accessed and cannot be used by any other process running on the device. A forgotten memory leak, on the other hand, occurs when the memory block is still reachable but is not de-allocated or accessed for the rest of the code execution. Hence memory leaks become a large problem when systems are constantly running and servicing requests. Memory is allocated and never de-allocated; therefore, a system's memory resources are quickly consumed to exhaustion.

V. EVALUATION PLAN

We plan on standardizing the `memcheck` tool and seeing if it picks up different kinds of memory leaks. We will perform this standardization over all testing platforms to see if all memory leaks are picked up consistently. The test script will be developed by this group and will attempt to include all memory leaks discussed in the relevant works section. During this standardization we plan on collecting performance metrics such as execution time. Ideally we observe that Valgrind detects all memory leaks, such that we can assume that success rate is uniform across all platforms, allowing us to soundly compare performance metrics across all platforms independent of success rate. After completing this "proof-of-principle" case, we want to expand this project to more open source systems that are available as test cases. We do not expect Valgrind to find many memory leaks on well-designed systems, but with these systems we can gain insight into how performance varies with different scale code. For example, we may find that maker devices handle our standardization case just as well as conventional PC systems, but as we begin to scale up to larger systems of code we should be able to see performance variations across our test devices. We think that understanding this performance degradation with increasing scale code across a variety of platforms will be an interesting trend to study.

VI. URL

The detailed implementation and our benchmark test script is available at https://github.com/vivi51123/CS230-Software-Engineering. The website features documentation on how to install and use Valgrind. As the website evolves, more documentation and features on the comparison of Valgrind across different embedded maker platforms will be available. We hope this comparison study can provide some insights for developers in the IoT domain.

REFERENCES

[1] J. Clause and A. Orso, "LEAKPOINT: Pinpointing the Causes of Memory Leaks," *ICSE*, pp. 515–524, 2010.

[2] N. Nethercote and J. Seward, "Valgrind", ACM SIGPLAN Notices, vol. 42, no. 6, p. 89, 2007.

[3] S. Lu and Z. Li, “*BugBench*: Benchmarks for Evaluating Bug Detection Tools”, *UIUC*, 2005.

[4] D. Bond and K. McKinley, “Bell: Bit-Encoding Online Memory Leak Detection,” *ASPLOS*, pp. 61–72, 2006.

[5] Y. Sui, D. Ye, and J. Xue, “Static Memory Leak Detection Using Full-Sparse Value-Flow Analysis,” *ISSTA*, pp. 254–264, 2012.

[6] Seward, Julian, and Nicholas Nethercote. "Using Valgrind to Detect Undefined Value Errors with Bit-Precision." *USENIX Annual Technical Conference, General Track*. 2005.