ABSTRACT

D'ARGENIO. MICHAEL JOSEPH. A Low Footprint gprof-based Profiler for Microcontrollers. (Under the direction of Dr. Alexander Dean).

Microcontrollers are a highly constrained platform. They are selected for special-purpose applications to reduce the overhead and keep the cost minimal. Due to their use-cases, microcontrollers are much more restricted when it comes to memory, microarchitecture, speed, processing power, and energy consumption. Because microcontrollers must respond in real-time, performing optimizations can be critical. These optimizations can improve performance and reduce latency to enable the completion of complex tasks on limited hardware. Without profiling, engineers can waste significant development time on optimizations that result in little gain while overlooking opportunities for huge improvement.

Profiling is the key to unlocking critical optimizations for microcontrollers but attempting to profile microcontrollers exposes a paradox. Profiling finds opportunities for improvement; however, profiling can deplete the limited memory and overburden the microcontroller's processor. There are several profiling solutions for microcontrollers in existence. These solutions often require more memory and processing power than the programs can spare. In addition, many of these profilers are only suitable for higher-end microcontrollers with special debug hardware embedded in the chip. These solutions are typically proprietary with little technology transferred. They function only on their intended platforms. As a result, they have differing feature sets that perform with varying degrees of success.

This thesis attempts to shift that paradigm. The proposed framework tears down those silos of information by delivering a solution that works for all microcontrollers and advances the on-going efforts to produce an open-source, processor-agnostic development toolchain. To adapt gprof for a microcontroller environment, this thesis proposes the novel idea of dividing the

profiling responsibilities between the target microcontroller and the host personal computer. By divvying up the profiling responsibilities, it minimizes the overhead on the microcontroller and allows the program to execute unencumbered. The result is a profiler with identical functionality that generates the same, expressive profiles as the original implementation. The profiler surpasses existing solutions by limiting the demand on the microcontroller's constrained resources. This profiler is ripe for adoption and continued growth due to its elegant approach and its embrace of the burgeoning, open-source development effort within the embedded systems community.

A Low Footprint gprof-based Profiler for Microcontrollers

by
Michael Joseph D'Argenio Jr.

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the degree of
Master of Science

Computer Engineering

Raleigh, North Carolina
2020

APPROVED BY:

Dr. Alexander Dean Committee Chair	Dr. Aydin Aysu
Dr. James Tuck	Dr. Ginger Yu

DEDICATION

To my friends.
The ones that chose me.

To my family.
The ones that didn't choose me but chose to stick by me anyways.

Thanks for all of the love and support. And for listening to my gripes…

Dedicated in special memory to Louis Michael D'Argenio.

BIOGRAPHY

Michael D'Argenio was born in Omaha, Nebraska in 1992. He attended North Carolina State University for his B.S. degree in Electrical Engineering. While in school, he participated in the University Honors Program, completed a co-op at Duke Energy, served as the program director at WKNC, and developed the initial ecoPRT prototype. He graduated in 2014 and joined Schneider Electric as a Hardware Developer. He developed and helped manage the North American Electric Vehicle Charging Product Line. He went on to become the University Outreach Program Coordinator and developed modules for a motor starter product line.

In 2018, he returned to North Carolina State University with a goal of transitioning to a career in education. Throughout his time in graduate school, he served as a teaching assistant for the Electrical and Computer Engineering Senior Design Program, completed the Teaching and Communication Certificate through the Graduate School, and performed research and development in open source design and analysis tools for embedded systems under the direction of Dr. Alexander Dean. Upon graduation, he will be working as the Innovation Lab Coordinator at Ravenscroft School in Raleigh where he will be teaching engineering, robotics, and computer science courses.

ACKNOWLEDGMENTS

I would like to acknowledge and thank Dr. Alexander Dean. He was not only responsible for half of my credits towards my graduate degree but also served as a fantastic mentor. He never failed to inspire me, challenge me, and occasionally help reign in some of my ideas and direct me when I needed it. May we one day both find the time to pursue all of the lofty goals and project ideas we have.

I would also like to acknowledge and thank Erich Styger. His blog is an unending fountain of knowledge in this domain and his ideas helped inspire this project.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1 – INTRODUCTION

1.1 Background

Profiling typically refers to some form of dynamic program analysis that can measure a variety of metrics while the program is executing. These measurements can include how long a program or section of code takes to run, how much memory is used, how many times a function or basic block is executed, how much power/energy consumed, and how frequently a section of code is run. Profiling is most often used to optimize a program. By using a profiler, software engineers can identify chunks of code that require further optimization to improve execution time, energy consumption, response time, or any number of other factors. Profiling can also be used to understand how a program behaves or to help identify bugs by leaving breadcrumbs along the execution path.

Profilers can be classified by their type of output. Traditionally, there are two types: flat profile and call graph profile [1]. The flat profile estimates the total execution time of a function across iterations. The call graph profile does much the same thing but can also track the number of times a function was called and the caller-callee relationships between functions. One of the first profilers was a flat profiler called prof included on Unix systems in 1973. gprof was introduced on Unix systems in 1982 [2], [3]. It built upon prof and included a call graph profiler. In 1988, a version of gprof with much the same functionality was written for the GNU Project [4], [5]. It was included as a part of GNU binutils, a suite of programming tools for managing binaries. The GNU version of gprof is the profiler used in this thesis to provide analysis for microcontroller development.

Profiling is usually achieved through instrumentation, statistical profiling, event-based profiling, or simulation methods [1]. The profiler discussed in this application uses a combination of instrumented and statistical profiling. In instrumented code, profiling data is

captured by inserting instructions into the target program. The slightly modified executable is called the instrumented program. The instrumented program should theoretically be identical in functionality to the original target program, but there can be some slight performance reduction witnessed due to the overhead of the additional instructions. Statistical profiling is typically performed via a process called sampling where the operating system interrupts the program at steady intervals to capture information off of the call stack. There are some known shortcomings of statistical analysis that can lead to false conclusions. These issues are discussed in more detail in Chapter 2.

In the interest of taking a comprehensive look at profiling, event-based and simulation profiling methodologies are used in many applications although they are not suitable for the intended use-case of this thesis. These methodologies provide a tremendous amount of insight into execution but require a significant amount of overhead. Event-based profilers require a supervisory operating system that can trap events, catch exceptions, and add hooks for profiling. Simulation profiling requires a high-level hypervisor to read the target program instructions in one by one and interpret it as the processor would. By replicating the processor's functionality, the program can create an accurate profile of the target program. These profiling tactics can be effective, but the massive amount of overhead required makes it impractical for a microcontroller development environment.

This thesis focuses primarily on profiling with instrumentation and statistical approaches to optimize performance for a GCC-compiled processor used in microcontroller applications. GNU gprof provides a flat profile using statistical program counter sampling and a call graph profile using instrumentation hooks to capture the caller function and callee function memory

addresses. This profile creates a rather holistic view of the program to aid in the debug and analysis of microcontrollers.

1.2 Significance of the Study

Profiling is an important tool in any software engineer's repertoire. Profiling can help identify hotspots in a program that are ripe for optimization. It gives the engineer insight into how the program actually behaves versus how the engineer thinks or expects it to behave. Squashing these assumptions and having a reliable profile is crucial to effective optimizations. Donald Knuth said the following about erroneous execution assumptions:

"Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%. A good programmer will not be lulled into complacency by such reasoning, he [they] will be wise to look carefully at the critical code; but only after that code has been identified. It is often a mistake to make a priori judgments about what parts of a program are really critical, since the universal experience of programmers who have been using measurement tools has been that their intuitive guesses fail." [6]

This notion of unfocused optimizations is further driven home by Amdahl's law [7]. Amdahl's law discusses the notion of a theoretical speedup in the execution of a specific program based upon a quantified improvement. It is typically discussed in the context of increasing the number of processors or improvements to the microarchitecture, but it holds true when computing the speedups of an optimization. Equation 1a explains the relationship

illustrated by Amdahl's law. To illustrate the importance of profiling, consider the following example.

Optimization A speeds up a particular function Z by a factor of 5. Function Z consumes 25% of the program's initial execution time. Using Amdahl's law, it is understood that Optimization A would provide an overall speedup of 1.25. Now by performing Optimization B on the same Function Z that consumes 25% of the program's execution time, the program is now sped up by a factor of 500. Despite the 100x speedup of Function Z, Optimization B would only provide an overall speedup of 1.332. This speedup is only about a 6% improvement overall. Equation 1b proves this observation. The maximum overall speedup is bounded by the fraction of execution time affected by the optimization. Even if there was an infinite speedup for this function, its maximum overall speedup would be $1.\overline{333}$.

$$S_{overall}(s) = \frac{1}{(1-f) + \frac{f}{s}} \qquad (Equation\ 1a)$$

$$\lim_{s\to\infty} S_{overall}(s) = \frac{1}{(1-f)} \qquad (Equation\ 1b)$$

$S_{overall}$ – $the\ theoretical\ speedup\ of\ the\ execution\ of\ the\ entire\ optimized\ program.$

s – $the\ rate\ of\ speedup\ due\ to\ the\ optimization.$

f – $the\ fraction\ of\ execution\ time\ that\ is\ benefitting\ from\ the\ optimization\ speedup.$

Amdahl's law demonstrates there is a theoretical maximum to any optimization. It highlights that engineers would be wise to follow the rather trite idiom to "work smarter, not harder." By focusing on the wrong section of code, needless development hours would be sunk into an optimization that provides minimal improvements. Profiling allows the engineer better

insight into where the greatest optimization might lie. A truthful and expressive profile will help the engineer balance optimization gains with development time.

Profiling is incredibly useful in any application; however, it can be absolutely critical in embedded applications. Microcontrollers must meet a very precise set of requirements for a single, special purpose. They are much more limited than general-purpose computers in terms of processing speed, memory, instruction sets, functional units, pipeline capabilities, power, and a myriad of other factors. Profiling is an excellent way to optimize programs to fit within these constraints and to reduce costs.

Despite the strong case for profiling microcontrollers, profiling is rarely performed. There is a paradox at the core of profiling microcontrollers. By profiling microcontrollers, engineers can make better utilization of the limited hardware. However, profiling can be computationally costly. Adding instrumentation can take up precious memory space and reduce performance. Microcontrollers typically have real-time computing constraints and the hardware is not as capable at hiding latencies as it is with general-purpose computing devices. This latency can introduce new bugs into the program by affecting response timing. Other profiling methodologies require even more overhead. There is a significant need for a lightweight and effective approach for profiling microcontrollers.

1.3 Motivation

This crucial need for a microcontroller profiler has been identified by many. There are already several existing solutions that are discussed in detail in Chapter 3. However, many of them only work for higher-end microcontrollers and require lots of available memory or execution time. There is a need for a lightweight, low-cost solution that works across the board for the wide range of microcontrollers that exist today. This thesis lays out one such solution that

will work for any microcontroller with a serial port that can run GCC-compiled code. This solution makes use of the existing gprof framework because of its simplicity, elegance, and widespread adoption. Despite being introduced nearly forty years ago, gprof is still widely used today [2].

Profiling can be expensive in terms of processing and memory usage. This solution makes modifications to the standard gprof implementation to allow it to be used within a microcontroller development environment with as little overhead as possible. This implementation does not require a costly external debugger or debug functionality baked into the chip. It meets the need for an across the board profiling solution for microcontrollers without sacrificing functionality. A low footprint, open-source, hardware-independent solution such as this will greatly reduce the barrier and help to proliferate the use of profiling in microcontroller development.

1.4 Preview of the Work

In order to avoid overburdening the microcontroller, this thesis proposes offloading a majority of the profiling processing to a host PC. This tactic also ensures minimal memory consumption by the profiler on the microcontroller. A one-way serial connection is set up between the target microcontroller and the host PC to facilitate this interface. The target microcontroller is responsible for capturing the profiling data and writing it to the serial port. The host PC is responsible for interpreting that data and creating the profile for the program. By sharing the profiling effort, the microcontroller can utilize its full memory and meet its performance requirements while also providing expressive and accurate profiling data.

The profiling is performed in 4 different stages. First, there is the instrumentation that is performed at compilation. This inserts instructions into the object code to capture the call graph

arcs. Then there is the target microcontroller (MCU) application that triggers the profiling data capture and transmits the information via the serial interface. Next there is the host personal computer (PC) application which receives the data transmission and generates the gmon.out file per the specification and implementation laid out in the GNU binutils specification [5]. Finally, gprof is run and it writes the flat and call graph profiles to `stdout`. The profiling process for this thesis is laid out in Figure 1.

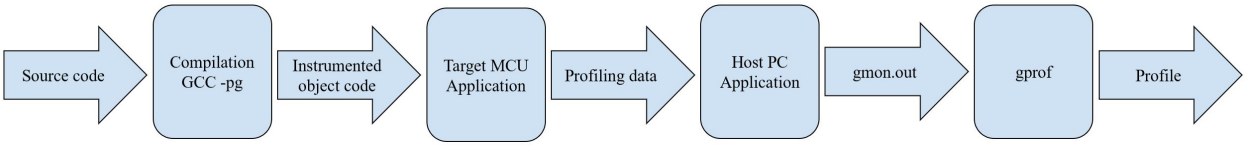

Figure 1 - Profiling Process Flow

To successfully implement this profiling solution, the system must be set up as shown in Figure 2. There is a target MCU and a host PC with a serial interface connecting them. The target MCU can send the profiling data via UART, SPI, I2C, or a USB debug console port based on the specific application. Depending upon the interface chosen and the PC's capabilities, a USB to serial converter (e.g. FTDI FT232) could be required to facilitate this communication.

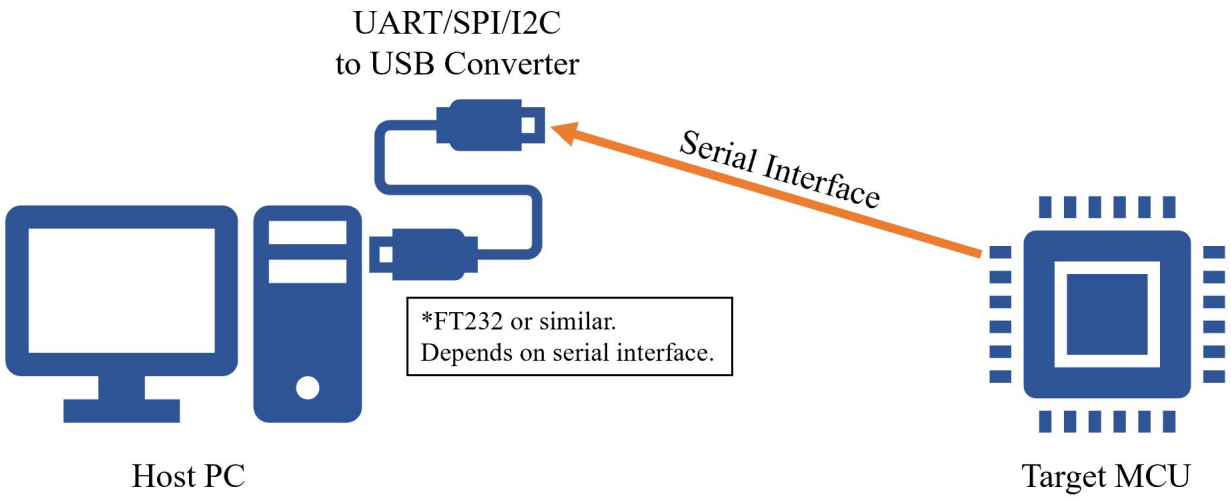

Figure 2 - Profiling Physical Interfaces

This novel approach provides a low footprint solution ideal for microcontrollers while providing the same level of detail as other profiling solutions in more traditional settings. The validation and analysis show this solution accurately captures stack data from the target MCU at runtime. The host PC application is able to effectively create a representative portrait of the program's execution. This solution is proven to be just as effective at finding opportunities for optimization as gprof in a standard C development environment. This implementation provides a rather elegant profiling solution for *all* microcontrollers by leveraging an open-source, tried-and-true solution.

1.5 Outline for Thesis

The rest of the thesis is organized as follows. Chapter 2 discusses some of the drawbacks of a statistical profiling technique. Chapter 3 surveys the related work in the field and discusses the pros and cons of these solutions. Chapter 4 investigates the theory behind this new profiling framework for microcontrollers. It illustrates the design theory of the target microcontroller's profiling program and the host PC's profiling interpreter program. Chapter 5 examines the profiler overhead required on the microcontroller. Chapter 6 describes the validation of the system design. Chapter 7 details how to implement this solution on other microcontrollers. Chapter 8 discusses the conclusions drawn from this work and potential future work that could occur in this space. The code and results for this thesis are provided in a GitHub repository detailed in Appendix A.

CHAPTER 2 – PROFILING CAVEATS

2.1 Introduction

Before discussing this solution any further, there are some important caveats to understand about statistical profiling. In this case, statistical profiling means interrupting the program at a periodic interval to see where the processor is by sampling the program counter. The drawbacks have been thoroughly discussed and are well-understood [5 Sec. XIX], [8], [9]. These limitations do not mean the profiling data is wrong, but they are important to understand to ensure the data is not misused. The profiling data is always correct, but it is not always representative. The profiler does not necessarily lie, but it can lead to false conclusions. The mistruths of statistical profiling can fall into two camps: unintentional synchronization and multiple caller conflation.

2.2 Unintentional Synchronization

Unintentional synchronization occurs when certain functions synchronize with the period of the statistical profiler. For example, say the period of the profiler is set to five milliseconds for a particular program. The program's goal is to record the roll, pitch, and yaw of the device every 5 milliseconds using an accelerometer, a magnetometer, and a gyroscope. A timer interrupt occurs every five milliseconds and triggers the processor to retrieve measurements. At this point, the program performs sensor fusion calculations and sends the data via SPI to be logged on an SD card. The program's timing diagram is shown in Figure 3. The profile would show `Acc_Read()` dominating 100% of the execution time when it really only consumes about 1%. This profile would insinuate to the engineer that they should focus their efforts on optimizing `Acc_Read()`. These efforts would result in an almost unnoticeable performance improvement, and the engineer would likely miss some large optimization wins in `Calculation()`.

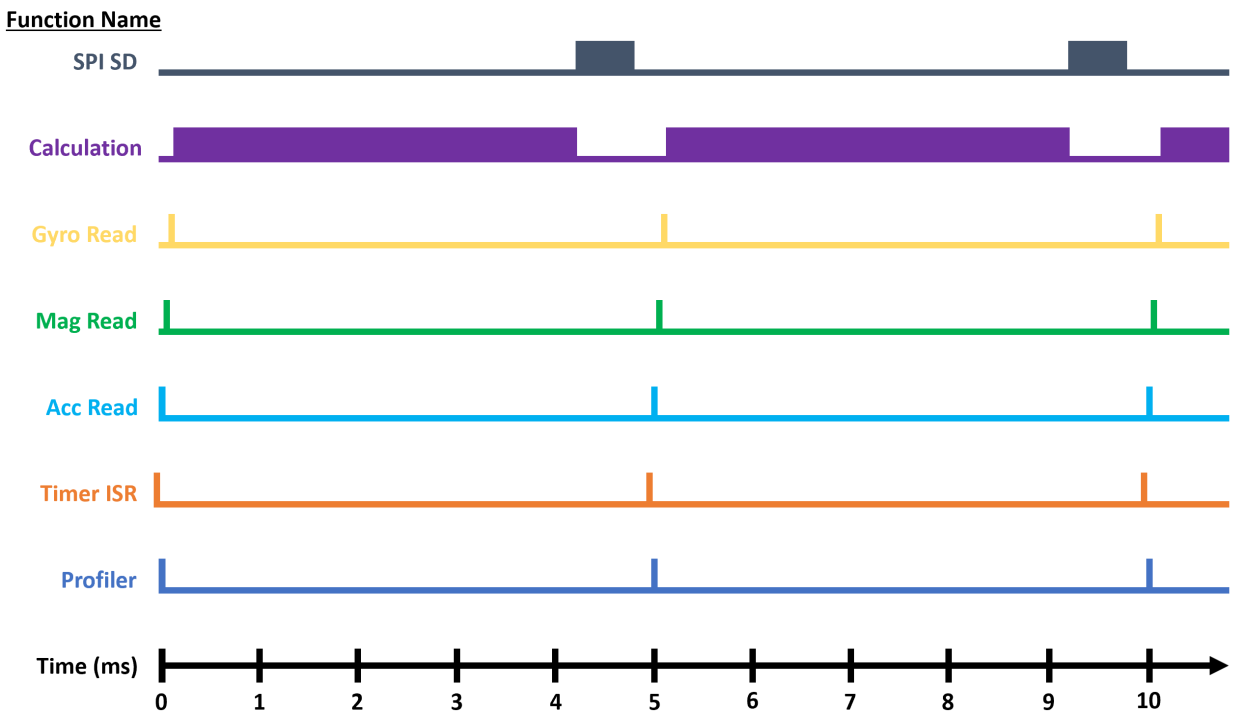

Figure 3 - Unintentional Synchronization Example Timing Diagram

This example is overly simplistic since there is only one process and the periods are identical, but it illustrates the issue with unintentional synchronization. It is important to select a proper period for statistical profiling to ensure it does not synchronize with the program. To be sure there are no issues with the profiling data, it is helpful to modify the profiling period and compare the data from the separate runs. If the function distribution looks the same across the different periods, it is fairly certain that the profile is representative. Programs that utilize a real-time operating system (RTOS) are particularly susceptible to this issue. Many RTOS's check priorities to perform task/thread switching on a certain interval. Depending upon the RTOS implementation and the program counter retrieval method, one hundred percent of the samples could potentially occur in the RTOS handler/hook functions. It is important to fully understand the issues with statistical sampling in such a volatile system to avoid wasting time searching for optimizations in the wrong places.

2.3 Multiple Caller Conflation

Multiple caller conflation can occur when a single function has multiple parent functions. If the statistical profile is triggered and captures a program counter sample in a function with two distinct callers, it is impossible to distinguish which caller invoked the callee function at that point and where the execution time should be attributed [8]. Without understanding the call chain or the effects the different input arguments from various parent functions could have, the flat profile for this function will be meaningless. It is impossible to determine which caller function is utilizing the callee function more and dominating execution times. Since the stack sizes, call depths, input arguments can vary widely, it is impractical to incorporate this information in a lightweight profiling solution for microcontrollers.

```
void doSomething(uint32_t x) {
  for (uint32 i=0; i<x; i++);
}

void fast() {
  doSomething(1);
}

void slow() {
  doSomething(0xffffffff);
}

int main () {
  fast();
  slow();
}
```

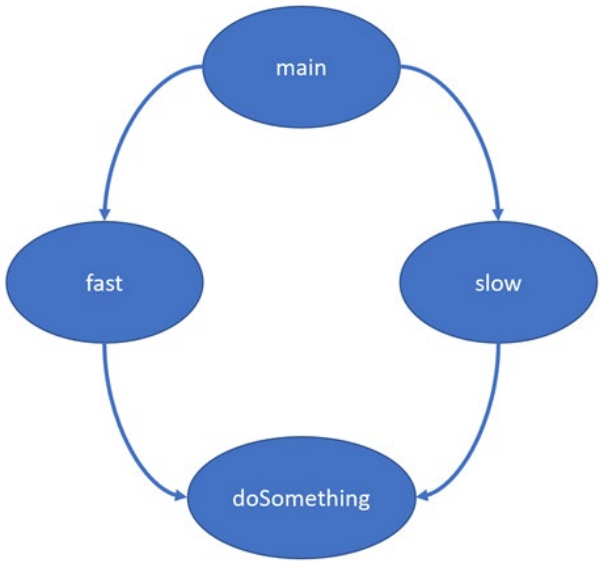

Figure 4 - Multiple Caller Conflation Example: Code and Call Graph

Figure 4 shows an example of such a program. Both `fast()` and `slow()` call the function `doSomething()`, but the duration of `doSomething()` is determined by the argument given to it by its caller function. The loop in `doSomething()` executes only once in `fast()`, but the loop executes over four billion times in `slow()`. Despite this fact, the gprof flat profile for this program shows the execution time split evenly between the `fast()` and `slow()` functions at 4.37 seconds apiece. gprof correctly captures the number of times each function is called and successfully calculates the total amount of execution time for `doSomething()`. However, it makes a false assumption that the execution time is divided up equally between each of the parent functions since each function (`fast` and `slow`) is only called once [5 Sec. XIX]. This problem is highlighted in the flat profile shown in Figure 5.

```
Flat profile:
Each sample counts as 0.01 seconds.
%   cumulative   self              self     total
time   seconds   seconds    calls   s/call   s/call  name
100.00    8.74     8.74        2     4.37     4.37  doSomething
0.00      8.74     0.00        1     0.00     4.37  fast
0.00      8.74     0.00        1     0.00     4.37  slow
```

Figure 5 - Multiple Caller Conflation gprof Output

Understanding the false assumptions made in estimating execution times is an important trick to utilizing the profiler data correctly and avoiding false extrapolations. There are some modifications to the source code that can be done to help gprof output a more representative profile. In the case previously discussed, two instances of `doSomething()` could be created to assist the profiler in associating execution times with the function's parents. Functions can also be aggressively inlined, either in source code or through preprocessor directives, to simplify the call chain depth and improve the resulting profile.

CHAPTER 3 – RELATED WORK

3.1 Existing Profilers

There are several existing profilers including Valgrind, OProfile, gprof, perf, and more. While these profiling technologies may occasionally be discussed in relation to embedded systems, they are not used for microcontroller development. These profilers require a significant amount of overhead and some combination of the standard C library and operating system to run correctly. They work great for an embedded Linux environment; however, that is not the target of this thesis. Porting these solutions as is to run on a microcontroller and provide meaningful insight would be fruitless. The code and performance overhead would be too great. gprof is the one exception to this rule. Generating a profile with gprof is lightweight enough that profiling microcontrollers seems reasonable. Section 3.4 discusses an existing solution in this space. This thesis presents another alternative solution.

3.2 Simulators

As discussed previously, simulators can be used for profiling. Simulators provide a tremendous amount of insight into what is happening inside a processor. They provide program status, variable values, register values, stack information, the contents at every memory address, and much more. However, simulators do not exist for every microcontroller and they can vary wildly in quality and usefulness across manufacturers or even across product families. The main issue with simulators is that they exist several levels of abstraction above the chip they are made to emulate. Since the program no longer runs on its native hardware, there can be some inaccuracies or issues that arise from executing outside of its usual timing constraints. Oftentimes, microcontrollers interface with external peripheral devices such as motors, sensors, and other components. These components provide inputs and outputs to the microcontroller which will either need to be modeled in software or fed into the simulator. Replicating this

functionality and ensuring its accuracy is a large undertaking. It makes simulation profiling for microcontrollers a tough sell in most situations.

3.3 Tracing

Many microprocessor manufacturers will provide some sort of tracing functionality [10]. Tracing is a special method of logging a program's status during runtime using trace or print statements. Tracing can mean different things and provide varying levels of insight based on the specific implementation. It can range from simple print statements reporting values to dumping out the machine code for every single instruction that is run. Tracing is typically used for debugging, but this same information can be exploited for profiling as well. Tracing typically provides a lot of data that can be used to construct very detailed profiles. However, there is oftentimes a lot of preliminary effort to manipulate the trace data to create a working profile.

Tracing is usually implemented at a low-level to keep the CPU burden minimal. The low overhead means there is not a lot of packaging of the messages done before they are transmitted (e.g. the trace dumping out object code instruction by instruction). This practice results in a very high volume of unique trace statements. Keeping the overhead on the target processor minimal shifts the burden over to the receiving end. The host PC must interpret all of these different messages and extrapolate some meaning from them. Tracing is a great tool because it can provide you with a completely exhaustive amount of data about your program's execution; however, the trace data alone does not provide a lot of insight. The data must be compiled and used in a methodical way to provide intuitive analysis and debugging information.

Tracing functionality is often reserved for the higher-end processors. For example, the ARM Cortex-M series of microprocessors intended for use in low-cost microcontrollers only offer this functionality for their M3, M4, and M7 processors [11]. These higher-end processors

provide tracing information through the Instrumentation Trace Macrocell (ITM) and Embedded Trace Macrocell (ETM). Figure 6 [12] shows a breakdown of the debug and trace functionalities of the higher-end ARM Cortex-M Processors.

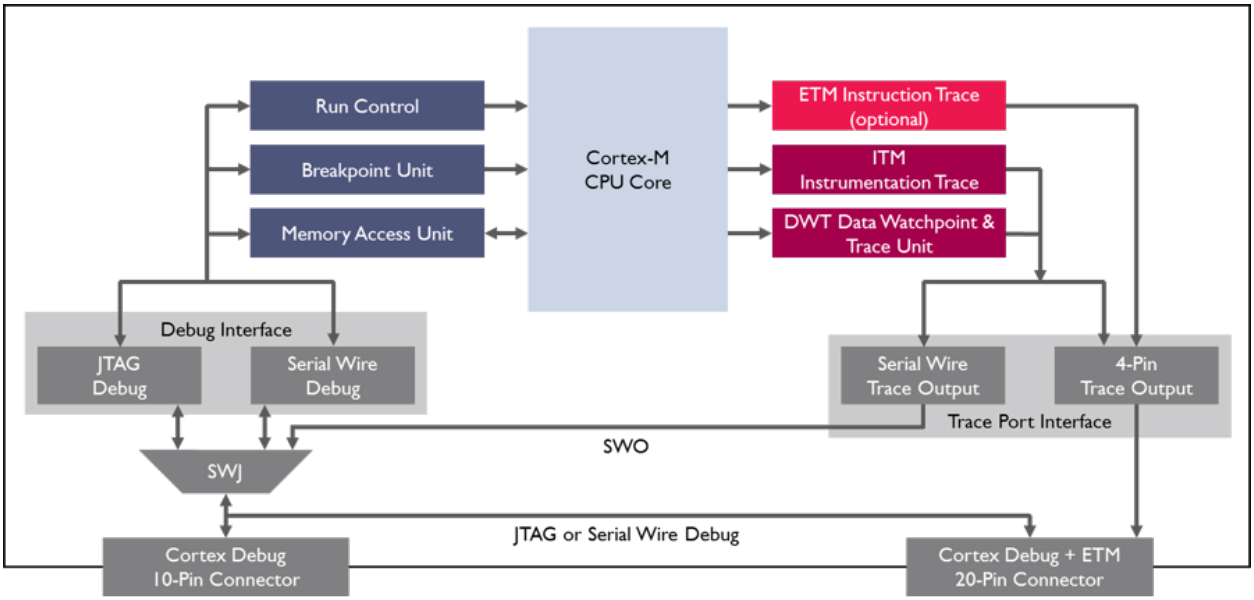

Figure 6 - ARM Cortex-M Debug Capabilities

The ITM is essentially a printf interface that adds instrumented code to the source to provide user-defined information on the trace. It can be used to check the value of variables, registers, memory addresses at various points throughout the runtime. Because it is user-defined, it can be easily modified to provide typical profiling data. ETM provides a more classical implementation of tracing. It provides the program counter and the machine code for all executed instructions (e.g. "`0x00000C32 STR R2, [R1]`"). As stated before, this level of detail provides us perfect insight into the runtime of the program, but the amount of data can be overwhelming. Due to the sheer volume of data, ETM can often drastically slow down the program and/or require reducing the clock frequency. Streaming the vast amount of data requires costly, special-purpose hardware like the Keil ULINKpro and Segger J-Trace and can often require an Ethernet

interface to the host PC to keep up with data rates. Tracing can be a great tool if that level of granularity is needed, but many microcontroller platforms do not have tracing capabilities.

3.4 Semihosting gprof

There is an existing solution that uses gprof in an embedded environment. This solution was developed by Erich Styger and his input helped provide direction for this thesis [13]. His solution offers gprof profiling for ARM Cortex-M processors by storing the profiling data locally. At the end of profiling, the processor compiles the data and writes gmon.out to the host PC. This file transfer is performed using semihosting and a Segger J-Link debug probes. This solution ports the Cygwin for i386 gprof to a GNU ARM architecture with some modifications to run better on an ARM Cortex-M architecture. The solution provides statistical, periodic program counter sampling in addition to a call graph arc counter. An overview of the system proposed in this solution can be seen in Figures 7 and 8.

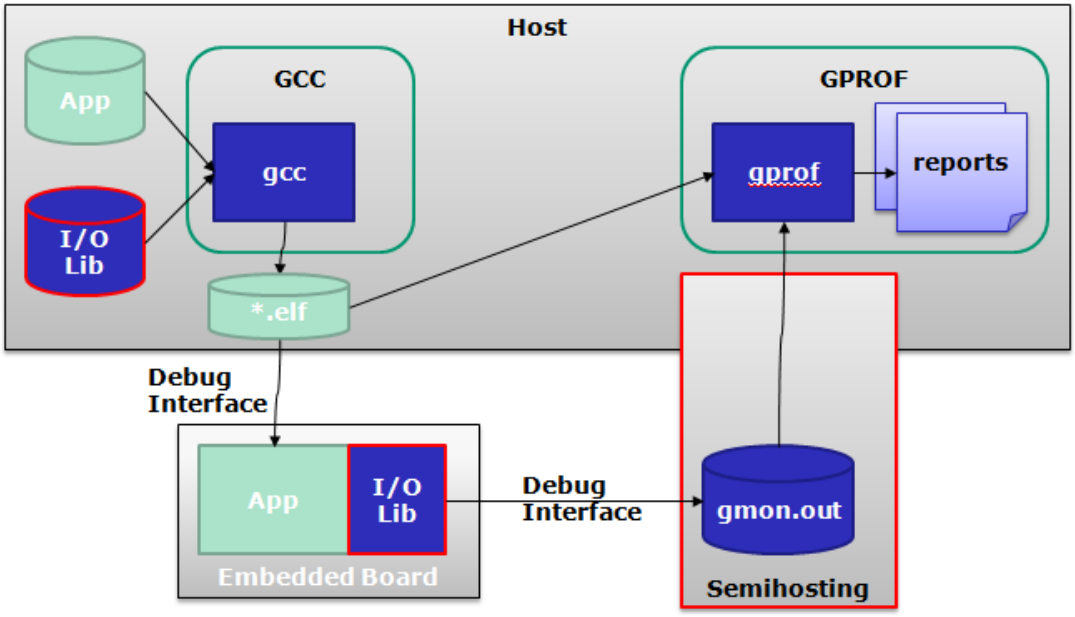

Figure 7 - gprof System Overview

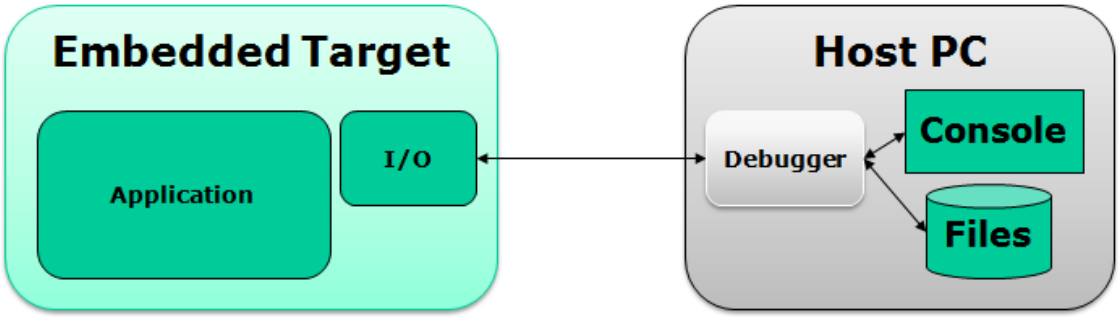

Figure 8 - Semihosting Overview

This solution is quite elegant and novel. It provides full gprof profiling functionality and insight into the embedded system. The CPU's performance with profiling is pretty near its normal runtime execution. However, due to the requirement for local storage on the embedded target, there is a much higher demand on RAM. There are three large data structures to store the profiling data. There is one array of counters for program counter samples and two array structure to track the callee/caller addresses along with the counter for the number of call arc occurrences. There is some memory address index hashing that occurs so there is not a counter value for each memory address; however, there is still a significant amount of RAM overhead to store these data structures. As shown in Section 4.4.4, a program utilizing 4 kB of ROM memory requires almost 9 kB of RAM for these data structures.

Another downside to this methodology is that it only works for processors that support semihosting. The semihosting functionality is required to provide a means to retrieve the output file that contains the profiling data from the target microcontroller. Semihosting requires additional ROM usage to provide the code for file I/O handling. Some microcontrollers have additional hardware functionality in the debugger to help provide this semihosting interface; however, most microcontrollers will require a rather costly external debugger like a Segger J-Link Debugger. If the program in question has available RAM and the necessary debugger

solutions, this methodology is rather effective. For instances with much tighter RAM space without this file transferring capability, this thesis provides an alternate solution with a much lower memory footprint.

3.5 Conclusion

To summarize the points made above and draw a more succinct comparison between the options for profiling microcontrollers, see the compiled data in Table I. This table makes for a much simpler comparison to decide on the best profiling tool for a specific application. Below is the definition of the comparison parameters in the columns of the table. They are rated as either being positive (✓), neutral (-), or negative (✗).

Definition of comparison parameters:

- **ROM Usage** – whether or not the profiler consumes significant ROM.
- **RAM Usage** – whether or not the profiler consumes significant RAM.
- **CPU Performance** – whether or not the profiler dominates execution time and reduces CPU performance.
- **Expressive Profile** – whether or not the profiler generates a detailed and expressive profile that avoids false inferences.
- **Indicative of Runtime Execution** – whether or not the profiler throttles clock speed or requires inherently inaccurate processor or peripheral simulations.
- **Ease of Use** – whether or not the profiler is easy to use and extract meaningful data from.
- **Cross-Platform Support** – whether or not the profiler can be adopted across platforms.

Table I - Profiling Solution Comparison

Solution	ROM Usage	RAM Usage	CPU Performance	Expressive Profile	Indicative of Runtime Execution	Ease of Use	Cross-Platform Support
Existing Profilers	×	×	×	✓	✓	✓	✓
Simulators	N/A	N/A	N/A	✓	×	×	×
Tracing	×	×	×	✓	-	-	-
Semihosting gprof	✓	×	✓	-	✓	-	-
Thesis - gprof	✓	✓	-	-	✓	-	✓

✓ Positive - Neutral × Negative

CHAPTER 4 - SYSTEM DESIGN

4.1 System Overview

gprof is a profiling and performance analysis tool originally developed for Unix systems. A few years later, another implementation was written for the GNU project and packaged as a part of the GNU binutils. It is a free and open-source solution that combines both instrumentation and statistical profiling methodology. This thesis converts gprof from the GNU C library or glibc to a more lightweight implementation intended for microcontrollers. The main idea is to divvy up profiling processing responsibilities between the target microcontroller (MCU) and a host personal computer (PC). This solution can be applied to any microcontroller that has a serial interface and whose source code can be compiled using GCC.

gprof has three different types of output it can generate. The first type is the flat profile which is generated using statistical profiling to sample the program counter. It creates a histogram showing execution time per function. The second type is the call graph profile which uses instrumented code to record the callee and caller addresses at the time of each function call. It counts the number of times each parent calls one of its children and constructs a call graph. It is also used to provide more details to the flat profile. Finally, there is the annotated source listing which uses basic-block information to label the execution count of each line of code. Instrumentation is added in at each branch instruction to count the number of times a basic block is executed. While the basic-block counting can be useful for detecting anomalies within a function, it is not often used because it can overburden the processor. Providing this level of detail would require too much overhead for this lightweight embedded solution. It would quickly exhaust memory, slow execution, and introduce profiling errors. Instruction counting the functionality is not included in this solution.

gprof profile generation happens in 3 steps: compile-time, runtime, and post-runtime. At compile-time, the program is compiled with the "-pg" compiler flag to let GCC know to add instrumentation to the object code to record the profiling data. During runtime, the program runs these lines of instrumented code and begins to collect the profiling data. At program exit, the internal profiling data structures are written to the gmon.out file per the standard. After runtime, the gprof program is used to construct the program's profile. The runtime program's executable file and the generated gmon.out file as provided as input. gprof uses this information to construct the flat profile and the call graph profile. The typical approach for gprof profiling is succinctly outlined below.

1. **Compile-time**: GCC adds instrumented code to the program to record profiling data.
2. **Runtime**: Program runs & records profiling data. Creates gmon.out file on program exit.
3. **Post-Runtime**: Input executable file and gmon.out to gprof. gprof generates profile.

The novel idea this thesis proposes is to divide the runtime step into two stages. The first stage is carried out on the target microcontroller while the second stage is carried out on the host PC. By implementing the profiling analysis in this way, it greatly reduces the processing and memory demand on the target microcontroller. This overhead reduction allows this approach to be more readily adopted across all microcontroller platforms. This solution includes lower-end chips which previously could not be profiled. The new division of labor in the runtime stage is captured below.

1. **Compile-time**: GCC adds instrumented code to the program to record profiling data.
2. **Runtime**: Program executes. Profiling data is captured.

a. **MCU**: Program executes. Instrumented instructions transmit profiling data via serial interface to the host PC.
b. **PC**: Application receives profiling data via serial interface throughout target MCU's program execution. Constructs gmon.out file upon program exit.

3. **Post-Runtime**: Input executable file and gmon.out to gprof. gprof generates profile.

Figure 9 clearly lays out the three distinct steps in the standard gprof implementation. Figure 10 contrasts this implementation to show the novel approach dividing the Runtime portion into the Target MCU Application and the Host PC application. In this approach, compile-time occurs on the host PC. Runtime program execution occurs on the target MCU while the host PC continually collects profiling data throughout the runtime duration. Finally, the host PC constructs the gmon.out file upon program exit and runs this data into gprof. gprof generates the program's profile.

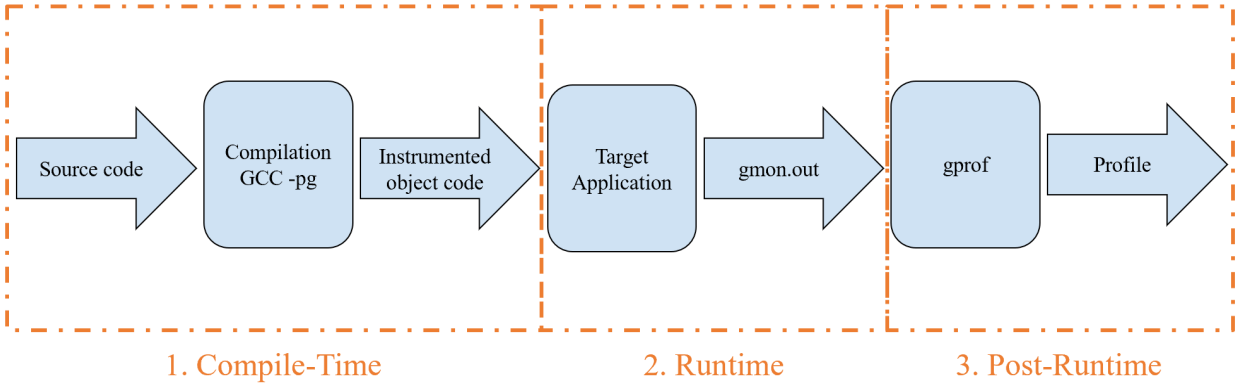

Figure 9 - Standard gprof Profiling Process

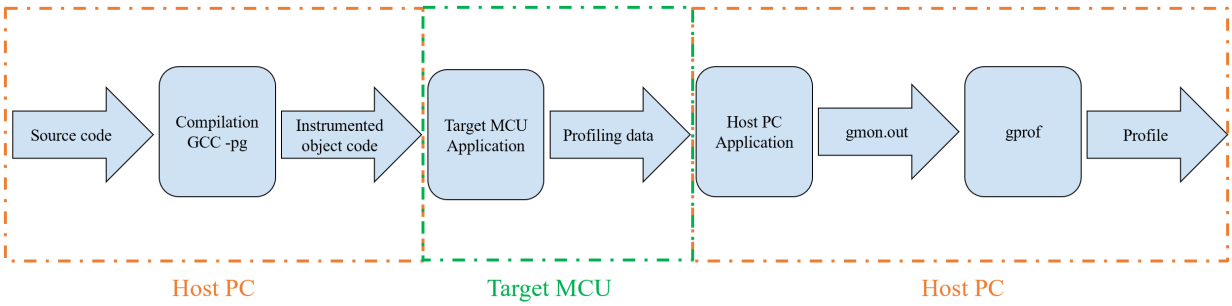

Figure 10 - New gprof Profiling Process

As previously mentioned, a serial interface is required to facilitate the transfer of data between the microcontroller and the PC. The type of serial interface does not matter. It can be whatever is suitable or most available for the specific application whether that be UART, SPI, I2C, or some proprietary debug console port or trace. However, it is important to select a serial communication method with a relatively high data throughput. If the throughput is too low, it can hang the program's execution while it waits for the serial data transmission. The profiler needs only one-way communication. A USB to serial converter such as an FT232 chip may be required to provide the connection between the PC and microcontroller. Figure 11 shows an example of the required connection to implement this solution.

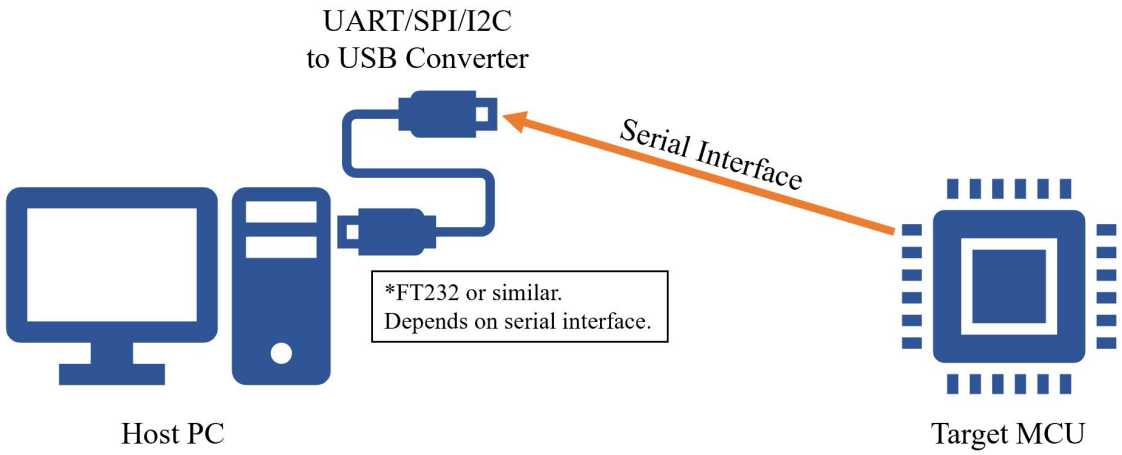

Figure 11 - Profiling Physical Interfaces

4.2 When & What to Profile

It might not always be a good idea to profile a program from startup to program exit. During startup, there might be a lot of peripheral initialization functions that dominate execution. If you want to profile the program to find optimization opportunities in the main code, those initialization functions may skew the profiling data and cover up some other areas of interest. On the other hand, you may only want to profile the initialization sequence to see what parts of the code could be optimized to bring up the system faster. There are many different ways in which this profiling methodology could be used. Because of this, it is important to be able to start and stop profiling at will. Initialization and deinitialization functions are provided in order to only capture the portion of execution that is of interest. It is important to note that the compiler directed calls to `__gnu_mcount_nc()` cannot be removed. However, a static variable can be used to prevent any additional unnecessary instructions from executing to reduce latency when profiling is stopped.

It is also important to note the profiler initialization itself has a number of steps and can take some time to execute. The serial port, the periodic timer interrupt for program counter sampling, and the queue structure must all be initialized before profiling can begin. These initialization functions also depend on startup and clock configuration code for the processor. It is important to be cognizant of these dependencies and structure the system's initialization in a way that makes sense for that application. Profiling before the correct peripherals are initialized can result in a hard fault.

It is important to deliberately select which files and functions are being profiled. There are specific files that should always be excluded from profiling. An entire project cannot be compiled with “-pg” flag. Instead, the project must be compiled on a per-file basis to include

certain source files with the “-pg” flag and exclude other source files by omitting the “-pg” flag. Below is a list of files that cannot be profiled. It summarizes all of the various instructions and functions that should be excluded from profiling. The list is not an exhaustive one, but it should serve as a guide to finding functions and files in other projects that cannot be profiled. Section 7.4 discusses more troubleshooting tips for determining which files should be profiled.

Files to Exclude from Profiling (not an exhaustive list)

- Profiler – Profiling the profiler source files will trigger endless recursion and will cause the stack to overflow. This includes the mcount.S, gmon_arc.c, gmon_profil.c, gmon_queue.c, and gmon_serial.c.
- Processor Startup – Profiling processor startup and clock configuration code will cause a hard fault in most systems.
- Critical Sections – Profiling functions that contain critical sections of code will cause a hard fault. Most processors have macros and/or functions to protect critical sections to prevent resource conflicts that can cause errors. Serial interfaces, buffers, file systems, etc. are common examples of resources that require critical section protection.
- Interrupts – Depending upon how the processor handles interrupts, profiling interrupts may not be possible. If the processor is hitting a hard fault upon interrupt, excluding interrupt functions from arc profiling may solve this problem.
- RTOS Library – Most files that are part of an RTOS library can cause issues when profiled. The ways in which the context switching is handled could potentially cause a hard fault or some other unintentional error. Avoiding RTOS files can save a lot of headaches. Profiling RTOS files is typically not helpful since they most likely come from a standard library and are pretty well optimized. Most RTOS are comprised of many

hook and stub functions that take very little time to execute and profiling them would not provide much insight. These functions would only pollute the program's profile.

4.3 Target Microcontroller Application

4.3.1 Overview

The target microcontroller application is responsible for collecting profiling data and transmitting that data to the host PC via a serial interface. As mentioned earlier, this gprof implementation will be generating a flat profile and a call graph profile. Thus, instrumented code will need to be inserted to periodically sampling the program counter and to collect caller/callee function relationships. The program needs a framework for communicating the profiling data. This framework involves configuring a serial interface and establishing a system for enqueueing the data to be transmitted. The example application is implemented for an ARM Cortex-M processor. All of the files and functions involved in implementing the profiling application on the target microcontroller are summarized below. Figure 12 shows the hierarchy and relationships between these functions. The source code for this application is provided in the GitHub repository discussed in Appendix A.

Files and Functions for Target Microcontroller Application

```
mcount.S – handles low-level assembly call arc retrieval
  |--> __gnu_mcount_nc() – stub to retrieve callee/caller addresses
gmon_arc.c – handles call arc transmission
  |--> init_gprof() – initializes/starts profiler
  |--> deinit_gprof() – deinitializes/stops profiler
  |--> _mcount_internal() – call arc serial transmission
gmon_profil.c – handles periodic program counter sampling
  |--> init_Timer() – initializes periodic timer and interrupt
  |--> Timer_ISR() – timer overflow interrupt service routine (ISR)
gmon_profil.h – header file for periodic program counter sampling
```

```
gmon_queue.c – handles queueing for serial buffer
  |--> TxQ_Init() – initializes transmit queue
  |--> TxQ_Size() – determines current size of queue
  |--> TxQ_Enqueue() – enqueues a byte of data
  |--> TxQ_Dequeue() – dequeues a byte of data
gmon_queue.h – header file for queueing
gmon_serial.c – handles serial interface for profiling data
  |--> init_Serial() – initializes serial port and interrupt
  |--> Serial_ISR() – transmit data register empty ISR
gmon_serial.h – header file for serial interface
```

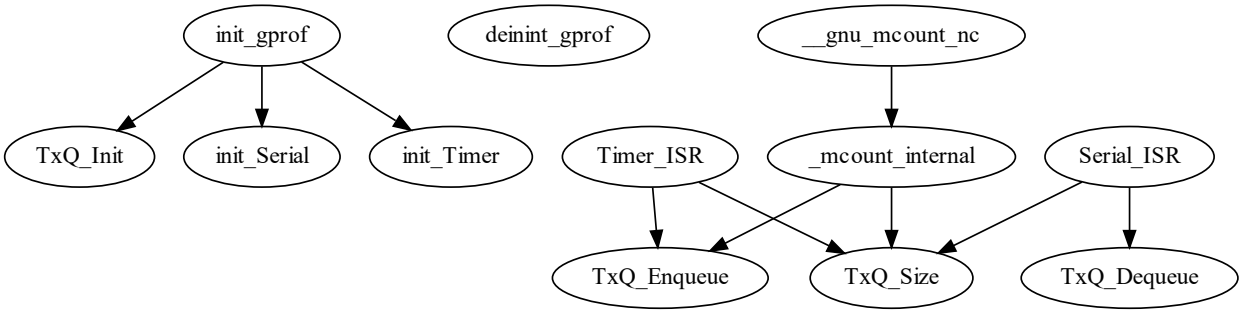

Figure 12 - Target Profiling Call Graph

4.3.2 Call Arcs – mcount.S & gmon_arc.c

Call arcs show the relationship between caller functions and callee functions. They are used to help track the number of times a specific function calls another specific function. Call arcs coupled with the periodic program counter samples create a holistic view of how the program executed and analyze where the processor spent its time. The caller and callee memory addresses are retrieved off the stack by a hook function. The hook function intercepts every function call, records the "from" or caller address and the "to" or callee address, and then directs the processor back to the intended callee function.

In order to use this hooking method, the source code must be compiled with GCC and the "-pg" compiler flag. The "-pg" flag tells the processor to call `__gnu_mcount_nc()` before

executing the function that was invoked. To more easily access the stack and successfully push and pop values, `__gnu_mcount_nc()` is implemented in assembly. In GCC 4.4 and newer, there is an initial link register (lr) push performed before the call to `__gnu_mcount_nc()`. After the call to `__gnu_mcount_nc()`, the stack appears as shown in Figure 13. After the initial link register push, The calling procedure for `__gnu_mcount_nc()` follows the standards of the AAPCS (Procedure Call Standard for the ARM Architecture) [14].

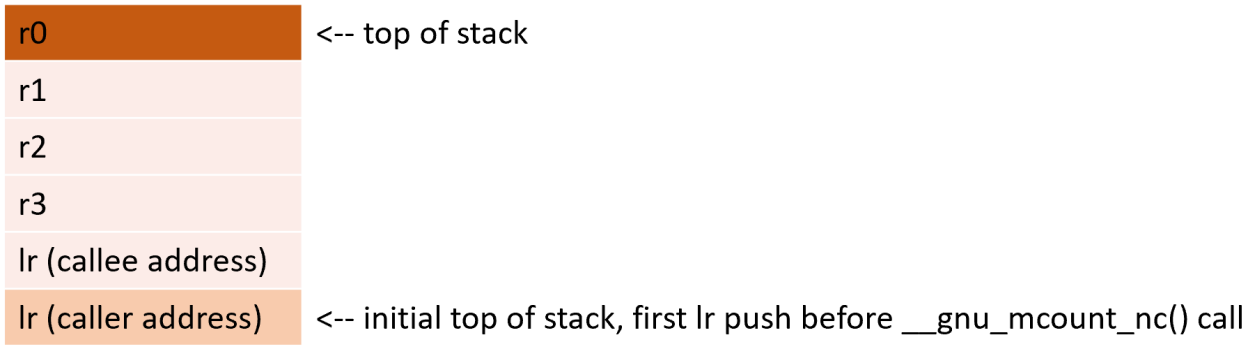

Figure 13 - __gnu_mcount_nc Call Stack

`__gnu_mcount_nc()` has to retrieve the caller and callee address, pass them to `_mcount_internal()`, and branch to the caller address. Once the program has entered `__gnu_mcount_nc()`, the link register contains the callee address. The caller address is stored in the old link register which is an offset of 4 away from the current link register and an offset of 20 away from the stack pointer. The callee address is moved to r1 and the caller address is moved to r0. r0 and r1 are passed as arguments to `_mcount_internal()`. The caller address is moved to lr to provide a return address for the callee function. The callee address is moved to ip, the intra-procedure-call scratch register. After the call to `_mcount_internal()`, the program branches to callee function using the ip register. The code for `__gnu_mcount_nc()` is shown below in Figure 14.

```
.globl __gnu_mcount_nc
.type __gnu_mcount_nc, %function

__gnu_mcount_nc:
    /* save registers */
    push {r0, r1, r2, r3, lr}

    /* r1 contains callee address */
    mov r1, lr
    /* move callee address to ip */
    mov ip, r1

    /* r0 contains caller address */
    ldr r0, [sp, #20]
    /* moves caller address to lr */
    mov lr, r0

    /* jump to _mcount_internal() implementation */
    bl _mcount_internal

    /* restore saved registers */
    pop {r0, r1, r2, r3}

    /* pops pc/lr and ip */
    add sp, sp, #8

    /* branch to callee */
    bx ip
```

Figure 14 - __gnu_mcount_nc Code

`_mcount_internal()` is responsible for packaging the caller or "from" address and the callee or "to" address for the call arc and transmitting the information to the host PC. To distinguish between a program counter's memory address and a call arc's memory address on the receiving end, the microcontroller transmits an ASCII 'A' (0x41) for a call arc sample and an ASCII 'P' (0x50) for a program counter sample. The total data package per call arc is 7 bytes. There is 1 byte for the 'A', 3 bytes for the "from" address, and 3 bytes for the "to" address. The data is assembled and enqueued to the transmit ring buffer to be sent to the serial port. Figure 15 shows an example data packet where the address of the caller function is 0x014AF4 and the address of the callee function is 0x11C4B2.

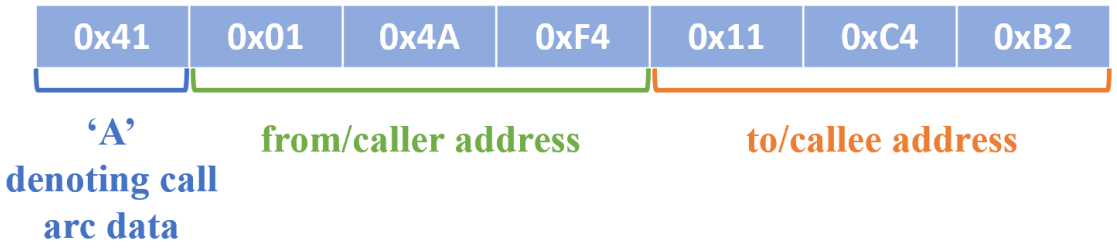

Figure 15 - Call Arc Data Packet

The "from" and "to" addresses are passed as arguments into `_mcount_internal()`, so there is no processing of the memory addresses to be done. The only work performed in `_mcount_internal()` is the packaging and transmission of the data. This procedure will vary across environments based on the queue structure and serial interface. Memory address size can also play a role. This processor has a 32-bit memory address; however, the memory addresses do not range from 0x000000 to 0xFFFFFFFF. The microcontroller used in this example only has 128 kB of ROM. The leading zeros can be stripped away, and the resulting address will fit in a 3-byte wide packet.

Figure 16 shows the example implementation for an ARM Cortex-M processor using UART with a ring buffer. One of the important commonalities that must be maintained in adopting this profiling solution for other environments is preventing the call arc data packet enqueueing and the program counter data packet enqueueing from interrupting each other. If this happens, the data will quickly become corrupted. This implementation solves this problem by disabling the periodic timer interrupt for program counter sampling for the duration of the call arc data packet enqueueing. Once the interrupt is disabled, the data is enqueued in an 8-bit wide array byte by byte. There are checks to confirm the queue is not full and the data was successfully enqueued before advancing to the next byte of data. After the entire call arc has been enqueued, `_mcount_internal()` returns to `__gnu_mcount_nc()` which immediately branches to the callee function.

```
void _mcount_internal(uint32_t *frompcindex, uint32_t *selfpc) {

  // if gprof has been initialized
  if (g_Initialized) {

    // copy from and to address to serial buffer
    uint8_t bp[7];
    bp[0] = 'A';
    bp[1] = (uint8_t)(((uint32_t)frompcindex & 0xFF0000)>>16);
    bp[2] = (uint8_t)(((uint32_t)frompcindex & 0x00FF00)>>8);
    bp[3] = (uint8_t)((uint32_t)frompcindex & 0x0000FF);
    bp[4] = (uint8_t)(((uint32_t)selfpc&0xFF0000)>>16);
    bp[5] = (uint8_t)(((uint32_t)selfpc&0x00FF00)>>8);
    bp[6] = (uint8_t)((uint32_t)selfpc&0x0000FF);

    // disable interrupt to prevent data pollution
    PIT_PDD_DisableInterrupt(PIT_BASE_PTR, PIT_PDD_CHANNEL_1);
    // enqueue local serial buffer
    for (int i=0; i<7; i++) {
      // copy characters up to null terminator
      while (TxQ_Size() == MAX_Q_SIZE)
        ; // wait for space to open up
      while(!TxQ_Enqueue(bp[i]));
    }
    //re-enable interrupt
    PIT_PDD_EnableInterrupt(PIT_BASE_PTR, PIT_PDD_CHANNEL_1);
  }
}
```

Figure 16 - _mcount_internal() Code

4.3.3 Program Counter Samples – gmon_profil.c

Unlike call arc counting, there is no automatic instrumentation for program counter sampling. Instead, a timer peripheral must be used to provide the precise timing for program counter sampling. The timer interrupts the program at a specified frequency to record the program counter value, package the data packet, and enqueue the data for transmission to the host PC. In the example application, a peripheral called the periodic interrupt timer (PIT) is used. The timer counts up to a specified value before overflowing and causing an interrupt. The timer is initialized with an overflow value which is calculated in gmon_profil.h as shown in Figure 17. SAMPLE_FREQ_HZ_TO_TICKS computes the number of timer ticks required to achieve the target profiler sampling frequency based on the system clock frequency.

```
/* profiling frequency. */
#define PROF_HZ         1000
#define SystemCoreClock (48e6)
#define SAMPLE_FREQ_HZ_TO_TICKS ((SystemCoreClock/(2*PROF_HZ))-1)
```

Figure 17 - Profiler Frequency Setting

A number of different methods can be used to provide the periodic control required for precise profiling including a hardware timer interrupt, an RTOS task, or another form of preemptive scheduler. Regardless of the method, the central aim is to avoid deviating from the specified frequency. gprof assumes a constant, precise period. Slight inaccuracies in the sampling period can result in the profiler providing inaccurate analysis or drawing false conclusions. To maintain this exact timing, the interrupt or task must be given a higher priority than all other interrupts or tasks. This priority arrangement ensures the profiler will not have to compete for the processor to begin execution. The function will never be blocked or interrupted. The example programs in Appendix A show one way to establish periodicity for program counter sampling.

Retrieving the program counter seems like a trivial task, but there are some subtle nuances that make it difficult. The methodology will change based on the system and statistical profiling technique used. When the interrupt is triggered, the program counter is stored on the stack to be used as the return address. During the context switch to the interrupt or task for the profiler, other data is also pushed onto the stack. In some cases, the manufacturer may provide a built-in function for retrieving the value off of the stack. These example programs do not have a built-in function. Instead, the stack pointer is used with an offset to retrieve the correct value. There is a macro function defined to retrieve the stack pointer in the standard system source files.

In this example, an ARM Cortex-M0+ processor is used. The Cortex-M0+ has two stack pointers: the main stack pointer (MSP) and the process stack pointer (PSP). The MSP is used

initially and in handler mode during interrupts. The PSP is used in thread mode, typically for RTOS applications. When the timer overflow interrupt occurs, the MSP is being used. However, the program counter value could have been pushed onto either the MSP or the PSP based on the specific program. The profiler must understand whether the processor was using an RTOS or not to retrieve the program counter from the correct stack. Figure 18 shows how this example handles it. The user defines a macro at compile time to tell the processor whether the program uses an RTOS.

```
// Comment out USING_RTOS definition if not using RTOS
// #define USING_RTOS
#define HW_RET_ADX_OFFSET (24)
#define IRQ_FRAME_SIZE (8)
#define PC_OFFSET (24)

#ifdef USING_RTOS // Don't need these since PC is on PSP, not MSP
  #define FRAME_SIZE  (0)
  #define CUR_SP      (__get_PSP())
#else // Using MSP, so stack frames are also on the MSP stack
  #define FRAME_SIZE  (IRQ_FRAME_SIZE + PC_OFFSET)
  #define CUR_SP      (__get_MSP())
#endif
```

Figure 18 - Stack Pointer Selection

Once the profiler knows which stack pointer is being used, it needs to understand the offset from the stack pointer to the memory address storing the needed program counter. This offset will vary across processors. The system's call standard [14] must be referenced to compute the value. Figure 18 shows the definitions to calculate the offset for both the MSP and PSP implementations. Figure 19 shows how these values are used in the interrupt to retrieve the value. There is an additional issue that could arise for an RTOS program. The Cortex-M0+ processor initially uses the MSP. Once the RTOS is started, the PSP is utilized. If the statistical profiler tries to sample the program counter before the RTOS is started, it will try to reference

the PSP before it has a legal value. The processor will likely attempt to access an out-of-bounds memory location and cause a hard fault. This case can be handled by detecting and handling an illegal memory address. It can also be avoided by prohibiting PSP access until the RTOS has begun.

Once the stack pointer and the offset have successfully been attained, the program counter value can easily be acquired. Figure 19 shows the code for the timer's interrupt service routine for reference. Since the timer initialization is largely processor-specific, that function is only shown in Appendix A. The macros in gmon_profil.h handle most of the heavy lifting to compute the offset and retrieve the stack pointer memory addresses required. The ISR is responsible for retrieving the program counter based on those values and enqueuing the data packet to the buffer just like in `_mcount_internal()`. The program counter data packet is packaged up into 4 bytes. As mentioned previously, it uses a 'P' (0x50) to denote that the next 3 bytes will be a program counter's memory address. Figure 20 shows an example program counter data packet. The size of this data packet can change based on the valid memory address range. This example application has a small memory address range, so the program counter can easily be compressed to 3 bytes. This data packet is then enqueued and transmitted as will be discussed in Section 4.3.4.

```
/* PIT interrupt service routine to sample the program counter */
Timer_ISR(Cpu_ivINT_PIT) {
  static uint32_t pc, sp;

  // retrieve stack pointer
  // add additional frames on stack to retrieve pc
  sp = (CUR_SP + FRAME_SIZE + HW_RET_ADX_OFFSET);
  // retrieve program counter
  pc = *(uint32_t*)(sp);

  uint8_t bp[4];
  bp[0] = 'P';
  bp[1] = (uint8_t)((pc&0x00FF0000)>>16);
  bp[2] = (uint8_t)((pc&0x0000FF00)>>8);
  bp[3] = (uint8_t)(pc&0x000000FF);

  // enqueue data packet one byte at a time
  for (int i=0; i<4; i++) {
    // wait for space to open up
    while (TxQ_Size() == MAX_Q_SIZE);
    // enqueue next byte
    while (!TxQ_Enqueue(bp[i]));
  }
}
```

Figure 19 - PC Sampling Interrupt

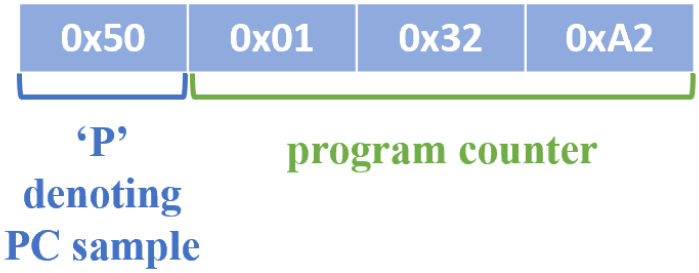

Figure 20 - Program Counter Data Packet

4.3.4 Transmit Queue – gmon_queue.c

For safe and efficient access to profiling data, a circular ring buffer is used. The buffer is statically defined in gmon_queue.c to protect from data corruption. In addition, there are static variables defined to track the head, tail, and size of the buffer queue. These definitions are shown in Figure 21. `MAX_Q_SIZE` is defined externally in the header file. The size of the buffer can be grown or shrunk based on the specific application. If the serial interface selected has a much higher data throughput than the volume of profiling data collected, the buffer can be shrunk to

reduce RAM consumption. Utility functions for initializing or clearing the queue and retrieving the size of the queue are shown in Figure 22 and Figure 23 respectively.

```
static uint8_t TxQ[MAX_Q_SIZE]; /* ring queue data buffer */
static uint16_t TxQHead; /* queue head index */
static uint16_t TxQTail; /* queue tail index */
static uint16_t TxQSize; /* queue size data */
```

Figure 21 - Static Queue Variable Declarations

```
void TxQ_Init() {
  TxQHead = 0;
  TxQTail = 0;
  TxQSize = 0;
}
```

Figure 22 - Queue Initialization Function

```
uint16_t TxQ_Size() {
  return TxQSize;
}
```

Figure 23 - Queue Size Function

Enqueuing and dequeuing data from the ring buffer are considered critical sections of code. Each function accesses and modifies the ring buffer. Without protecting these regions of code, they could interrupt each other and simultaneously access the buffer causing unwanted behavior. Before accessing the ring buffer, the program stores the current interrupt masking state and disables interrupts. After the ring buffer has been successfully modified, the saved interrupt masking state is restored. This critical section protection is shown in the `TxQ_Enqueue()` and `TxQ_Dequeue()` in Figures 24 and 25.

```
uint8_t TxQ_Enqueue(uint8_t data) {

  // if queue is full, return error
  if (TxQSize==MAX_Q_SIZE) {
    return 0; // failure
  }
  // else we can enqueue new element
  else {
    // save current masking state
    // disable interrupts
    EnterCritical();
    // store data, increment tail
    TxQ[TxQTail++] = data;
    TxQTail %= MAX_Q_SIZE;
    // increment size
    TxQSize++;
    // restore interrupt masking state
    ExitCritical();
    return 1; // success
  }
}
```

Figure 24 - Enqueue Function

```
uint8_t TxQ_Dequeue() {
  uint8_t temp = 0;

  // don't dequeue if queue is empty
  if (TxQSize) {
    // save current masking state
    // disable interrupts
    EnterCritical();
    // remove data
    temp = TxQ[TxQHead];
    // increment head and decrement size
    // unused entries for debugging
    TxQ[TxQHead++] = '_';
    TxQHead %= MAX_Q_SIZE;
    TxQSize--;
    // restore interrupt masking state
    ExitCritical();
  }
  return temp;
}
```

Figure 25 - Dequeue Function

4.3.5 Serial Interface – gmon_serial.c

The last portion of the target microcontroller application is the serial interface. The serial interface is used to transmit profiling data from the microcontroller to the host PC during runtime. The type of serial interface does not matter. It can be whatever is most available or most suitable for the specific application. The only design consideration that requires deliberation is the data throughput. The serial port must keep up with the volume of data being generated by the profiler. The volume of data can vary drastically across programs. Some programs may call many different functions in a short amount of time generating a large amount of call arc data traffic. Other programs can call very few functions that take a long time to execute producing very low traffic volumes. For reference, the example applications discussed in this thesis utilize a UART serial interface running at 1.5 Megabaud.

Serial interfaces employ either an interrupt or polling approach. Since the communication in this application is one-way, it is only worth discussing data transmission. In a polling approach, the transmit data register empty bit of the serial port is polled. The byte of data is only moved to the data register once the data register is empty and the bit is set high. In an interrupt approach, the byte of data is enqueued into a buffer structure in memory. An interrupt is triggered once the data register is empty. The interrupt service routine then dequeues the next byte of data and moves it to the transmit data register. The serial interrupt method is the most rugged approach for profiling. Unlike the polling method, there is no fear execution might stall. Polling could potentially cause the processor to crash if it blocks while in an interrupt service routine. Using the interrupt methodology has the added benefit of evenly dividing up data transmission over the program's execution. By more evenly distributing the serial processing, the program has a lower chance of the serial buffer filling up.

Even with the benefits of implementing a serial interface with an interrupt, the serial data traffic could still be too great. If the volume of data is too great, the buffer could fill up and cause the processor to halt. If this happens, the serial interface is likely bottlenecking the data throughput. A good way to test if this is the case is to reduce the amount of data. This can easily be done by lowering the program counter sampling frequency. If the new sampling frequency fixes the issue, the profiler data is overwhelming the serial interface. Some troubleshooting tips for diagnosing and fixing this issue in are provided in Section 7.4. Section 8.2.2 discusses some future areas of work to further reduce serial traffic.

4.4 Host PC Application

4.4.1 Overview

In this thesis, the typical runtime gprof profiling is divided between the target microcontroller and the host PC. The host PC profiling application picks up where the target microcontroller application ended. The target microcontroller captured the call arc and program counter samples and transmitted these to the host PC application. The host PC application is now responsible for interpreting these messages, storing them in temporary data structures, and generating the gmon.out file upon. Once the host PC application is finished executing, gprof can run with the generated gmon.out file and the executable file from the microcontroller to construct the profile. All files and functions involved in implementing the profiling application on the host PC are summarized below. The call graph shown in Figure 25 illustrates the relationships between these functions. The source code for this application is provided in the GitHub repository discussed in Appendix A.

Files and Functions for Target Microcontroller Application

main.c – handles main workflow and parsing of the input data
  |--> main() – main workflow
  |--> hexconverter() – converts input data to correct format
mcount.c – handles call arc processing and output file generation
  |--> _mcount_internal() – processes call arc samples
gmon.c – handles call arc processing and output file generation
  |--> moncontrol() – start/stop control of profiling
  |--> monstartup() – initialization of profiling
  |--> _mcleanup() – cleanup function for profiler
  |--> write_gmon() – writes profiling data out to gmon.out
  |--> write_hist() – writes program counter sample histogram
  |--> write_call_graph() – writes call graph arc samples
gmon.h – header file for call arc profiler
gmon_out.h – header file specifying output format
profil.c – handles program counter sample processing
  |--> profil() – start/stop/configure statistical profiling
  |--> profil_count() – processing statistical profiling samples
profil.h – header file for program counter sample processing

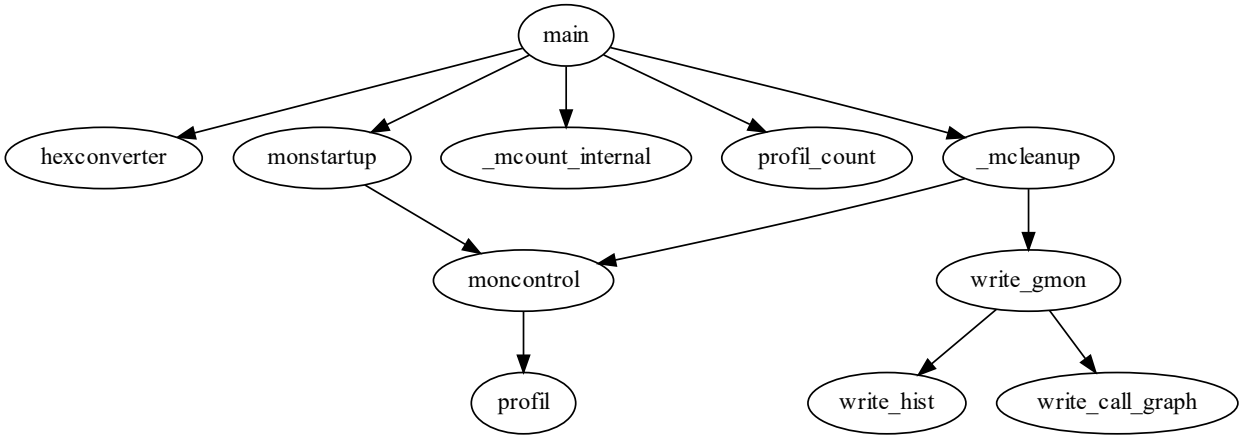

Figure 26 - Host PC Application Call Graph

4.4.2 Serial Receiver

To successfully implement this two-stage profiling approach, an appropriate serial receiver must be established. The serial receiver can take many different forms based on the

serial protocol used and the host PC's capabilities. The data can be transmitted using UART, SPI, I2C, USB, or any other type of protocol. Based on the selected protocol, the host PC may require some intermediary hardware. This hardware can come in the form of a serial converter like an FTDI FT232 chip or a logic analyzer like an Analog Discovery 2. The serial converter or logic analyzer will require some software for data capture. Some common serial monitor solutions are Tera Term, Putty, and Realterm.

The example outlined in this thesis uses the microcontroller's UART port. An FTDI FT232R chip is used to convert from UART to USB. Realterm serial monitor was employed to capture the profiling data and write it to a text file. Some issues can arise when the serial monitor is interpreting the profiling data. The profiling data is packaged into two different packets for call arcs and program counter samples shown in Figure 27. Most serial monitors will interpret 8-bit chunks of the memory address as a hex representation of an ASCII character. Some serial monitors may even try to interpret the parts of the memory addresses as a UTF-8 or UTF-16 character. Some of these characters can be interpreted as special control characters. These interpretations will force the serial monitor to take some action instead of recording the value. The missing values will result in a partial profile from assembled from corrupt data.

Figure 27 - Profiling Data Packets

There are several different ways to prevent the serial monitor from distorting the profiling data. There is a setting in most serial monitors to force them to interpret the data 8 bits at a time and write those 8 bits to file regardless of their interpreted ASCII or Unicode control meaning. If

the serial monitor does not have this capability, there is another setting in most serial monitors to force the data to be captured as hex. This setting will cause each nibble of the hexadecimal memory address to be translated into its ASCII value. This value is then written to the file with its 8-bit representation. For example, 'A' is used to denote a call arc and is transmitted over the serial port using its 8-bit hex representation 0x41 (or binary 0100 0001). If this value were written to the file with the "captured as hex "setting turned on, the '4' and the '1' would be interpreted separately as 0x34 and 0x31 respectively.

Figures 28 and 29 show a conversion from the original memory address to this new interpretation for both a call arc sample and a program counter sample. This process may seem convoluted, but it ensures that no data is misconstrued. Once all the data is written to a text file by the serial monitor, the text file is passed to the host PC application for processing.

Figure 28 - Original Hex Representation

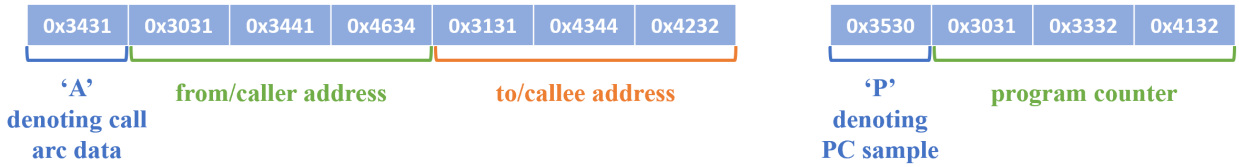

Figure 29 - Converted Hex to ASCII Hex Representation

4.4.3 Main While Loop – main.c

The main.c file contains the `main()` and `hexconverter()` functions. The `main()` function is responsible for all of the I/O file control. It is also responsible for handling the control flow of profiler data processing. It begins by initializing the profiler data structures by calling

`monstartup()` and finishes by writing those structures out to a file by calling `_mcleanup()`. The `main()` function parses the input file, processes each profiling sample individually, and writes to the gmon.out output file. The input file is a text file that was generated by the serial console program in Section 4.4.2. As discussed in that section, the data can be provided in two formats based on the capabilities of the serial console program. If the serial console program could not write the control characters and had to convert each hexadecimal nibble to its ASCII representation, the hex conversion implementation must be used. These two different parsing algorithms are selected by a simple macro in profil.h. If HEX_CONVERT is defined, the program is looking for data in the format shown in Figure 28. If not, it is looking for standard ASCII data as shown in Figure 29.

The `hexconverter()` function implements this translation. It uses a simple switch statement to convert from the ASCII representations of each hex digit of the memory address to their numerical values. The memory address in this example is three bytes long, but the serial console breaks up each nibble of the memory address into its ASCII-represented byte. When it gets to the host PC application, the memory address is now six bytes. The application reads in each byte, uses the hex converter program to reinterpret it as a nibble, and ORs it together to reconstruct the full memory address of the program counter. Now, the memory address can be passed on to `profil_count()` for sample counting. Figure 30 shows this processing. The same process is followed for interpreting the “from” and “to” memory addresses for the call graph arcs.

```
// P = 0x50
// if the data is a PC sample
if (type[0] == '5' && type[1] == '0') {
  uint8_t pcArray[MEM_SIZEX];
  uint32_t pc = 0;

  // capture pc address
  for (int j=0; j<MEM_SIZEX; j++) {
    fscanf(fp, "%c", &pcArray[j]);
  }

  // convert numbers to a single number
  for (int j=0; j<MEM_SIZEX; j++) {
    pcArray[j] = hexconverter(pcArray[j]);
    pc |= (pcArray[j] << (MEM_SIZEX-1-j)*4);
  }

  // program counter sampling processing
  profil_count(pc);
}
```

Figure 30 - Memory Address Hex Converter

As mentioned in Sections 4.2.2 and 4.2.3, a ‘P’ (0x50) is transmitted to signal that the following bytes represent a program counter and an ‘A’ (0x41) is sent to signal a call graph arc. If the hex conversion is required, the ‘P’ is represented as ‘5’ (0x35) and ‘0’ (0x30). The ‘A’ is represented as ‘4’ (0x34) and ‘1’ (0x31). The main while loop looks for these “type” characters to determine what sort of processing must be done on the following bytes. If the program detects an ‘A’, it knows will receive two memory addresses: the “from” address followed by the “to” address. If the program detects a ‘P’, it knows it will receive a memory address corresponding to a program counter. Based on the type of sample, it will call the corresponding function to process the data. The main while loop continues processing the data until it has reached the end of the file. At this point, the profiling data is moved from the various structs in the program’s memory space to the gmon.out file. Once the file has been successfully created, the program exits.

4.4.4 gmon Handling – gmon.c

The gmon.c file contains the following functions: `monstartup()`, `moncontrol()`, `_mcleanup()`, `write_gmon()`, `write_hist()`, and `write_call_graph()`. These functions are responsible for setting up the profiling data structures, starting profiling, writing the gmon.out file, and cleaning up the data structures. This file was adopted from the GNU C Library (glibc) version 2.30 and modified for use in this application.

Before discussing the purposes of these functions, it is important to understand the data structures that are storing the profiling data. The three key data structures that are used are `froms[]`, `tos[]`, and `kcount[]`. These data structures are bundled into one large struct called `_gmonparam`. The `froms[]` array represents the memory address of the caller functions. To avoid a direct mapping of the target microcontroller program's memory to this data structure, a divider called `HASHFRACTION` is used. `HASHFRACTION` makes assumptions about how far apart consecutive call instructions can be in memory. If the `HASHFRACTION` is too large, data can be lost. `HASHFRACTION` is defined as 2 which is a rather safe bet and reduces the required RAM on the host PC. This value means our `froms[]` array length is the total program ROM or text size divided by 2. Due to the nature of this approach, RAM size should not cause issues since the host PC's RAM size is comparatively infinite compared to the target microcontroller's ROM size.

The `froms[]` array contains a list of index values into the `tos[]` array modeling the call arc. The `tos[]` array contains the rest of the information needed to complete the call graph. Each entry contains the `selfpc` or callee address, the counter for that particular call arc, and a link to the next entry in the table. Similarly to the `froms[]` array, the `tos[]` array has a macro called `ARCDENSITY` to determine how large the array should be as compared to the microcontroller program's memory. `ARCDENSITY` defines a percentage of text space that should

be allocated for the `tos[]` array. The default value is 3, which should be more than sufficient for most cases. However, the program could still fail if the microcontroller program being profiled contains a rather large number of very tiny functions. In this case, the `ARCDENSITY` can be increased. Again, this increase should have no real ramifications on the host PC program as its memory size far outstrips the microcontroller's memory size. Finally, `kcount[]` is an array of counters incremented for every program counter occurrence. Similar to `froms[]`, `kcount[]` has a divider called `HISTFRACTION` to conserve memory by avoiding a direct one-to-one mapping.

These parameters can be easily dialed in for the specific applications by modifying the corresponding macros defined in gmon.h. These default settings should be appropriate for most if not all cases. In reviewing how the allocation of data structures for profiling using gprof is handled, it becomes abundantly clear why this lightweight solution is necessary. For a typical application, `froms[]` requires 2 bytes per entry, `tos[]` requires 12 bytes per entry, and `kcount[]` requires another 2 bytes for memory. Even if a value of 2 is used for `HASHFRACTION` and `HISTFRACTION` and a more conservative percentage value of 2 is used for the `ARCDENSITY`, a program utilizing 4 kB of memory would require 8.96 kB of RAM for profiling alone.

$$froms[\,]\ memory\ allocation = \frac{text\ size * size\ of\ entry}{HASHFRACTION}$$

$$froms[\,]\ memory\ allocation = \frac{4096 * 2\ B}{2} = 4096\ B\ or\ 4\ kB$$

$$tos[\,]\ \ memory\ allocation = test\ size * arc\ density\ \% * size\ of\ entry$$

$$tos[\,]\ \ memory\ allocation = 4096 * 2\% * 12\ B \sim 984\ B$$

$$kcount[\,]\ memory\ allocation = \frac{text\ size * size\ of\ entry}{HISTFRACTION}$$

$$kcount[\,]\ memory\ allocation = \frac{4096 * 2\ B}{2} = 4096\ B\ or\ 4\ kB$$

`monstartup()` is used to initialize the profiling data structures. It is called at the beginning of the `main()` function. `monstartup()` takes the start and end address of the program's memory as arguments. The memory addresses `MEM_START` and `MEM_END` are defined as macros in profil.h. Using these values, it computes the text size. From the text size, `monstartup()` can compute and dynamically allocate the appropriate space for the `froms[]`, `tos[]`, and `kcount[]` arrays as described earlier. The computed values and array pointers are all stored in the `_gmonparam` data structure as shown in Figure 31. After initializing the data structure, it makes a call to `moncontrol()`. `moncontrol()` is used to start and stop the profiling process. It changes the state and uses `profil()` (discussed more in Section 4.4.6) to signal to the statistical profiler to start or stop profiling. The complete call chain is shown in Figure 32. The code excerpts for `moncontrol()` and `monstartup()` are shown in Figures 33 and 34, respectively.

```
/*
 * The profiling data structures are housed in this structure.
 */
struct gmonparam {
 long int        state;      /* profiling state */
 unsigned short  *kcount;    /* array of PC sample counters */
 unsigned long   kcountsize; /* size of kcount[] array in bytes */
 ARCINDEX        *froms;     /* array of hashed from addresses containing index into tos[] */
 unsigned long   fromssize;  /* size of froms[] array in bytes */
 struct tostruct *tos;       /* array of tos addresses with counter */
 unsigned long   tossize;    /* size of tos[] array in bytes */
 long            tolimit;    /* max number of tos[] elements */
 unsigned long   lowpc;      /* lower memory address bound */
 unsigned long   highpc;     /* upper memory address bound */
 unsigned long   textsize;   /* total memory size */
 unsigned long   hashfraction;     /* divider for froms[] hash */
 long            log_hashfraction; /* precomputed shift amount for hash division */
};
```

Figure 31 - gmonparam Data Structure

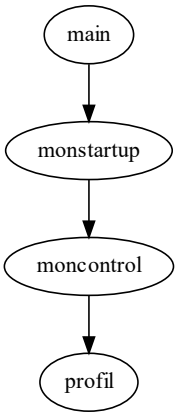

Figure 33 - monstartup() Call Chain

```
static void moncontrol (int mode) {
  struct gmonparam *p = &_gmonparam;

  /* Don't change the state if we ran into an error.  */
  if (p->state == GMON_PROF_ERROR)
    return;

  if (mode)
    {
      /* start */
      profil((void *) p->kcount, p->kcountsize, p->lowpc, s_scale);
      p->state = GMON_PROF_ON;
    }
  else
    {
      /* stop */
      profil(NULL, 0, 0, 0);
      p->state = GMON_PROF_OFF;
    }
}
```

Figure 32 - moncontrol Code

```
void monstartup (uint32_t lowpc, uint32_t highpc) {
  int o;
  char *cp;
  struct gmonparam *p = &_gmonparam;

  /*
   * round lowpc and highpc to multiples of the density we're using
   * so the rest of the scaling (here and in gprof) stays in ints.
   */
  p->lowpc = ROUNDDOWN(lowpc, HISTFRACTION * sizeof(HISTCOUNTER));
  p->highpc = ROUNDUP(highpc, HISTFRACTION * sizeof(HISTCOUNTER));
  p->textsize = p->highpc - p->lowpc;
  p->kcountsize = ROUNDUP(p->textsize / HISTFRACTION, sizeof(*p->froms));
  p->hashfraction = HASHFRACTION;
  p->log_hashfraction = -1;
  /* The following test must be kept in sync with the corresponding
     test in mcount.c.  */
  if ((HASHFRACTION & (HASHFRACTION - 1)) == 0) {
      /* if HASHFRACTION is a power of two, mcount can use shifting
   instead of integer division.  Precompute shift amount. */
      p->log_hashfraction = ffs(p->hashfraction * sizeof(*p->froms)) - 1;
  }
  p->fromssize = p->textsize / HASHFRACTION;
  p->tolimit = p->textsize * ARCDENSITY / 100;
  if (p->tolimit < MINARCS)
    p->tolimit = MINARCS;
  else if (p->tolimit > MAXARCS)
    p->tolimit = MAXARCS;
  p->tossize = p->tolimit * sizeof(struct tostruct);

  cp = calloc (p->kcountsize + p->fromssize + p->tossize, 1);
  if (! cp)
    {
      ERR("monstartup: out of memory\n");
      p->tos = NULL;
      p->state = GMON_PROF_ERROR;
      return;
    }
  p->tos = (struct tostruct *)cp;
  cp += p->tossize;
  p->kcount = (HISTCOUNTER *)cp;
  cp += p->kcountsize;
  p->froms = (ARCINDEX *)cp;

  p->tos[0].link = 0;

  o = p->highpc - p->lowpc;
  if (p->kcountsize < (uint32_t) o)
    s_scale = ((float)p->kcountsize / o ) * SCALE_1_TO_1;
  else
    s_scale = SCALE_1_TO_1;

  moncontrol(1);
}
```

Figure 34 - monstartup Code

The final call chain of the host PC application begins with `main()` invoking `_mcleanup()` after all the profiling data samples have been processed. `_mcleanup()` frees the memory that was dynamically allocated and calls `moncontrol()` and `write_gmon()`. Figure 35 shows the complete call chain. Figure 36 shows the code for `_mcleanup()` and `write_gmon()`. `moncontrol()` again calls `profil()` to end and cleanup the processing of statistical profiling samples. `write_gmon()` is responsible for migrating the profiling information from the data structures in the program's memory to an output file. It creates a file called gmon.out in the working directory. gprof specifies a specific format for the gmon.out file so it can be interpreted correctly. This application uses version 0x51879. gprof expects a special header that identifies the lower bound memory address, upper bound memory address, gmon version, profiling rate, and the size of the histogram. This header informs gprof how much data to expect and how the data will be formulated.

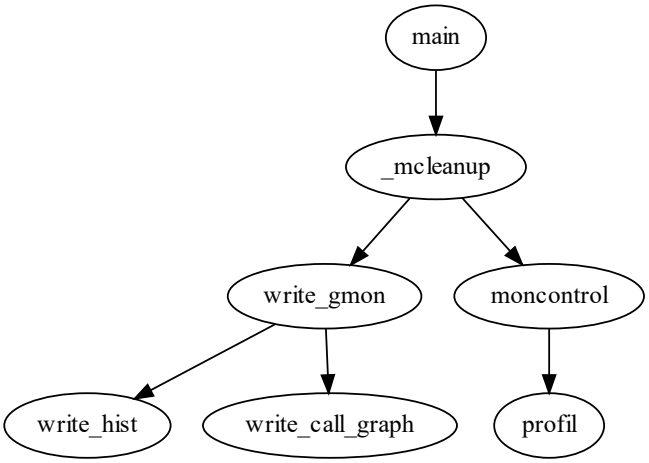

Figure 35 - _mcleanup() Call Chain

```
static void write_gmon (void) {

  /* create gmon.out file */
  int fd = -1;
  fd = open ("gmon.out", O_CREAT|O_TRUNC|O_WRONLY|O_BINARY, 0666);
  if (fd < 0) {
    ERR("_mcleanup: gmon.out\n");
    return;
  }

  /* write gmon header */
  struct gmon_hist_hdr thdr, *hdr;
  hdr = &thdr;
  thdr.low_pc = _gmonparam.lowpc;
  thdr.high_pc = _gmonparam.highpc;
  thdr.hist_size = _gmonparam.kcountsize + sizeof (thdr);
  thdr.prof_rate = PROF_HZ;
  thdr.version = GMON_VERSION;
  write(fd, (char *)hdr, sizeof *hdr);

  /* write PC histogram */
  write_hist (fd);
  /* write call-graph */
  write_call_graph (fd);
  /* close file */
  close (fd);
}

void _mcleanup (void) {
  /* stop */
  moncontrol (0);
  /* create gmon.out */
  if (_gmonparam.state != GMON_PROF_ERROR)
    write_gmon ();
  /* free the memory. */
  free (_gmonparam.tos);
}
```

Figure 36 - _mcleanup & write_gmon Code

After `write_gmon()` creates the gmon.out file and writes the header, it calls `write_hist()`. `write_hist()` merely writes the `kcount[]` array directly to gmon.out. After `write_hist()` completes `write_call_graph()` executes. Writing arc profiling data to gmon.out is slightly more involved since it requires traversing the call graph. Every write consists of a caller address, a callee address, and a counter signifying the number of times that

arc was traversed. To fully traverse the call graph, the program steps through each element of the `froms[]` array. If a `tos[]` index is present in that element, it jumps into the `tos[]` array. It visits all of the links related to that `tos[]` index and writes each call arc to the file. After visiting every element in the froms[] array, `write_call_graph()` returns to `write_gmon()` which closes gmon.out file and returns. Finally, `_mcleanup()` returns, and the program exits. `write_hist()` and `write_call_graph()` are shown in Figure 37.

```
static void write_hist (int fd) {
  write(fd, _gmonparam.kcount, _gmonparam.kcountsize);
}

static void write_call_graph (int fd) {
  ARCINDEX from_index, to_index;
  uint32_t from_len;
  uint32_t frompc;

  from_len = _gmonparam.fromssize / sizeof (*_gmonparam.froms);
  for (from_index = 0; from_index < from_len; ++from_index) {
    if (_gmonparam.froms[from_index] == 0)
      continue;

    frompc = _gmonparam.lowpc;
    frompc += (from_index * _gmonparam.hashfraction * sizeof (*_gmonparam.froms));
    for (to_index = _gmonparam.froms[from_index]; to_index != 0; to_index = _gmonparam.tos[to_index].link) {
      struct rawarc arc;
      arc.raw_frompc = frompc;
      arc.raw_selfpc = _gmonparam.tos[to_index].selfpc;
      arc.raw_count = _gmonparam.tos[to_index].count;
      write(fd, &arc, sizeof(arc));

    }
  }
}
```

Figure 37 - write_hist & write_call_graph Code

4.4.5 Call Graph Arcs – mcount.c

The mcount.c file contains only the `_mcount_internal()` function. This function is responsible for processing the individual call graph arcs. Its input arguments are the “from” memory address and the “to” memory address for the call arc. This function was adapted from the GNU C Library (glibc) version 2.30 for this implementation. The code is shown in Figure 38.

`_mcount_internal()` first checks the arguments to make sure their address values are sensible based on the program's understanding of the lower and upper memory bounds. Using the `HASHFRACTION`, it calculates the `froms[]` element corresponding to the from memory address received. The value stored in that `froms[]` element provides the index for the `tos[]` array. If the value is 0, the arc has not been traversed before and a new `tos[]` element needs to be created with the destination address and a counter of 1. If the `froms[]` element provides a valid `tos[]` index value, the program moves on to that element in the `tos[]` array. There is a member of the `tostruct` called link which helps track the call chain. The link contains either the `tos[]` index value of the next call in the chain or a 0 if it is the end of the chain. The program checks to make sure the incoming `selfpc` and the `selfpc` of the `tos[]` element match to confirm that it has found the correct destination in the chain. Typically, the correct `selfpc` is at the head of the chain and there is no need to traverse the links. Once the correct destination is found and the counter is incremented, `_mcount_internal()` returns a positive status.

```
void _mcount_internal(uint32_t frompc, uint32_t selfpc) {
  ARCINDEX *frompcindex;
  struct tostruct *top, *prevtop;
  struct gmonparam *p;
  ARCINDEX toindex;
  int i;

  p = &_gmonparam;

  /*
   * check that frompcindex is a reasonable pc value.
   * for example: signal catchers get called from the stack,
   *    not from text space.  too bad.
   */
  frompc -= p->lowpc;
  if (frompc > p->textsize)
    goto done;

  /* The following test used to be
    if (p->log_hashfraction >= 0)
     But we can simplify this if we assume the profiling data
     is always initialized by the functions in gmon.c.  But
     then it is possible to avoid a runtime check and use the
     same `if' as in gmon.c.  So keep these tests in sync.  */
  if ((HASHFRACTION & (HASHFRACTION - 1)) == 0) {
    /* avoid integer divide if possible: */
      i = frompc >> p->log_hashfraction;
  } else {
      i = frompc / (p->hashfraction * sizeof(*p->froms));
  }
  frompcindex = &p->froms[i];
  toindex = *frompcindex;
  if (toindex == 0) {
    /*
     *  first time traversing this arc
     */
    toindex = ++p->tos[0].link;
    if (toindex >= p->tolimit)
      /* halt further profiling */
      goto overflow;

    *frompcindex = toindex;
    top = &p->tos[toindex];
    top->selfpc = selfpc;
    top->count = 1;
    top->link = 0;
    goto done;
  }
  top = &p->tos[toindex];
  if (top->selfpc == selfpc) {
    /*
     * arc at front of chain; usual case.
     */
    top->count++;
    goto done;
  }
    /*
     * have to go looking down chain for it.
     * top points to what we are looking at,
     * prevtop points to previous top.
     * we know it is not at the head of the chain.
     */
    for (; /* goto done */; ) {
      if (top->link == 0) {
        /*
         * top is end of the chain and none of the chain
         * had top->selfpc == selfpc.
         * so we allocate a new tostruct
         * and link it to the head of the chain.
         */
        toindex = ++p->tos[0].link;
        if (toindex >= p->tolimit)
          goto overflow;

        top = &p->tos[toindex];
        top->selfpc = selfpc;
        top->count = 1;
        top->link = *frompcindex;
        *frompcindex = toindex;
        goto done;
      }
      /*
       * otherwise, check the next arc on the chain.
       */
      prevtop = top;
      top = &p->tos[top->link];
      if (top->selfpc == selfpc) {
        /*
         * there it is.
         * increment its count
         * move it to the head of the chain.
         */
        top->count++;
        toindex = prevtop->link;
        prevtop->link = top->link;
        top->link = *frompcindex;
        *frompcindex = toindex;
        goto done;
      }

    }
  done:
    p->state = GMON_PROF_ON;
    return;
  overflow:
    p->state = GMON_PROF_ERROR;
    return;
}
```

Figure 38 - _mcount_internal Code

4.4.6 Program Counter Samples – profil.c

There are two functions defined in profil.c: `profil()` and `profil_count()`. `profil()` is responsible for setting up the local data structure for program counter samples. The local structure is a static structure that points to the `kcount[]` array for safe access. `profil()` is used to start and stop whatever hardware or software is responsible for the periodic triggering for capturing program counter samples. Since this implementation does not have to acquire program counter samples and only processes them, the only goal of `profil()` is to set up a local static copy of the profiling data and set the parameters used to convert memory addresses to `kcount[]` index values. The local declarations are in Figure 39 and the `profil()` setup is in Figure 40.

```
static uint32_t nsamples;
static uint32_t pc_offset;
static uint32_t pc_scale;

/* global profinfo for profil() call */
static struct profinfo prof = {
  PROFILE_NOT_INIT, /* profiling state */
  0,  /* profiling counters */
  0,  /* range to be profiled - lowpc */
  0,  /* range to be profiled - highpc */
  0   /* scale value of bins */
};
```

Figure 39 - Static profil Declarations

```
/*  Enable statistical profiling, writing samples of the PC into at most
    SIZE bytes of SAMPLE_BUFFER; every processor clock tick while profiling
    is enabled, the system examines the user PC and increments
    SAMPLE_BUFFER[((PC - OFFSET) / 2) * SCALE / 65536].  If SCALE is zero,
    disable profiling.  Returns zero on success, -1 on error.  */
int profil (char *samples, uint32_t size, uint32_t offset, uint32_t scale) {
  nsamples = size;
  pc_scale = scale;
  pc_offset = offset;

  if (scale > 65536) {
    errno = EINVAL;
    return -1;
  }

  /* profiling on */
  if (scale) {
    memset(samples, 0, size);
    memset(&prof, 0, sizeof prof);
    prof.counter = (uint16_t*)samples;
    prof.lowpc = offset;
    prof.highpc = offset + ((((size >> 1) << 16) / scale) << 1);
    prof.scale = scale;
    prof.state = PROFILE_ON;
  }
  /* profiling off */
  else
    prof.state = PROFILE_OFF;

  return 0;
}
```

Figure 40 - profil Code

`profil_count()` is tasked with incrementing the counter at the corresponding index for the program counter's memory address that was encountered. Figure 41 displays the full code for `profil_count()`. When used in its typical application, this function is triggered by whatever mechanism is providing the periodicity for statistical sampling. Since the program counter sample collection is happening inside the microcontroller, `main()` invokes `profil_count()`. Its only responsibility is incrementing the corresponding counter. `profil_count()` uses the values calculated in `profil()` to compute the index value. It performs a simple check to make sure the index value is in bounds before incrementing the counter at that array index.

```
/* increment profile counter */
void profil_count (uint32_t pc) {
  uint32_t i = ((pc - pc_offset) / 2) * pc_scale / 65536;
  // make sure its in bounds
  if (i < nsamples)
    prof.counter[i]++;
}
```

Figure 41 - profil_count Code

4.5 gprof Profile

Once the host application has successfully executed, it will have created the gmon.out file. This file contains the combined profiling data from the target microcontroller's execution. In order to create the profile, the gmon.out file must be run through gprof with the executable file from the target microcontroller. Providing the executable file allows gprof to fill in the gaps and add intelligent names to the profile samples and call graph arcs. Once gprof is installed, it is run through the command line via the command shown below. gprof will dump the profile to stdout.

```
gprof blinky.elf gmon.out

(prints out in terminal)

gprof blinky.elf gmon.out > gprof_profile.txt

(writes to txt file)
```

The flat profile shows the distribution of execution time across the different functions in the target program. The functions are shown in decreasing order of execution time consumption. The flat profile is generated predominantly from the program counter samples, but it also contains some information from the call graph arcs. Table II shows the definitions of each portion of the flat profile. Figure 42 shows a sample gprof flat profile from an RGB blinky program. A full gprof output is shown in the Github repository linked in Appendix A.

Table II - Flat Profile Definitions

Column Name	Column Description
%	The percentage of execution time each function consumes.
cumulative	The running cumulative sum of execution times by function.
self	The raw execution time of each function.
calls	The number of times the function was invoked based on call graph arcs.
self ms/call	The average amount of time per call spent in that function.
total ms/call	The average amount of time spent in that function and its callee functions.
name	The name of the function.

```
Flat profile:

Each sample counts as 0.001 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 59.35     10.22    10.22                             loop
 18.93     13.48     3.26    13155     0.25     0.25  WAIT1_WaitCycles
  9.97     15.19     1.72                             WAIT1_Wait100Cycles
  9.51     16.83     1.64                             _mcount_internal
  1.16     17.03     0.20                             WAIT1_Wait10Cycles
  1.08     17.21     0.19    13155     0.01     0.26  WAIT1_WaitLongCycles
  0.01     17.21     0.00                             __gnu_mcount_nc
  0.00     17.21     0.00       26     0.00   132.46  WAIT1_Waitms
  0.00     17.21     0.00       26     0.00     0.00  control_LEDs
  0.00     17.21     0.00       14     0.00     0.00  BitIoLdd4_ClrVal
  0.00     17.21     0.00       13     0.00     0.00  BitIoLdd2_ClrVal
  0.00     17.21     0.00       13     0.00     0.00  BitIoLdd2_SetVal
  0.00     17.21     0.00       13     0.00     0.00  BitIoLdd3_ClrVal
  0.00     17.21     0.00       13     0.00     0.00  BitIoLdd3_SetVal
  0.00     17.21     0.00       12     0.00     0.00  BitIoLdd4_SetVal
```

Figure 42 - Blinky Sample gprof Flat Profile

Reading the call graph profile can be a little less intuitive. Functions are given their own indexes and are listed in ascending order of indexes. Each function has its own entry in the call graph. These entries are demarcated by the horizontal lines. Each entry has at least one line showing the data for that entry's function. The functions listed above that function in the entry are its parents. The functions listed below are its children. For example, the entry with index [3] in Figure 43 corresponds to the function WAIT1_Waitms [3]. Its parent is main [4] and its child is WAIT1_WaitLongCycles [2]. The definitions of the terms in the call graph are shown in Table III. Tables IV & V show the individual definitions for the parent functions and child functions respectively.

Table III - Call Graph Profile Definitions

Column Name	Column Description
index	A unique number given to every function.
% time	The percentage of execution time each function and its children consume.
self	The raw execution time of each function.
children	The total execution spent in the subroutine calls made by this function.
called	The number of times the function was invoked based on call graph arcs.
name	The name of the function with the index number.

Table IV - Call Graph Profile Parent Definitions

Column Name	Column Description
self	Estimate of the execution time spent in this entry's function when this parent was the caller.
children	Estimate of the execution time spent in this entry's children when this parent was the caller.
called	The number of times this parent called this entry's function over the total number of times this function was called.
name	The name of the parent function with the index number.

Table V - Call Graph Profile Child Definitions

Column Name	Column Description
self	Estimate of the execution time spent in this child function when this entry's function was the caller.
children	Estimate of the execution time spent in the child functions of this child function when this entry's function was the caller.
called	The number of times the function called this child over the total number of times the child was called.
name	The name of the child function with the index number.

```
                     Call graph (explanation follows)

index % time    self  children    called     name
                                                 <spontaneous>
[1]     59.4   10.22    0.00                 loop [1]
-----------------------------------------------
                0.19    3.26   13155/13155       WAIT1_Waitms [3]
[2]     20.0    0.19    3.26   13155         WAIT1_WaitLongCycles [2]
                3.26    0.00   13155/13155       WAIT1_WaitCycles [5]
-----------------------------------------------
                0.00    3.44      26/26          main [4]
[3]     20.0    0.00    3.44      26          WAIT1_Waitms [3]
                0.19    3.26   13155/13155       WAIT1_WaitLongCycles [2]
-----------------------------------------------
                                                 <spontaneous>
[4]     20.0    0.00    3.44                 main [4]
                0.00    3.44      26/26          WAIT1_Waitms [3]
                0.00    0.00      26/26          control_LEDs [10]
-----------------------------------------------
                3.26    0.00   13155/13155       WAIT1_WaitLongCycles [2]
[5]     18.9    3.26    0.00   13155         WAIT1_WaitCycles [5]
-----------------------------------------------
                                                 <spontaneous>
[6]     10.0    1.72    0.00                 WAIT1_Wait100Cycles [6]
-----------------------------------------------
                                                 <spontaneous>
[7]      9.5    1.64    0.00                 _mcount_internal [7]
-----------------------------------------------
                                                 <spontaneous>
[8]      1.2    0.20    0.00                 WAIT1_Wait10Cycles [8]
-----------------------------------------------
                                                 <spontaneous>
[9]      0.0    0.00    0.00                 __gnu_mcount_nc [9]
-----------------------------------------------
                0.00    0.00      26/26          main [4]
[10]     0.0    0.00    0.00      26          control_LEDs [10]
                0.00    0.00      14/14          BitIoLdd4_ClrVal [11]
                0.00    0.00      13/13          BitIoLdd2_ClrVal [12]
                0.00    0.00      13/13          BitIoLdd2_SetVal [13]
                0.00    0.00      13/13          BitIoLdd3_ClrVal [14]
                0.00    0.00      13/13          BitIoLdd3_SetVal [15]
                0.00    0.00      12/12          BitIoLdd4_SetVal [16]
-----------------------------------------------
                0.00    0.00      14/14          control_LEDs [10]
[11]     0.0    0.00    0.00      14          BitIoLdd4_ClrVal [11]
-----------------------------------------------
                0.00    0.00      13/13          control_LEDs [10]
[12]     0.0    0.00    0.00      13          BitIoLdd2_ClrVal [12]
-----------------------------------------------
                0.00    0.00      13/13          control_LEDs [10]
[13]     0.0    0.00    0.00      13          BitIoLdd2_SetVal [13]
-----------------------------------------------
                0.00    0.00      13/13          control_LEDs [10]
[14]     0.0    0.00    0.00      13          BitIoLdd3_ClrVal [14]
-----------------------------------------------
                0.00    0.00      13/13          control_LEDs [10]
[15]     0.0    0.00    0.00      13          BitIoLdd3_SetVal [15]
-----------------------------------------------
                0.00    0.00      12/12          control_LEDs [10]
[16]     0.0    0.00    0.00      12          BitIoLdd4_SetVal [16]
```

Figure 43 - Blinky Sample Call Graph Profile

To make it easier to interpret the call graph, José Fonseca wrote a Python script to create a visual call graph from a gprof profile [15]. The program converts the text output into a dot graph which can be interpreted by GraphViz and written to an image file. The full details and user guides are provided in the GitHub repository [15]. Once the Python package has been installed with pip, it is very easy to generate a call graph image from the command line. Figure 44 shows the call graph for the gprof output detailed in Figures 42 and 43. It was generated with the following command.

```
gprof blinky.elf gmon.out | gprof2dot -e 0.0 -n 0.0 | dot -Tsvg -o ./blinky.svg
```

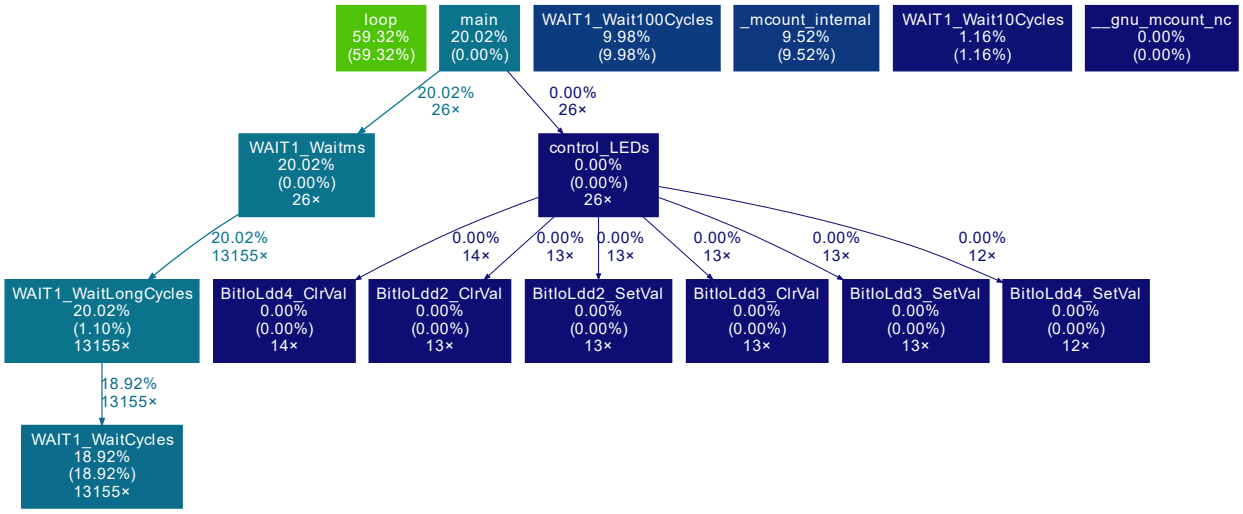

Figure 44 - Blinky Sample gprof2dot Call Graph

CHAPTER 5 – MICROCONTROLLER OVERHEAD ANALYSIS

5.1 Overview

It is important to quantify the overhead on the microcontroller required by the profiler. Since microcontrollers have limited hardware, detailing the memory and performance requirements of this profiling solution is imperative to confirm the solution will work in a specific application. This chapter will discuss the ROM and RAM requirements in addition to the performance overhead. Due to the nature of this profiling approach, there will be some variability in the overhead requirements depending upon the specific program and the microcontroller platform. The overhead analysis data is converted into a more generalized form which allows for more accurate application to new environments. By generalizing the data, inferences can be made more easily about the overhead necessitated by the profiler in new environments.

There are subtle differences from program to program and from microcontroller and microcontroller that cause this variability in overhead. These differences include the serial interface used, the queue structure employed, the processor architecture, and the frequency of profiler samples. For instance, particular types of serial interfaces on other processors may require different amounts of processing time and could affect program size. Queue implementations could require more instructions to protect data and require a larger queue size which would impact the ROM and RAM. The microarchitecture also plays a large role. The profiler could take longer to process samples due to the clock speed, instruction set architecture, or the cycles per instruction.

However, the biggest factor in the drastic differences in overhead is the program's rate of profiler samples. The rate of the program counter samples is easy to determine. It varies from

program to program, but the frequency is configured by the user. However, the number of call graph arc samples can be difficult to ascertain and is entirely dependent upon the program. Some programs have a lot of very short functions that call each other in quick succession. These types of programs generate a huge number of call graph arc samples. Other programs can have only a few functions with long execution times. These programs will generate a small number of call graph arc samples. The variability in call graph arc samples is what causes the huge range in profiler performance latencies. Since there is such variability, providing an overall latency in this analysis would not offer any insight. No inferences could be made about performance as this profiler was ported to new programs and systems. Instead, this analysis discusses the *latency per sample* to provide a more standardized metric that can be used to calculate expected performance delay in other environments.

The overhead analysis was performed on the KL25Z from NXP which has an ARM Cortex-M0+ core. The profiler used a 64 byte transmit buffer with a UART port operating at 1.5 MegaBaud. As discussed, the profiler's overhead will look different on other microcontrollers. This performance overhead analysis is meant to serve as a guidepost. It provides a framework to understand how this profiler will perform in other environments and provides a means for comparison against existing profiler solutions. The ROM usage, RAM usage, and performance latency of the profiler are discussed in the subsequent sections. Section 5.5 compares these metrics to the overhead required by the Semihosting gprof profiler discussed in Section 3.4.

5.2 ROM Analysis

To analyze the profiler's ROM usage, the program with the profiler source code must be compiled to an executable file. The executable file generated for the KL25Z is in the .elf format. The executable file must be inspected to find the ROM or text size of each of the functions

associated with the profiler. There are a number of different tools available to parse executable files including GNU ELF Parser, ELF Parser, Ghidra, and others. gprof can even be used to inspect the .elf file. Using the "-d" command forces gprof to output a vast amount of debug information including a memory map with the start and end address of every function. I was able to parse the executable file to find the size of each profiler function by using an ELF parser available through my Eclipse IDE. The results are shown in Table VI. The profiler consumes 1340 bytes of ROM. As discussed, this number should remain fairly consistent across different environments. Some subtle differences could arise based on microarchitecture, compiler optimization levels, serial interface, and queue implementations.

Table VI - Target MCU ROM Usage

Name	**Size**
__gnu_mcount_nc	44 B
init_gprof	36 B
deinit_gprof	48 B
_mcount_internal	184 B
init_PIT	184 B
Cpu_ivINT_PIT	216 B
TxQ_Init	40 B
TxQ_Size	20 B
TxQ_Enqueue	120 B
TxQ_Dequeue	116 B
init_UART2	260 B
Cpu_ivINT_UART2	76 B
Total	**1344 B**

5.3 RAM Analysis

When reviewing the overhead of a profiler, it is important to also consider the RAM consumption. This profiler does not dynamically allocate memory, so it is only necessary to

consider the static and automatic allocation. GCC can provide this information at compile-time by using the "-fstack-usage" option. This option tells GCC to generate an .su file for every source file. These .su files list the maximum stack usage per function. Since there is no dynamic heap memory consumed, these .su files provide sufficient information for understanding the total RAM consumption of the profiler. Table VII shows the combined information from the .su files. The maximum stack size can be calculated by using the hierarchical call graph and assuming all interrupts can be pushed on the stack at the same time. This absolute worst-case scenario would push all non-initialization functions on the stack resulting in a maximum stack size of 136 bytes. There can be some variance seen in the RAM usage by this profiler based on the size of the transmit queue buffer.

Table VII - Target MCU RAM Usage

Function	**Size**	**Type**
__gnu_mcount_nc	8 B	automatic
init_gprof	8 B	automatic
deinit_gprof	8 B	automatic
_mcount_internal	32 B	automatic
init_PIT	8 B	automatic
Cpu_ivINT_PIT	24 B	automatic
TxQ_Init	8 B	automatic
TxQ_Size	8 B	automatic
TxQ_Enqueue	32 B	automatic
TxQ_Dequeue	16 B	automatic
init_UART2	8 B	automatic
Cpu_ivINT_UART2	16 B	automatic
TxQ	64 B	static
TxQHead	2 B	static
TxQTail	2 B	static
TxQSize	2 B	static
Maximum Stack Size	**136 B**	

5.4 Performance Latency Analysis

To capture the performance overhead required by the profiler, timing analysis was carried out to determine the total time spent executing profiler code. As discussed in Section 5.1, there is significant variability in the rate of profiler data samples to be processed based on the runtime program and the microcontroller. To standardize the data to be more readily used to estimate the performance overhead in different applications, the latencies were measured on a per sample basis (i.e. the amount of time it takes to process and transmit a call graph arc sample and the amount of time it takes to process and transmit a program counter sample). These metrics can be used to easily extrapolate the total overhead required in new applications. The per sample program counter latency can be multiplied by the specified frequency to determine the total overhead per second. The same can be done for call graph arcs by heuristically determining or estimating the number of call graph arcs.

To accurately reflect the profiler performance overhead, there are three important time intervals that must be captured. They are the program counter sample processing delay (time spent in `Timer_ISR`), the call graph arc sample processing delay (time spent in `__gnu_mcount_nc`), and the serial data transmission delay (time spent in `Serial_ISR`). To measure each of these delays, a GPIO pin was set high upon entry into each function and set low upon exit to each function. A logic analyzer was used to measure the function's execution time by looking at the output pin's pulse width.

To accurately account for all the time required to process and transmit the profiler samples, the serial data transmission latency must be added to the sample processing latency. The serial data transmission latency reflects the amount of time required to transmit one byte of data, so this latency must be multiplied by the number of bytes in the profiler sample. Five cycles

must also be subtracted from the total delay to account for the time it takes modify the GPIO output. Since context switches did not occur during the measured execution time but factor into the performance overhead, the context switch delays must be added to the total latency. Fifteen cycles were added for each profiler-specific context switch. Table VIII shows the values that were measured. Equations 2a and 2b show how the latencies per sample were calculated.

Table VIII - Target MCU Performance Latency

Description	**Latency**	**# of cycles**
Call Graph Arc Sample Processing Latency	29.24 us	1404
Program Counter Sample Processing Latency	13.96 us	671
1 Byte Serial Transmission Latency	3.27 us	162
GPIO Pin Control Latency	104.17 ns	5
Context Switch Latency	312.50 ns	15
Latency per Call Graph Arc Sample	**57.07 us**	**2739**
Latency per Program Counter Sample	**30.08 us**	**1444**

$$\boldsymbol{Latency\ per\ Call\ Grach\ Arc\ Sample}$$

$$= CG\ Processing\ Latency - GPIO\ Latency + (Context\ Switch\ Latency * \#\ of\ Context\ Switches) + (Serial\ Transmission\ Latency * \#\ of\ bytes) \quad (Equation\ 2a)$$

$$= 29.24\ \text{us} - 104.17\ \text{ns} + (312.50\ ns * 16) + (3.27\ us * 7) = 57.07\ us$$

$$\boldsymbol{Latency\ per\ Program\ Counter\ Sample}$$

$$= PC\ Processing\ Latency - GPIO\ Latency + (Context\ Switch\ Latency * \#\ of\ Context\ Switches) + (Serial\ Transmission\ Latency * \#\ of\ bytes) \quad (Equation\ 2b)$$

$$= 13.96\ \text{us} - 104.17\ \text{ns} + (312.50\ ns * 10) + (3.27\ us * 4) = 30.08\ us$$

5.5 Overhead Comparison

To highlight the benefit of this low-footprint and relatively lightweight approach, the same profiler memory and performance metrics were also captured for the semihosting gprof profiler discussed in Section 3.4 [13]. The two different profilers were run on an identical program for the KL25Z with the same core configurations to reduce any unnecessary variables. The results for ROM usage, RAM usage, and latency are provided here.

Table IX shows the ROM comparison between the two profilers. The ROM of the profiler described in this thesis consists of the program counter sampler, call graph arc sampler, and the related serial and queueing code. The semihosting gprof approach consumes much more ROM even though it does not require a serial interface or a queueing structure. The necessary ROM to capture the program counter samples and the call graph arc samples is similar in both profilers. The additional ROM usage in the semihosting gprof approach comes from the library functions required to facilitate the file I/O manipulation for profiling. In this approach, the target microcontroller is responsible for creating the gmon.out file and writing to it.

Table IX - Profiler ROM Comparison

Profiler	**ROM Usage**
Thesis - gprof	1344 B
Semihosting gprof	2000 B

Table X shows the RAM usage of the two different profiling methodologies. There are two important types of RAM that must be considered: static and dynamic. As discussed earlier, the profiler discussed in this thesis only allocates RAM statically. This profiler has 70 bytes of static RAM usage and the worst-case maximum stack size is 136 bytes. The semihosting approach has a total static RAM allocation of 32 bytes with a worst-case maximum stack size of

336 bytes. However, the semihosting profiler also consumes additional RAM at run time. The profiler dynamically allocates space in RAM to store the arrays containing the individual call graph arc samples and program counter samples. The size of these arrays is based on the text size or the total ROM containing the program. Section 4.4.4 discusses in more detail how the dynamic memory size is computed. This calculation is summarized in Equation 3 assuming usage of the default hash divisors. A 4 kB program requires 8.96 kB of dynamic RAM on top of the 338 bytes of RAM already required for static variables and stack usage.

Table X - Profiler RAM Comparison

	Static Size	**Maximum Stack Size**	**Dynamic RAM**
Thesis - gprof	70 B	136 B	0 B
Semihosting gprof	32 B	336 B	* 8.96kB for 4kB ROM

$$* Total\ Dynamic\ RAM$$

$$= \frac{text * size\ of\ entry}{HASHFRACTION} + test * arc\ density\ \% * size\ of\ entry + \frac{text * size\ of\ entry}{HISTFRACTION} \quad (Equation\ 3)$$

$$= \frac{4096 * 2\ B}{2} + 4096 * 2\% * 12\ B + \frac{4096 * 2\ B}{2} = 9176\ B\ or\ 8.96\ kB$$

Table XI shows the latency comparisons between profilers to understand the performance overhead. As discussed, measuring the profiler's overall performance latency is not helpful. This information will not be indicative of profiler performance on other programs and across different microcontrollers. Therefore, the latency data was captured per call graph arc and per program counter sample. The Semihosting gprof approach requires much less performance overheard. The approach discussed in this thesis sacrifices some of the profiler's speed to conserve the

microcontroller's limited memory. The data enqueueing and serial data transmission is quite processor-intensive and is responsible for the increased latency.

Table XI - Profiler Latency Comparison

Description	**Thesis - gprof Latency**	**Semihosting gprof Latency**
Call Graph Arc Sample Processing Latency	29.24 us	6.21 us
Program Counter Sample Processing Latency	13.96 us	4.65 us
1 Byte Serial Transmission Latency	3.27 us	N/A
Debug Pin Control Latency	104.17 ns	104.17 ns
Context Switch Latency	312.50 ns	312.50 ns
Latency per Call Graph Arc Sample	**57.07 us**	**6.73 us**
Latency per Program Counter Sample	**30.08 us**	**5.28 us**

In conclusion, the semihosting gprof profiler provides an incredibly fast and lightweight approach. However, the low performance overhead comes at a significant memory cost. If the microcontroller cannot supply the RAM needed to facilitate this semihosting approach, this thesis provides a great solution. Despite being about six times slower than the semihosting approach, the profiler described in this thesis is still relatively lightweight when compared with other existing profiler solutions thanks to the distributed workload. It sacrifices speed but drastically reduces the memory footprint by using a serial port to communicate profiling data. Section 8.2.2 discusses some additional work that can be done to reduce the latency while maintaining the low memory footprint.

CHAPTER 6 – VALIDATION & RESULTS

6.1 Overview

To test this lightweight gprof implementation, I used an NXP KL25Z microcontroller built on an ARM Cortex-M0+ processor. Some additional cross-platform testing was performed on the NXP K64F microcontroller based on the ARM Cortex-M4 processor. The example target microcontroller programs provided were developed for the KL25Z. There are two different programs provided. These programs were developed in MCUXpresso, an Eclipse-based IDE from NXP. Blinky is a blinking RGB LED program that uses a simple main while loop. Shield is a more complicated program that uses FreeRTOS tasks to show the full functionality of a shield board made for the KL25Z. Each program was run several different times to produce multiple outputs with which to validate the functionality. The validation steps and the results are shared in the following sections.

6.2 Validation Steps

6.2.1 Target MCU Program

Validating the program counter sampling requires several different steps. First, I had to confirm the program counters being captured were correct. This required stepping through the code in debug mode and using breakpoints to monitor the stack. Since the ARM Cortex-M series processors have two stack pointers (MSP and PSP), I had to be certain to test separate cases where each of the different stack pointers were in use. A distinct offset is used in each circumstance to retrieve the appropriate program counter, so I had to prove the program was grabbing the correct value off of the stack. On the left, Figure 45 shows the value of the stack pointer (sp), the stack pointer + offset, and the program counter (pc) in the watch window when a

breakpoint is set in the program counter sampling interrupt. On the right, it shows a memory map with values beginning at the stack pointer and ending at the program counter.

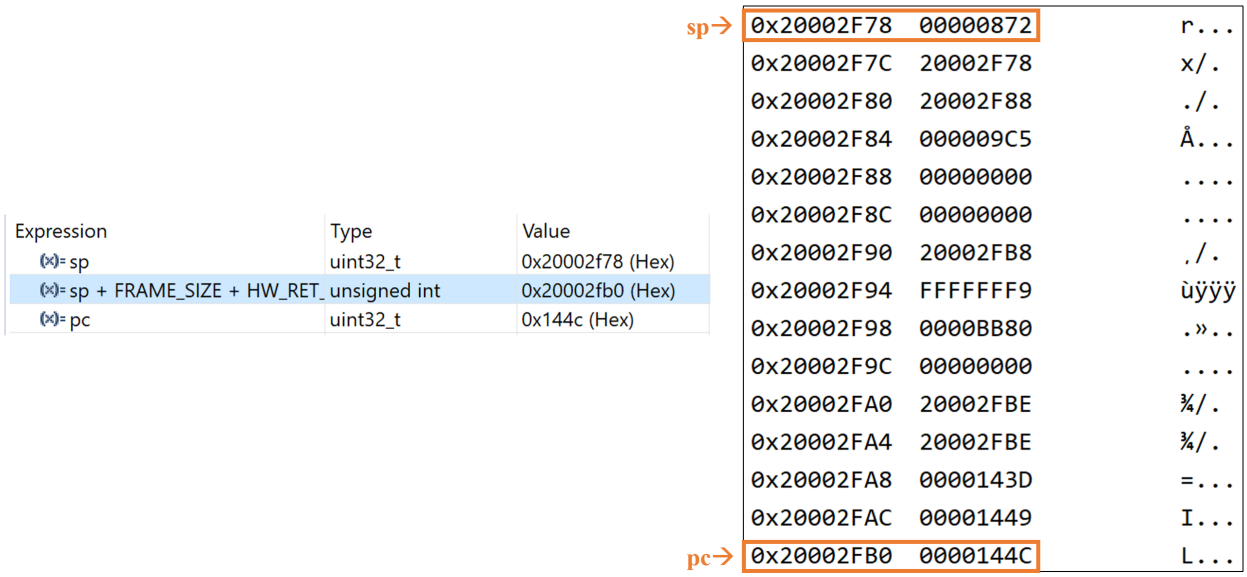

Figure 45 - Program Counter Verification

Once I confirmed the values being captured are accurate program counters, I connected a USB to serial interface to the KL25Z to monitor the serial data traffic. I stepped through the code in debug mode to compare the program counter value in the watch window to the value coming into my PC from the serial interface to verify that they match. Finally, it was necessary to check the timing accuracy of the program counter sampling. To be sure the sampling is occurring consistently at the specified interval, I added an instruction to toggle a GPIO pin in the program counter sampling interrupt. Using a logic analyzer, I recorded the pin status over time to confirm there were no deviations from the specified sampling frequency. Figure 46 shows a screenshot from the logic analyzer software of the pin toggling for a sampling frequency of 1 kHz.

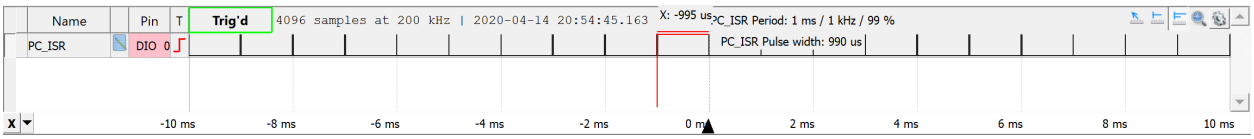

Figure 46 - PC Interrupt Timing

Validating the call arc required a similar approach. First, the callee and caller address capture had to be validated. To do this, I set a breakpoint in the `__gnu_mcount_nc()` assembly stub function and monitoring values in registers and on the stack. Figure 47 shows the status of the stack during the execution of `__gnu_mcount_nc()` for reference. `__gnu_mcount_nc()` moves the callee or "to" address to register r1 and the caller or "from" address to register r0. Figure 40 shows the memory map with values on the left beginning at the stack pointer. The callee address is in the link register and is an offset of #16 away from the stack pointer. The caller address is in the old link register and is an offset of #20 away from the stack pointer. Figure 48 also shows the register values on the right with the correct memory addresses stored in r0 and r1. These registers are used as input arguments to `_mcount_internal()`. By comparing these memory addresses to the .map file, it showed that the `Waitms()` function is calling `WaitLongCycles()` function which accurately depicted what was occurring in the program.

r0	<-- top of stack
r1	
r2	
r3	
lr (callee address)	
lr (caller address)	<-- initial top of stack, first lr push before __gnu_mcount_nc() call

Figure 47 - __gnu_mcount_nc() Stack

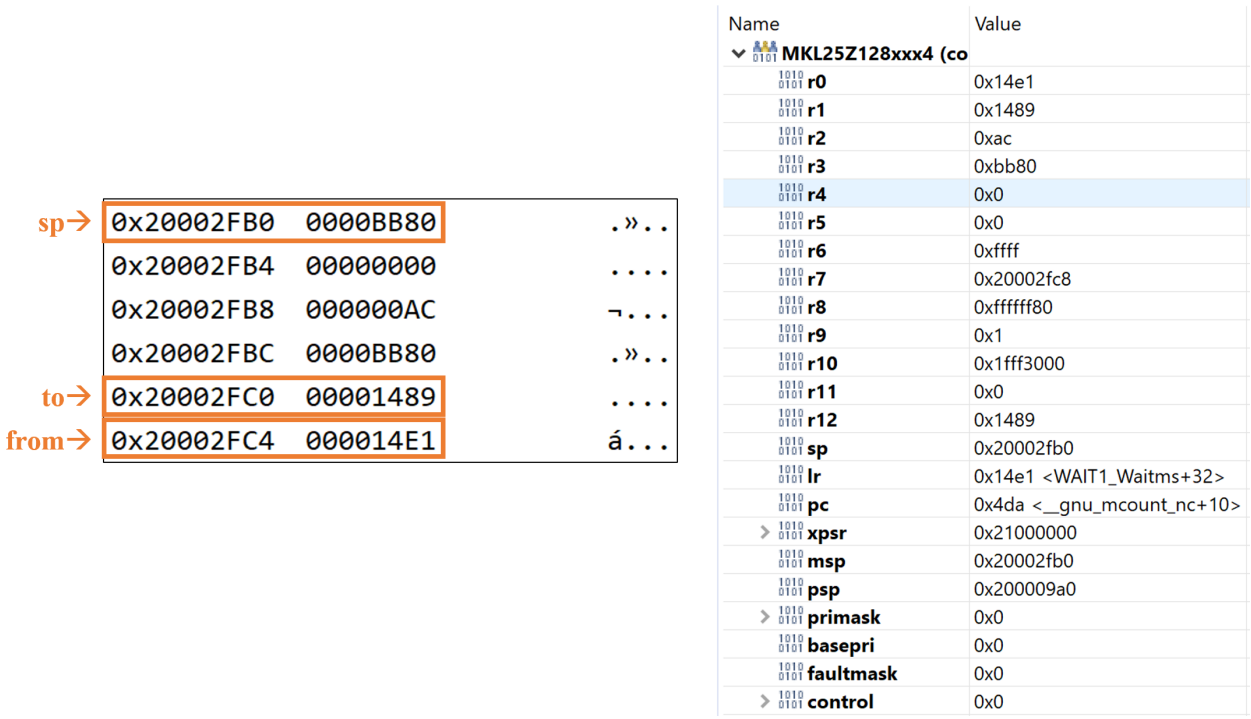

Figure 48 - Call Arc Address Verification

The “from” and “to” addresses being passed as arguments into `_mcount_internal()` have been validated. It was time to confirm the call arc samples being received on the serial port are identical and that there is no data pollution. I connected a USB to serial converter between the microcontroller and my PC and monitored the serial data being transmitted while stepping through the code. By doing this, I confirmed the memory addresses matched. I also ensured there was no cross pollution of program counter samples with call graph arc samples. Finally, I modified the Blinky program to make a fixed number of function calls. By setting a known, finite number of calls per function, I could reinterpret the data on the receiving end to make sure the number of calls per function matched. The data on the receiving end was identical validating that the entire call graph arc system worked.

6.2.2 Host PC Program

Even though the previous steps described validated the entire system, I found it necessary to be able to quickly validate the host PC program. I wrote a Python script (gmon_validation.py) to check the program's output. I selected Python because it allowed me the opportunity to more easily manipulate the data. The script also provided the added benefit of providing more confidence in the solution. By approaching the problem in a different manner, I could be positive the resulting profile is correct if this other approach produced the same output.

To be able to process the data in Python, I added a macro to main.c entitled `VALIDATION`. When this macro is defined, the host PC program converts the raw profiling data to a .csv file. This .csv file is used as the input to the Python script. The Python script accumulates this data and outputs two .csv files, one for program counter samples and one for call arcs. It sums up the number of occurrences. These two files can then be compared to the gprof profile using the .map file to understand how the memory addresses correspond to functions. Samples of these files are provided in GitHub repository: validation.csv, pc.csv, arc.csv. Figure 49 shows these results. A portion of the program counter samples from the output file pc.csv are on the left while the entirety of the call graph samples from the output file arc.csv are on the right.

pc.csv

```
Addr1,Counter
0x4d6,1
0x7ce,177
0x7f6,2
0x81c,14
0x8a0,1444
0x1466,399
0x1468,44
0x146a,2988
0x146c,500
0x146e,1196
.
.
.
```

arc.csv

```
Addr1,Addr2,Counter
0x527,0xf21,14
0x533,0xf45,12
0x545,0xdc5,13
0x551,0xde9,13
0x563,0xe75,13
0x56f,0xe95,13
0x5bd,0x4fd,26
0x5c7,0x1529,26
0x150d,0x1499,13155
0x153d,0x14e5,13155
```

Figure 49 - Python Validation Output

6.3 Results

Example Programs - There were 2 Target Example Programs for the KL25Z.

- Blinky program – prefix blink
- Shield program – prefix shield

Runs - There were 4 program executions captured for each of the example programs.

Results – There are 5 files provided for each run of each program. * *represents prefix.*

- *cg.png – gprof2dot call graph image
- *cg.svg – gprof2dot call graph scalable vector image
- *gmon.out – gmon output file from host PC application
- *gprof.txt – gprof output file from gprof
- *hex.txt – raw profiling data from target microcontroller

6.3.1 Blinky Program – blink1 run

This program flashes through the 8 different combinations of states of an RGB LED. It changes state every 500 milliseconds. This program simulates the more simplistic programs. It also provided a quick way to validate functionality and a framework that could easily be modified to test corner cases. The full data for this run and the rest of the Blinky runs are provided in the GitHub repository as discussed in Appendix A.

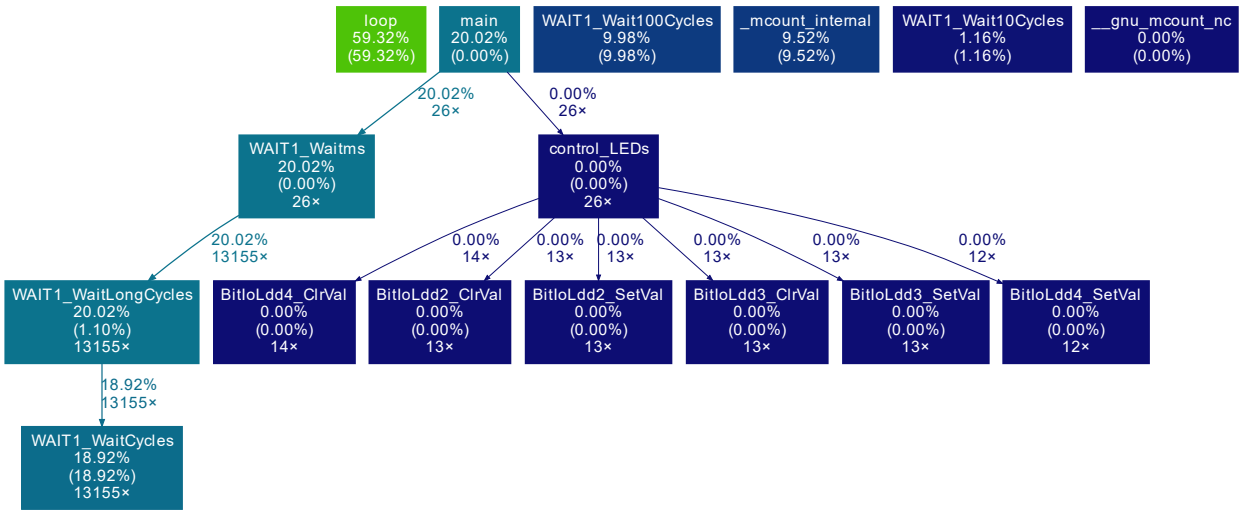

Figure 50 - Blinky Call Graph

```
Flat profile:
Each sample counts as 0.001 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 59.35     10.22    10.22                             loop
 18.93     13.48     3.26    13155     0.25     0.25  WAIT1_WaitCycles
  9.97     15.19     1.72                             WAIT1_Wait100Cycles
  9.51     16.83     1.64                             _mcount_internal
  1.16     17.03     0.20                             WAIT1_Wait10Cycles
  1.08     17.21     0.19    13155     0.01     0.26  WAIT1_WaitLongCycles
  0.01     17.21     0.00                             __gnu_mcount_nc
  0.00     17.21     0.00       26     0.00   132.46  WAIT1_Waitms
  0.00     17.21     0.00       26     0.00     0.00  control_LEDs
  0.00     17.21     0.00       14     0.00     0.00  BitIoLdd4_ClrVal
  0.00     17.21     0.00       13     0.00     0.00  BitIoLdd2_ClrVal
  0.00     17.21     0.00       13     0.00     0.00  BitIoLdd2_SetVal
  0.00     17.21     0.00       13     0.00     0.00  BitIoLdd3_ClrVal
  0.00     17.21     0.00       13     0.00     0.00  BitIoLdd3_SetVal
  0.00     17.21     0.00       12     0.00     0.00  BitIoLdd4_SetVal
```

Figure 51 - Blinky gprof Flat Profile

```
                     Call graph (explanation follows)

index % time    self  children    called     name
                                                 <spontaneous>
[1]     59.4   10.22    0.00                 loop [1]
-----------------------------------------------
                0.19    3.26   13155/13155       WAIT1_Waitms [3]
[2]     20.0    0.19    3.26   13155         WAIT1_WaitLongCycles [2]
                3.26    0.00   13155/13155       WAIT1_WaitCycles [5]
-----------------------------------------------
                0.00    3.44      26/26          main [4]
[3]     20.0    0.00    3.44      26          WAIT1_Waitms [3]
                0.19    3.26   13155/13155       WAIT1_WaitLongCycles [2]
-----------------------------------------------
                                                 <spontaneous>
[4]     20.0    0.00    3.44                 main [4]
                0.00    3.44      26/26          WAIT1_Waitms [3]
                0.00    0.00      26/26          control_LEDs [10]
-----------------------------------------------
                3.26    0.00   13155/13155       WAIT1_WaitLongCycles [2]
[5]     18.9    3.26    0.00   13155         WAIT1_WaitCycles [5]
-----------------------------------------------
                                                 <spontaneous>
[6]     10.0    1.72    0.00                 WAIT1_Wait100Cycles [6]
-----------------------------------------------
                                                 <spontaneous>
[7]      9.5    1.64    0.00                 _mcount_internal [7]
-----------------------------------------------
                                                 <spontaneous>
[8]      1.2    0.20    0.00                 WAIT1_Wait10Cycles [8]
-----------------------------------------------
                                                 <spontaneous>
[9]      0.0    0.00    0.00                 __gnu_mcount_nc [9]
-----------------------------------------------
                0.00    0.00      26/26          main [4]
[10]     0.0    0.00    0.00      26          control_LEDs [10]
                0.00    0.00      14/14          BitIoLdd4_ClrVal [11]
                0.00    0.00      13/13          BitIoLdd2_ClrVal [12]
                0.00    0.00      13/13          BitIoLdd2_SetVal [13]
                0.00    0.00      13/13          BitIoLdd3_ClrVal [14]
                0.00    0.00      13/13          BitIoLdd3_SetVal [15]
                0.00    0.00      12/12          BitIoLdd4_SetVal [16]
-----------------------------------------------
                0.00    0.00      14/14          control_LEDs [10]
[11]     0.0    0.00    0.00      14          BitIoLdd4_ClrVal [11]
-----------------------------------------------
                0.00    0.00      13/13          control_LEDs [10]
[12]     0.0    0.00    0.00      13          BitIoLdd2_ClrVal [12]
-----------------------------------------------
                0.00    0.00      13/13          control_LEDs [10]
[13]     0.0    0.00    0.00      13          BitIoLdd2_SetVal [13]
-----------------------------------------------
                0.00    0.00      13/13          control_LEDs [10]
[14]     0.0    0.00    0.00      13          BitIoLdd3_ClrVal [14]
-----------------------------------------------
                0.00    0.00      13/13          control_LEDs [10]
[15]     0.0    0.00    0.00      13          BitIoLdd3_SetVal [15]
-----------------------------------------------
                0.00    0.00      12/12          control_LEDs [10]
[16]     0.0    0.00    0.00      12          BitIoLdd4_SetVal [16]
```

Figure 52 - Blinky gprof Call Graph

6.3.2 Shield Program – shield1 run

This program showcases the functionality of a shield constructed for the KL25Z. It uses FreeRTOS to govern the tasks. This program models the more complex end of the spectrum. There are 9 different tasks which handle a command-line interface, RGB LED blinking, navigating FatFS, SD card reading/writing, touchscreen reading, display updating, accelerometer reading, speaker sound managing, and refilling the sound buffer. The full data for this run and the rest of the Shield runs is provided in the GitHub repository as discussed in Appendix A.

* can zoom in for more details

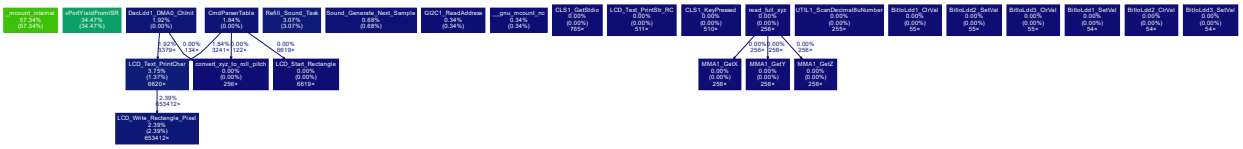

Figure 54 - Shield Call Graph

```
Flat profile:

Each sample counts as 0.2 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 57.34     33.60    33.60                             _mcount_internal
 34.47     53.80    20.20                             vPortYieldFromISR
  3.07     55.60     1.80                             Refill_Sound_Task
  2.39     57.00     1.40   653412     2.14     2.14  LCD_Write_Rectangle_Pixel
  1.37     57.80     0.80     6620   120.85   332.33  LCD_Text_PrintChar
  0.68     58.20     0.40                             Sound_Generate_Next_Sample
  0.34     58.40     0.20                             GI2C1_ReadAddress
  0.34     58.60     0.20                             __gnu_mcount_nc
  0.00     58.60     0.00     6619     0.00     0.00  LCD_Start_Rectangle
  0.00     58.60     0.00      765     0.00     0.00  CLS1_GetStdio
  0.00     58.60     0.00      511     0.00     0.00  LCD_Text_PrintStr_RC
  0.00     58.60     0.00      510     0.00     0.00  CLS1_KeyPressed
  0.00     58.60     0.00      256     0.00     0.00  MMA1_GetX
  0.00     58.60     0.00      256     0.00     0.00  MMA1_GetY
  0.00     58.60     0.00      256     0.00     0.00  MMA1_GetZ
  0.00     58.60     0.00      256     0.00     0.00  convert_xyz_to_roll_pitch
  0.00     58.60     0.00      256     0.00     0.00  read_full_xyz
  0.00     58.60     0.00      255     0.00     0.00  UTIL1_ScanDecimal8uNumber
  0.00     58.60     0.00       55     0.00     0.00  BitIoLdd1_ClrVal
  0.00     58.60     0.00       55     0.00     0.00  BitIoLdd2_SetVal
  0.00     58.60     0.00       55     0.00     0.00  BitIoLdd3_ClrVal
  0.00     58.60     0.00       54     0.00     0.00  BitIoLdd1_SetVal
  0.00     58.60     0.00       54     0.00     0.00  BitIoLdd2_ClrVal
  0.00     58.60     0.00       54     0.00     0.00  BitIoLdd3_SetVal
```

Figure 53 - Shield gprof Flat Profile

```
                Call graph (explanation follows)


granularity: each sample hit covers 4 byte(s) for 0.34% of 58.60 seconds

index % time    self  children    called     name
                                                 <spontaneous>
[1]     57.3   33.60    0.00                 _mcount_internal [1]
-----------------------------------------------
                                                 <spontaneous>
[2]     34.5   20.20    0.00                 vPortYieldFromISR [2]
-----------------------------------------------
                0.39    0.69    3241/6620        CmdParserTable [7]
                0.41    0.71    3379/6620        DacLdd1_DMA0_ChInit [6]
[3]      3.8    0.80    1.40    6620         LCD_Text_PrintChar [3]
                1.40    0.00  653412/653412      LCD_Write_Rectangle_Pixel [5]
-----------------------------------------------
                                                 <spontaneous>
[4]      3.1    1.80    0.00                 Refill_Sound_Task [4]
-----------------------------------------------
                1.40    0.00  653412/653412      LCD_Text_PrintChar [3]
[5]      2.4    1.40    0.00  653412         LCD_Write_Rectangle_Pixel [5]
-----------------------------------------------
                                                 <spontaneous>
[6]      1.9    0.00    1.12                 DacLdd1_DMA0_ChInit [6]
                0.41    0.71    3379/6620        LCD_Text_PrintChar [3]
                0.00    0.00     134/256         convert_xyz_to_roll_pitch [18]
-----------------------------------------------
                                                 <spontaneous>
[7]      1.8    0.00    1.08                 CmdParserTable [7]
                0.39    0.69    3241/6620        LCD_Text_PrintChar [3]
                0.00    0.00    6619/6619        LCD_Start_Rectangle [11]
                0.00    0.00     122/256         convert_xyz_to_roll_pitch [18]
-----------------------------------------------
                                                 <spontaneous>
[8]      0.7    0.40    0.00                 Sound_Generate_Next_Sample [8]
-----------------------------------------------
                                                 <spontaneous>
[9]      0.3    0.20    0.00                 GI2C1_ReadAddress [9]
-----------------------------------------------
                                                 <spontaneous>
[10]     0.3    0.20    0.00                 __gnu_mcount_nc [10]
-----------------------------------------------
                0.00    0.00    6619/6619        CmdParserTable [7]
[11]     0.0    0.00    0.00    6619         LCD_Start_Rectangle [11]
-----------------------------------------------
                0.00    0.00     255/765         FatFS_SD_Task [156]
                0.00    0.00     510/765         Cmd_Line_Task [80]
[12]     0.0    0.00    0.00     765         CLS1_GetStdio [12]
-----------------------------------------------
                0.00    0.00     511/511         Read_Accel_Task [198]
[13]     0.0    0.00    0.00     511         LCD_Text_PrintStr_RC [13]
-----------------------------------------------
                0.00    0.00     510/510         CLS1_ReadParseCommandTable [68]
[14]     0.0    0.00    0.00     510         CLS1_KeyPressed [14]
-----------------------------------------------
```

Figure 55 - Shield gprof Call Graph Part 1

```
-----------------------------------------------
                0.00    0.00     256/256         read_full_xyz [19]
[15]     0.0    0.00    0.00     256         MMA1_GetX [15]
-----------------------------------------------
                0.00    0.00     256/256         read_full_xyz [19]
[15]     0.0    0.00    0.00     256         MMA1_GetX [15]
-----------------------------------------------
                0.00    0.00     256/256         read_full_xyz [19]
[16]     0.0    0.00    0.00     256         MMA1_GetY [16]
-----------------------------------------------
                0.00    0.00     256/256         read_full_xyz [19]
[17]     0.0    0.00    0.00     256         MMA1_GetZ [17]
-----------------------------------------------
                0.00    0.00     122/256         CmdParserTable [7]
                0.00    0.00     134/256         DacLdd1_DMA0_ChInit [6]
[18]     0.0    0.00    0.00     256         convert_xyz_to_roll_pitch [18]
-----------------------------------------------
                0.00    0.00     256/256         Read_Accel_Task [198]
[19]     0.0    0.00    0.00     256         read_full_xyz [19]
                0.00    0.00     256/256         MMA1_GetX [15]
                0.00    0.00     256/256         MMA1_GetY [16]
                0.00    0.00     256/256         MMA1_GetZ [17]
-----------------------------------------------
                0.00    0.00     255/255         FAT1_CheckCardPresence [144]
[20]     0.0    0.00    0.00     255         UTIL1_ScanDecimal8uNumber [20]
-----------------------------------------------
                0.00    0.00      55/55          RGB_Task [197]
[21]     0.0    0.00    0.00      55         BitIoLdd1_ClrVal [21]
-----------------------------------------------
                0.00    0.00      55/55          RGB_Task [197]
[22]     0.0    0.00    0.00      55         BitIoLdd2_SetVal [22]
-----------------------------------------------
                0.00    0.00      55/55          RGB_Task [197]
[23]     0.0    0.00    0.00      55         BitIoLdd3_ClrVal [23]
-----------------------------------------------
                0.00    0.00      54/54          RGB_Task [197]
[24]     0.0    0.00    0.00      54         BitIoLdd1_SetVal [24]
-----------------------------------------------
                0.00    0.00      54/54          RGB_Task [197]
[25]     0.0    0.00    0.00      54         BitIoLdd2_ClrVal [25]
-----------------------------------------------
                0.00    0.00      54/54          RGB_Task [197]
[26]     0.0    0.00    0.00      54         BitIoLdd3_SetVal [26]
-----------------------------------------------
```

Figure 56 - Shield gprof Call Graph Part 2

CHAPTER 7 – CROSS-PLATFORM IMPLEMENTATION CHECKLIST

To adopt this approach for other microcontroller platforms, some modifications must be made. The following checklist has been created for ease of use. Once the checklist has been completed in full, you should have a working implementation for your environment. The checklist is divided up into four sections: 7.1 Requirements (requirements for your development environment and microcontroller), 7.2 Target Microcontroller Application (modifications to collect profiling data on your microcontroller), 7.3 Host PC Application (modifications to appropriately process the samples from your program), and 7.4 Troubleshooting (guide to common issues and how to mitigate them).

7.1 Requirements

Target Microcontroller

- ☐ Must be compiled using GCC.
- ☐ Must have a serial interface with which to transmit profiling data to the host PC.

Host PC

- ☐ Must have an environment to compile and run C code (preferably GCC).
- ☐ Must have a way to receive serial transmission of profiling data from the target microcontroller. Specific requirements vary based on implementation.
- ☐ Must have serial monitor software to support receiving and processing serial data coming from the target microcontroller. (Realterm, Tera Term, Putty, etc.)

7.2 Target Microcontroller Application

Queue – Adopting queue implementation for a new environment.

- ☐ Adjust queue size.
- ☐ Implement critical section protections for queue access.

Serial – Configuring the serial interface.

- ☐ Select serial interface: UART, SPI, I2C, USB, Debug Console, etc.
- ☐ Select the data rate.
- ☐ Set up initialization of serial interface and transmit interrupt.
- ☐ Set up dequeuing in transmit interrupt.

Initialization/Deinitialization – Enabling the start and stop of profiling.

- ☐ Set up profiling initialization function `init_gprof()`: initialize serial interface, initialize queue, initialize program counter periodic sampling interrupt, enable interrupt, and set `g_Initialized` to 1 to enable arc counting.
- ☐ Set up profiling deinitialization function `deinit_gprof()`: disable program counter sampling interrupt and set `g_Initialized` to 0 to disable arc counting.
- ☐ Decide when to start and stop profiling in your program and call `init_gprof()` and `deinit_gprof()`. Follow the guidelines in Section 4.2.

Call Arcs – Configuring call graph arc counting.

- ☐ Adopt `__gnu_mcount_nc()` assembly stub to fit environment.
- ☐ Modify the length of the data packet if need be based on the memory address range of the target microcontroller. The example implementation had a 128 kB of ROM, so all memory addresses could fit in 3 bytes.
- ☐ Adapt `_mcount_internal()` for the specific buffer queue and serial interface being used in this implementation.

Program Counter Samples – Configuring program counter sampling.

- ☐ Determine the interrupt methodology for periodic sampling.
- ☐ Set up the initialization function.
- ☐ Select sampling frequency.
- ☐ Create the interrupt service routine.
- ☐ Create a way to easily enable and disable interrupts.
- ☐ Figure out if there is a built-in function to retrieve the requested program counter.
- ☐ If not, figure out how to retrieve the stack pointer value and how large of an offset there is from the stack pointer to the program counter value you want to retrieve.
- ☐ Determine whether there are multiple stack pointers and how to determine which one needs to be used at a given time. Can use macro definitions currently in place for choosing between ARM process stack pointer and main stack pointer.
- ☐ Adapt the interrupt service routine for the specific buffer queue and serial interface being used in this implementation.

Compilation – Generating instrumented object code.

- ☐ Add "-pg" compiler flag on a per-file basis to tell GCC to add instrumented code for call graph arc handling.
- ☐ Determine which files need to be included and which files need to be excluded from profiling. Follow the guidelines in Section 4.2.
- ☐ If library functions need to be profiled, statically link library functions.
- ☐ Compile the program using GCC.

7.3 Host PC Application

Receiving Serial Data – Configuring data receiver.

- ☐ Setup connection. Determine whether it requires some additional external hardware like a logic analyzer or USB to Serial Converter.
- ☐ Setup serial data acquisition. Determine what software is required and the appropriate configurations to receive data properly.

Host Application – Updating program-dependent variables.

- ☐ Update profiling data input file name (`#define FILENAME` in profil.h).
- ☐ Update whether or not the profiling data requires the hex conversion discussed in Section 4.4.3 (`#define HEX_CONVERT` in profil.h).
- ☐ Update lower bound address for microcontroller's program memory (`#define MEM_START` in profil.h).
- ☐ Update upper bound address for microcontroller's program memory (`#define MEM_END` in profil.h).
- ☐ Update the size of the profiling samples' memory addresses in bytes (`#define MEM_SIZE` in profil.h).
- ☐ Update the frequency of the periodic program counter sampling (`#define PROF_HZ` in profil.h).

7.4 Troubleshooting Common Issues

Processor Hard Fault

If there is a hard fault that occurs only when the "-pg" compiler flag is present, it is likely the program is trying to profile something it should not. It is inserting instrumented code in a place that breaks normal functionality. Typically, this is due to some atypical hook function for

an RTOS or event handler. This can also be caused by certain protection methods for critical sections. Attempt toggling the "-pg" flat on different files until there is no longer a hard fault. If removing the "-pg" compiler flag fixes the problem, that specific source file should not be profiled. The issue with the GCC "-pg" compiler flag is that it cannot be applied on a per-function basis, only a per-file basis. However, by excluding an entire file from profiling, the final profile may lack some important details. To mitigate this issue, the offending function(s) can be moved to a new file that is excluded from profiling. The original file can now be successfully included in profiling.

Suspended Profiler

If the processor hangs in `_mcount_internal()` or in the program counter sampling interrupt, it is likely because the serial port is not able to keep up with the amount of profiling data being generated. The buffer fills up and the processor halts because it cannot queue any more data. There are a couple of tactics to overcome this hurdle. Increasing the serial port's data rate would provide the best improvement. If the serial port cannot be sped up anymore, the queue size can be increased.

If neither of those two changes can be made or are not solving the problem, the only choice is to reduce the amount of data. A simple way to do that is to reduce the program counter sampling frequency. Another way to decrease traffic is to reduce the number of files being profiled. If this is not possible, you could also try collecting the call graph arcs for one execution period and the program counter samples for a separate execution period. As long as the factors are kept the same across between the two executions, the data could be recombined on the host PC and given you an accurate and representative profile.

CHAPTER 8 – CONCLUSIONS & FUTURE WORK

8.1 Conclusion

This thesis addresses a need in the microcontroller design space by providing a critical tool used for design, analysis, and debugging. Microcontrollers are heavily limited in memory, speed, processing power, energy, and cost. Profiling these programs can be vital to identifying optimization opportunities to meet these tight, real-time processing constraints. Traditional profiling solutions fail microcontrollers due to the large amount of overhead inherently required. To create a working profiler for all microcontrollers, this solution migrates an existing, open-source profiler into the microcontroller development space. To overcome the processing limitations of the microcontroller and avoid the pitfalls of other microcontroller profilers, this profiler divides the processing effort between the microcontroller and the host PC. Despite the separation of effort, the data is reassembled without any corruption. The profiler can generate a profile with as much detail and insight as the existing solutions.

8.2 Future Work

8.2.1 More Automated Solution

The first area for future work is to develop a more elegant and sleek solution for the host PC application. Presently, the solution requires many manual steps to collect the data and generate the gmon.out file. This labor-intensive process is due to the many different variables across development environments including the operating system, IDE, microcontroller, and type of serial interface used for profiler communication. It would be a huge development effort to build an automated solution with interoperability across all of the different platforms in the face of these many nuances. Ideally, there would be one solution that could collect the data in a variety of forms, process the data, and generate the gmon.out file on any platform.

One such solution would be to develop a plugin for the Eclipse IDE. Eclipse is already supported across many different operating systems. There is a grassroots movement within the industry to make it *the* development platform for microcontrollers. Instead of relying on proprietary, piecemeal IDEs from the manufacturers with varying functionality, projects like GNU MCU Eclipse [16] try to build an open-source solution developing design, analysis, and debug tools that work across all microcontrollers and advance the industry together. Building a plugin for Eclipse would allow for rapid adoption and further improvements amongst this community. The application could leverage the existing serial terminal program within Eclipse to reduce the complexity of interpreting the data from various sources using different protocols. Instead of having to make and run the source code for the host PC application, the program would run natively within Eclipse and could be configured using a GUI. It could also tie in the GNU solutions already available in Eclipse to create visual histograms, call graphs, and more. Future work in this area would lead to more easy and widespread usage.

8.2.2 Data Traffic Reduction

As discussed throughout this thesis, some cases could occasionally occur where the volume of profiling data is too great for the serial interface to handle. Some future work could be done in this area to help reduce the data traffic. These improvements would allow this solution to more easily manage and profile complex programs that generate a significant amount of traffic like RTOS-based programs. This issue could be approached in several different manners.

One solution would be to create a small local cache structure to exploit any locality in profiling samples. For example, temporal locality of call graph arcs could be exploited by creating a cache structure to store the most recent call arc occurrences. In this case, it is assumed that a function call could be made multiple times within a short amount of time. The cache

structure would contain the “from” address, the “to” address, and a counter. The call graph arcs would be temporarily stored and accumulated in this cache structure. Each new occurrence of that call arc would increment the counter. Instead of sending a data packet for every call arc occurrence, data would only need to be transmitted when the arc is evicted from the cache. Upon eviction, the “from” address, “to” address, and the counter would be sent to the host PC application to add that sum to the total counter for that call arc. This solution could tremendously reduce the volume of data being transmitted without consuming much more memory or processing power from the microcontroller. The cache size and replacement policies could be configured to exploit the locality that provides the greatest reduction in serial data traffic for that individual program. A similar cache structure could be created for program counter samples.

Another solution to the overloaded serial port is to employ a lookup table. This idea is based on an LCD display-based profiling solution by Dr. Alexander Dean [17]. The lookup table would be stored in the microcontroller’s memory. There would be an entry for each function, and each entry would contain a lower bound and an upper bound of the memory range associated with that function. This solution would require several compilations of the program to ensure consistent compilation before writing the functions’ memory address range to a source file in the program’s memory. Each memory address range would correlate to a specific index that could be matched to a function. Instead of sending the entire memory address in a serial data packet, the program can just send the index value. If we assume less than 256 functions, the data packet would be reduced from the typical 3-4 bytes for a memory address to 1 byte for an index number. Reducing the volume of data will improve the latency and ensure the serial interface is not overwhelmed.

REFERENCES

[1] J. Thiel, “An overview of software performance analysis tools and techniques: From gprof to dtrace,” Washington University in St. Louis, Tech. Rep, 2006.

[2] S. L. Graham, P. B. Kessler, and M. K. Mckusick, “Gprof,” *ACM SIGPLAN Notices*, vol. 17, no. 6, pp. 120–126, Jan. 1982.

[3] S. L. Graham, P. B. Kessler, and M. K. Mckusick, “gprof,” *ACM SIGPLAN Notices*, vol. 39, no. 4, pp. 49–57, 2004.

[4] J. Fenlason and R. Stallman, "GNU gprof," *GNU Binutils,* 1988. [Online]. Available: http://www.gnu.org/software/binutils. [Accessed: 18-Mar-2020].

[5] J. Fenlason and R. Stallman, “GNU gprof,” *GNU Manuals*. [Online]. Available: https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_mono/gprof.html. [Accessed: 18-Mar-2020].

[6] D. E. Knuth, “Structured Programming with go to Statements,” *ACM Computing Surveys (CSUR)*, vol. 6, no. 4, pp. 261–301, 1974.

[7] G. M. Amdahl, "Computer Architecture and Amdahl's Law," in Computer, vol. 46, no. 12, pp. 38-46, Dec. 2013.

[8] D. A. Varley, “Practical experience of the limitations of gprof,” *Software: Practice and Experience*, vol. 23, no. 4, pp. 461–463, 1993.

[9] J. M. Spivey, “Fast, accurate call graph profiling,” *Software: Practice and Experience*, vol. 34, no. 3, pp. 249–264, 2004.

[10] J. Kraft, A. Wall, and H. Kienle, “Trace Recording for Embedded Systems: Lessons Learned from Five Industrial Projects,” *Runtime Verification Lecture Notes in Computer Science*, pp. 315–329, 2010.

[11] R. Boys, “Application Note 287: NXP Kinetis: FRDM-K64F Cortex-M4 Lab: ARM Keil MDK Toolkit featuring Serial Wire Viewer and ETM Trace,” *ARM Keil*, 2016. [Online]. Available: http://www.keil.com/appnotes/docs/apnt_287.asp. [Accessed: 19-Mar-2020].

[12] “Coresight Debug,” “CoreSight Debug Block Diagram”, *ARM Keil*, 2018. [Online]. Available: http://www2.keil.com/images/default-source/mdk5/coresight.png. [Accessed: 19-Mar-2020].

[13] E. Styger, “Tutorial: Using GNU Profiling (gprof) with ARM Cortex-M,” *MCU on Eclipse*, 23-Aug-2015. [Online]. Available: https://mcuoneclipse.com/2015/08/23/tutorial-using-gnu-profiling-gprof-with-arm-cortex-m/. [Accessed: 18-Mar-2020].

[14] Arm Ltd, “Procedure Call Standard for the Arm Architecture - ABI 2019Q4 documentation,” *ARM Developer*, 31-Jan-2020. [Online]. Available: https://developer.arm.com/docs/ihi0042/latest. [Accessed: 20-Mar-2020].

[15] J. Fonseca, “gprof2dot,” *GitHub Repository*, 09-Jan-2020. [Online]. Available: https://github.com/jrfonseca/gprof2dot. [Accessed: 20-Mar-2020].

[16] “GNU MCU Eclipse,” *GNU MCU Eclipse*, 07-Sep-2015. [Online]. Available: https://gnu-mcu-eclipse.github.io/. [Accessed: 18-Mar-2020].

[17] A. Dean, “ESO-19,” *GitHub Repository*, 05-Apr-2019. [Online]. Available: https://github.ncsu.edu/agdean/ESO-19/tree/master/Tools/GetRegions. [Accessed: 20-Mar-2020].

APPENDICES

APPENDIX A – CODE AND RESULTS

The code and results for this thesis are provided in a GitHub repository. This includes the target microcontroller application, the host personal computer application, two example demo programs for a microcontroller with the profiling functionality, and the results from profiling those demo programs.

Link to the GitHub Repository:

https://github.com/mjdargen/Low-Footprint-gprof-for-Microcontrollers/

Directory Overview

```
|-- home
    |-- Host PC Application    - Host PC Application Source Code
    |   |-- Validation         - Python Validation for Host PC Application
    |-- KL25Z gprof Blinky     - Blinky Demo Program for Target Microcontroller
    |-- KL25Z gprof Shield     - Shield Demo Program for Target Microcontroller
    |-- Results                - gprof Outputs for Validation Runs
    |   |-- blinky             - Results for Blinky Program
    |   |-- shield             - Results for Shield Program
    |-- Target MCU Application - Target Microcontroller Profiling Source Code
```

File Structure

```
|-- home
    |-- .gitattributes
    |-- .gitignore
    |-- LICENSE
    |-- README.md
    |-- Host PC Application
    |   |-- gmon.c
    |   |-- gmon.h
    |   |-- gmon_out.h
    |   |-- main.c
    |   |-- Makefile
    |   |-- mcount.c
    |   |-- profil.c
    |   |-- profil.h
    |   |-- Validation
    |       |-- arc.csv
    |       |-- gmon_validation.py
    |       |-- pc.csv
    |       |-- validation.csv
    |-- KL25Z gprof Blinky
    |   |-- ...
    |   |-- ...
    |   |-- ...
    |-- KL25Z gprof Shield
    |   |-- ...
    |   |-- ...
    |   |-- ...
    |-- Results
    |   |-- blinky
    |   |   |-- blink1cg.png
    |   |   |-- blink1cg.svg
    |   |   |-- blink1gmon.out
    |   |   |-- blink1gprof.txt
    |   |   |-- blink1hex.txt
    |   |   |-- blink2cg.png
    |   |   |-- blink2cg.svg
    |   |   |-- blink2gmon.out
    |   |   |-- blink2gprof.txt
    |   |   |-- blink2hex.txt
    |   |   |-- blink3cg.png
    |   |   |-- blink3cg.svg
    |   |   |-- blink3gmon.out
    |   |   |-- blink3gprof.txt
    |   |   |-- blink3hex.txt
    |   |   |-- blink4cg.png
    |   |   |-- blink4cg.svg
    |   |   |-- blink4gmon.out
    |   |   |-- blink4gprof.txt
    |   |   |-- blink4hex.txt
    |   |   |-- blinky.elf
```

```
|   |-- shield
|       |-- shield.elf
|       |-- shield1cg.png
|       |-- shield1cg.svg
|       |-- shield1gmon.out
|       |-- shield1gprof.txt
|       |-- shield1hex.txt
|       |-- shield2cg.png
|       |-- shield2cg.svg
|       |-- shield2gmon.out
|       |-- shield2gprof.txt
|       |-- shield2hex.txt
|       |-- shield3cg.png
|       |-- shield3cg.svg
|       |-- shield3gmon.out
|       |-- shield3gprof.txt
|       |-- shield3hex.txt
|       |-- shield4cg.png
|       |-- shield4cg.svg
|       |-- shield4gmon.out
|       |-- shield4gprof.txt
|       |-- shield4hex.txt
|-- Target MCU Application
    |-- gmon_arc.c
    |-- gmon_profil.c
    |-- gmon_profil.h
    |-- gmon_queue.c
    |-- gmon_queue.h
    |-- gmon_serial.c
    |-- gmon_serial.h
    |-- mcount.S
```