

Charbel Badr

Towards Time-Proportional Profiling of Low-Power System-on-Chips

Master's thesis in EMECS

Supervisor: Magnus Jahre

Co-supervisor: Jonathan Rico

September 2023

Charbel Badr

Towards Time-Proportional Profiling of Low-Power System-on-Chips

Master's thesis in EMECS
Supervisor: Magnus Jahre
Co-supervisor: Jonathan Rico
September 2023

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science



Towards Time-Proportional Profiling of Low-Power System-on-Chips

Charbel Badr

September 4, 2023

Assignment Text

Energy efficiency is a critical constraint in Ultra-Low-Power (ULP) systems. Improving efficiency however requires understanding what the system spends time on, and the standard approach for acquiring this understanding is to apply performance profiling.

In this thesis, the student should identify a suitable profiling approach, implement the code profilers, and use them to analyze the performance of the Zephyr host and communication protocol stack on the nRF52840 platform, thereby identifying performance-critical code regions, i.e., the parts of the system which have the largest impact on overall execution time. Based on this analysis, the student should propose optimizations and explain why these optimizations are expected to improve energy efficiency. If time permits, the student should implement and evaluate the proposed optimizations.

Abstract

System-on-Chips (SoC) energy efficiency is in higher demand nowadays due to the positive impact that it can have on various tech markets such as lower-cost products and better user experience. Code profiling through tracing is considered the most advantageous in the scope of this thesis as a tool to achieve energy efficiency. We implemented three different types of code profilers: the statistical profiler, the deterministic profiler, and the linear profiler. After comparing their performances, the deterministic profiler was found to perform the best and the closest to being time proportional. Additionally, these code profilers were also used in a case study regarding the Bluetooth stack implementation provided by Nordic Semiconductor. An inefficiency was found and was dealt with by marginally editing the code.

Acknowledgments

I would like to thank my supervisor at NTNU Prof. Magnus Jahre and my supervisor at Nordic Semiconductor Jonathan Rico for their continuous technical support along the journey of this thesis. The discussions we had about the topics at hand were valuable inputs to this master thesis. I would also like to thank my family and friends for their emotional support. Their encouragement, advice, and strong belief in me have been invaluable.

Contents

Assignment Text	iii
Abstract	v
Acknowledgments	vii
Contents	ix
Figures	xi
Tables	xiii
Listings	xv
Acronyms	xvii
1 Introduction	1
1.1 Assignment Interpretation	2
1.2 Contributions	3
1.3 Thesis Organization	4
2 Background Information	5
2.1 Performance Analysis Approaches	5
2.1.1 Statistical Sampling	5
2.1.2 Binary Instrumentation	6
2.1.3 Tracing	6
2.2 Time Proportionality in Profiling	6
2.3 Bluetooth Host and Controller	7
2.4 nRF52840 SoC	8
3 Implementation	11
3.1 Tracing nRF52840 with JTrace	11
3.1.1 Debug and Trace Block	11
3.1.2 JLink SDK	12
3.1.3 Instruction Pipelining and Hazards	13
3.2 Building the Code Profilers	14
3.2.1 Statistical Profiler	14
3.2.2 Deterministic Profiler	15
3.2.3 Linear Profiler	17
3.2.4 Instruction to Function Mapping	18
3.2.5 Code Profile Formatting	18
4 Experimental Setup	21
4.1 MachSuite	21
4.1.1 FFT Strided	21

4.1.2	GEMM Blocked	22
4.2	Throughput Sample	22
5	Results	25
5.1	MachSuite Algorithms Profile	25
5.2	The Bluetooth Stack Profile	26
6	Conclusion	33
6.1	Conclusion	33
6.2	Limitations	33
6.3	Future Work	34
	Bibliography	35
A	Appendix	39

Figures

1.1	Energy Efficiency on Multiple Levels [1]	2
1.2	Hierarchy of Profiling	3
2.1	Assigning Cycles Based on NCI, LCI, and TIP [17]	7
2.2	Bluetooth Stack [19]	8
2.3	nRF52840 Block Diagram	9
3.1	Design Block Diagram	12
3.2	nRF52840 with the 20 Pin Interface Soldered.	13
3.3	Debug and Trace Module [23]	14
3.4	DT Profile Flowchart	15
3.5	Assembly Instruction Pie Chart in a Sample	17
4.1	Throughput Sample Flowchart	23
5.1	Cycle Distribution across Instruction Types in the FFT Benchmark .	26
5.2	Cycle Distribution across Instruction Types in the GEMM Benchmark	27
5.3	For Loop v.s While Loop Graph - Statistical Profile	29
5.4	For Loop v.s While Loop Graph - Deterministic Profile	30
5.5	For Loop v.s While Loop Graph - Linear Profile	30
5.6	Optimized Code Cycle Attribution	31

Tables

A.1	Statistical Profile - For Loop v.s While Loop	39
A.2	Deterministic Profile - For Loop v.s While Loop	39
A.3	Linear Profile - For Loop v.s While Loop	40

Listings

3.1	Pperf Tool Output	18
3.2	Sample Code Profile	19
5.1	Statistical Profile - While Loop	27
5.2	Deterministic Profile - While Loop	27
5.3	Linear Profile - While Loop	28
5.4	While Loop memcpy Implementation	28
5.5	For Loop memcpy Implementation	28
5.6	Statistical Profile - For Loop	28
5.7	Linear Profile - For Loop	28
5.8	Deterministic Profile - For Loop	29
5.9	Proposed Optimized Code	31

Acronyms

AHB	Advanced High performance Bus.	9
BLE	Bluetooth Low Energy.	2, 7, 8, 22
CPU	Central Processing Unit.	1, 5, 6, 9
DWT	Data Watchpoint and Trace.	12
ETM	Embedded Trace Macrocell.	12
FICR	Factory Information Configuration Registers.	9
FIFO	First In First Out.	6
FPB	Flash Patch and Breakpoint.	11
GPIO	General Purpose Input/Output.	2
ITM	Instrumentation Trace Macrocell.	12
JTAG	Joint Test Action Group.	5, 11
LCI	Last Committed Instruction.	6
NCI	Next Committing Instruction.	6
NVMC	Non-Volatile Memory Controller.	9
OS	Operating System.	1
PC	Program Counter.	5, 6
PMU	Performance Monitoring Unit.	5
RAM	Random Access Memory.	2

RTL Register Transfer Language. 1

RTOS Real Time Operating System. 26

SoC System on Chip. v, 1, 2, 4, 6, 8, 9, 13, 33

SWD Serial Wire Debug. 5, 11

TPIU Trace Port Interface Unit. 12

UICR User Information Configuration Registers. 9

Chapter 1

Introduction

The need for energy efficiency in SoCs has been on the rise due to the positive impact it can have on many big markets such as aerospace, mobile, and artificial intelligence applications. An achieved energy efficiency could be reflected in reduced product costs and better user experience. Energy efficiency can be dealt with on several levels. These include software level, OS/firmware level, architecture level, RTL design level, implementation level, and process technology level [1]. Figure 1.1 shows the strategies that each level has in order to make the SoC more energy efficient. For the scope of this thesis, we focus on the software level. To be able to achieve power-aware apps, we require the necessary tools that should be quite reliable in detecting inefficiencies. In other words, to be able to come across optimizing a code at hand, robust code profilers are a must, and implementing them will be one of the main contributions of this thesis.

Code profiling is the process of gathering parameters such as function calls, CPU cycles, and other metrics on a sample code while it is running [2]. In order to build a code profile, tracking instructions in a program is necessary. Analysis techniques mainly fall under 3 branches: statistical sampling, binary instrumentation, and tracing. Figure 1.2 summarizes the hierarchy of profiling.

1. **Statistical Sampling:** can be intrusive or non-intrusive. It is an approach that focuses on retrieving the executed instructions by periodically sampling the program counter. If it halts the CPU in the process, then it is intrusive. Gperftools [3], Pperf [4] and Pperf [5] are three examples of intrusive sampling applications. On the other hand, Lynsynn [6] is an example of a non-intrusive sampler.
2. **Binary Instrumentation:** is a software-based approach that focuses on editing the binary of a program to capture useful information about program flow and executed functions. This could either be static or dynamic depending on when the binary file editing happens. If it happens before the program starts, the instrumentation is static while if the editing happens during the running of the program, the instrumentation is classified as dynamic. Pin [7] and DynamoRio [8] are examples of dynamic instrumentation while Clang [9] is an example of static instrumentation.

Stage	Strategy
Software Level	Power Aware Apps
OS/Firmware Level	Power Aware OS, Firmware
Architecture Level	Power Management, DVFS
	System Components
RTL (Design) Level	Micro-Architecture
	Power Efficient RTL
Implementation Level	Logic Level
	Transistor Level
	Layout Level
Process Technology Level	Libraries, Macros
	Devices/Transistors
	Materials




Figure 1.1: Energy Efficiency on Multiple Levels [1]

- Tracing: is a hardware-based approach that works based on storing instructions either in a designated area in the RAM of the embedded device or streaming them directly to GPIO pins. Ninja [10] is an example of an application that uses tracing as a method of profiling.

We also include a case study where we put the implemented code profilers in action in to attempt to optimize the Bluetooth implementation on Nordic Semiconductor SoC. Bluetooth low energy (BLE) is an evolving technology that is found in many electronics that are used daily such as wireless speakers, wearables, and in-car entertainment systems [11]. Bluetooth was continuously being enhanced to include improved connectivity and security. Nordic Semiconductor is considered to be one of the leading companies working closely with Bluetooth [12]. By 2022, Nordic Semiconductor was shipping 1 million Bluetooth LE SoCs every day and they constitute 39% of the market share. This means that optimizing the energy consumption of the Bluetooth implementation provided by Nordic Semiconductor could have a significant positive impact on the market.

1.1 Assignment Interpretation

Based on what has been discussed, the list of tasks in this thesis should include the following:

- Task 1: Identify a method for instruction tracing required for code profiling

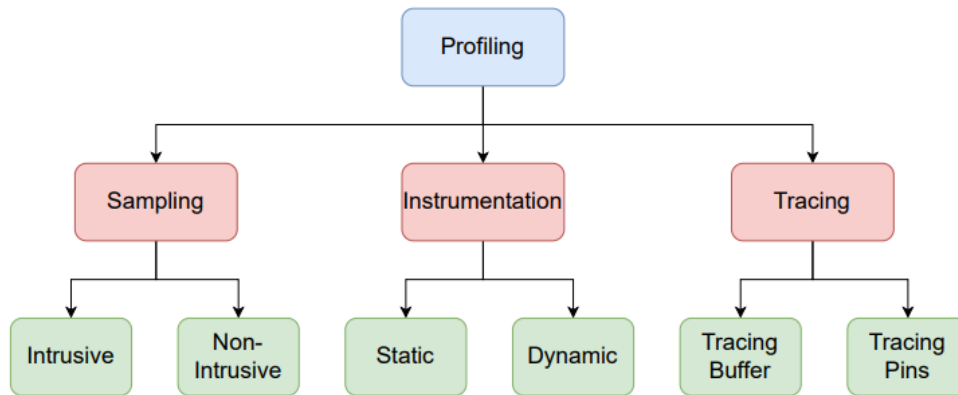


Figure 1.2: Hierarchy of Profiling

- Task 2: Based on the trace on hand, build different code profilers.
- Task 3: Attach the code profiler/s to a sample code provided by Nordic Semiconductor regarding the data path of the Bluetooth stack including both the host and controller and analyze them
- Task 4: Provide optimization techniques based on the analysis of code profilers.

Task 1 answers which profiling technique we chose for the scope of the thesis. This comes after studying the different analysis approaches and choosing the one with the most advantages and minimal disadvantages to the task at hand. Task 2 makes the process of analysis quite easier as a mere instruction trace without a code profile is much harder to analyze and understand. If multiple code profilers are implemented, then it is quite necessary that we compare their performances. Finally, Tasks 3 and 4 are in response to the analysis and optimization requirements found in the assignment text.

1.2 Contributions

My contributions after working on this master thesis include the following:

- Contribution 1: Understanding how Segger's JTrace [13] works as a tool for instruction tracing, see Section 3.1.
- Contribution 2: Building three types of code profilers "on the go" while getting the stream of instruction trace. This eliminates the need to wait for the whole instruction stream before processing it, see Section 3.2.
- Contribution 3: Identifying inefficiencies in the snippet of the code involving Bluetooth datapath after attaching the code profilers to it, see Section 5.2.
- Contribution 4: Providing an optimization technique that causes a speedup

in Bluetooth code, see Section 5.2.

Mainly, the contributions have a one-to-one relationship with the tasks mentioned above. Each contribution in this section comes to answer a task in the previous one.

1.3 Thesis Organization

The rest of the thesis will be organized as follows:

- Chapter 2 includes background information needed to understand the methods proposed specifically profiling techniques, time proportionality, nRF52840 SoC, and Bluetooth host and controller.
- Chapter 3 includes the design methodology and the implementation of the proposed solution. It focuses on understanding the method of instruction tracing using JTrace [13] and the process of implementing the code profilers.
- Chapter 4 includes the experimental setup of putting the code profilers in action. There are mainly two experiments, one to understand and analyze the behavior of the designed code profilers, and the second aims to improve the Bluetooth implementation provided by Nordic Semiconductor.
- Chapter 5 includes the results of the experiments performed and shows tables, figures, and statistics of the output. We also analyze the results in the light of theoretical concepts and previous work.
- Lastly, chapter 6 includes a conclusion of what has been done in this thesis and also sheds light on limitations and possible future work.

Chapter 2

Background Information

In this section, some technical concepts will be briefly explained as they are necessary to the understanding of the proposed implementation.

2.1 Performance Analysis Approaches

2.1.1 Statistical Sampling

Intrusive sampling primarily works by sampling PC values or other metrics using interrupts. The sampling program is usually run on the same host device as the program being sampled. One example here could be Gperftools [3] which functions on the basis of linking the library to the application code binary and running it. It samples at a rate of 100 samples per second. Another application that is considered an intrusive profiler is Pperf [4]. It mainly profiles ELF binary files and samples for internal and external PMU data including CPU and PC data for every thread. Similarly, the Linux kernel-based tool Perf [5] is an intrusive profiler that gathers PMU data. This tool can also keep track of threads' call stack issued from time interrupts, system calls, and other system events. The advantages of these tools are relatively smaller overheads and no additional hardware is required. On the other hand, one of the biggest disadvantages is the fact that they interrupt the CPU, which causes real-time applications to be greatly affected. Another drawback could be the inaccuracy of the profiler due to the sampling bias. In other words, parts of the code could be oversampled or undersampled.

Non-intrusive sampling works primarily without causing interruption to the application being sampled. This requires the sampler to be present on a different host than that of the application running. The transfer of sampling happens through protocols such as JTAG or SWD. Lynsyn [6] is an example of a non-intrusive profiler that gathers PC samples over JTAG. It can gather up to ten thousand measurements per second. However, it is quite important to mention that Lynsyn is not tracing but rather sampling the application which means it will miss tracking quite some instructions.

2.1.2 Binary Instrumentation

Binary instrumentation is the process of adding to or changing the binary of an application program in order to obtain metrics such as function execution count, code coverage, and other metrics. The process could either be static or dynamic. This being said, static binary instrumentation is changing the binary before the runtime while dynamic binary instrumentation usually involves changing the binaries during the run-time of the program. Pin [7] and DynamoRio [8] are tools based on dynamic binary instrumentation. Pin is built as an architecture-independent tool but could use architecture-specific optimizations when needed. Additionally, Pin uses inlining, liveness analysis, and other techniques to optimize the instrumentation. DynamoRio provides code caching techniques to handle complex code instrumentation and provides an overhead of zero to thirty percent on operating systems such as Linux and Windows. On the other hand, Clang instrumentation [9] is considered to be a static binary instrumentation tool. It provides code coverage, and security analysis and is compatible across languages such as C, C++, and Objective C. In general, static instrumentation tools have less overhead than dynamic ones but usually offer a more limited analysis because some scenarios only occur during runtime.

2.1.3 Tracing

Tracing buffers and pins keep track of every instruction and either save it in a buffer in memory or stream it out to another device through the tracing pins. The data in the buffer is overwritten based on a FIFO protocol. One way of accessing the tracing buffers is by regular JLink devices. The tracing pins, on the other hand, require an embedded tracing macrocell (ETM) and specified hardware such as JTrace to function properly [14]. This method can reach speeds of more than 100 MiB/s. Another application based on tracing is NINJA [10] which offers a malware analysis framework that traces applications and safely debugs them by using the ETM and the performance monitor units. This framework is independent of the operating system of the SoC and has low overheads. The main advantage of these methods is increased accuracy. However, this comes at the expense of increased hardware costs.

2.2 Time Proportionality in Profiling

In all of the approaches mentioned above, one important aspect to consider is the method by which the CPU cycles are being attributed to instructions at respective PC values and how time-proportional they are. Some of these methods include Next Committing Instruction (NCI) [15], Last Committed Instruction [6] (LCI), dispatch tagging heuristic [16], and Oracle method [17, 18]. Both, the next committing instruction and the last committed instruction approaches are self-explanatory, in the sense that they assign the cycles to the next and last com-

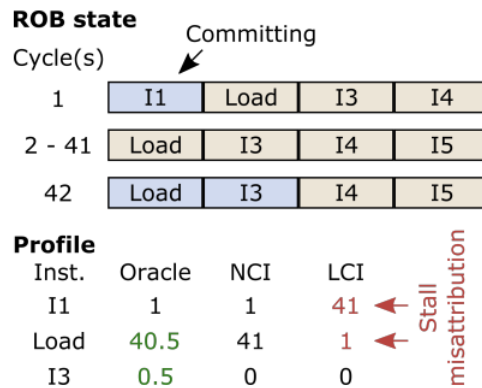


Figure 2.1: Assigning Cycles Based on NCI, LCI, and TIP [17]

mitted instructions respectively. The dispatch tagging assigns the cycles to the instruction fetched instead of the committed one. According to the findings of [17], the mentioned approaches are not time-proportional. However, they proposed a time-proportional instruction profiling approach, the Oracle method, by studying the edge cases and formulating a state diagram whereby they assign the cycles to instructions based on an informed decision. Fig. 2.1 shows the case when a load instruction stalls the pipeline. NCI and LCI blindly assign the cycles to the Load and I1 instructions respectively. The oracle profiler realizes that this is in the stalled state and assigns the cycles to the instruction causing the stall, that is the load.

2.3 Bluetooth Host and Controller

The Bluetooth stack is mainly divided into the Bluetooth host and the controller [19]. Figure 2.2 shows the protocols and layers inside each of the host and the controller.

In the Host, the layers include:

1. GAP: Generic Access Protocol. This layer is the get-way between the application layer and the Bluetooth stack. It defines the procedures and roles and manages setting up connections.
2. GATT: Generic Attribute Profile. It is a special case of the Attribute protocol. It defines mainly two roles which include the client and the server. This layer also defines the data structure of which the data and information are being exchanged through BLE.
3. ATT: Attribute Protocol. This is the more general protocol that GATT is built around. It defines the client-server architecture and organizes the data into

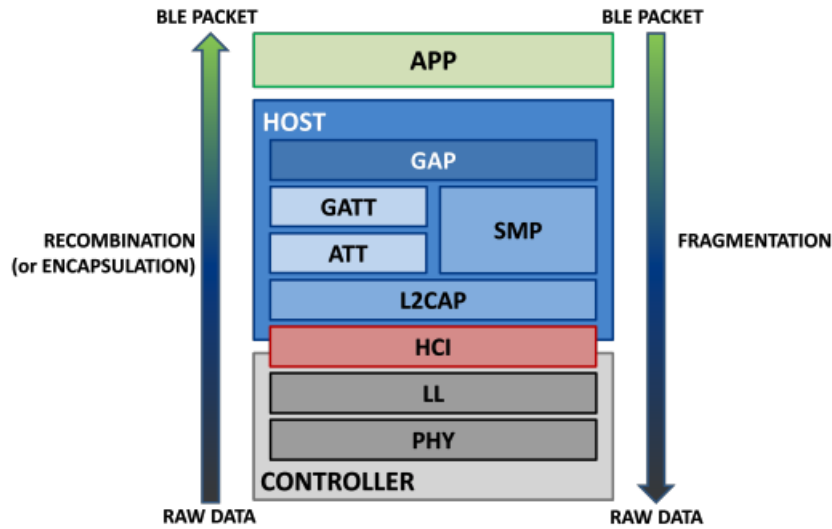


Figure 2.2: Bluetooth Stack [19]

attributes with unique identifiers.

4. SMP: Security Manager Protocol. This layer is responsible for the security of BLE transfers. It is composed of multiple security algorithms aimed at the encryption and decryption of BLE.
5. L2CAP: Logical Link Control and Adaptation Protocol. This layer primarily functions as a multiplexer. It receives data from lower layers into upper layers and vice versa.
6. HCI: Host Controller Interface. This layer is shared with the controller side and the main purpose of it is that it manages the interactions between the host and controller.

On the other hand, the layers on Controller side include:

1. HCI: This layer is shared with the host as described previously.
2. Link Layer: This layer mainly communicates with the physical layer and is responsible for Cyclic Redundancy Check (CRC) generation, advanced encryption standards (AES), and data whitening.
3. Physical Layer: This layer defines how packets are sent over the radio. It defines the hopping scheme and the transmit power.

2.4 nRF52840 SoC

This version of SoC is considered to be the most advanced in the nRF52 series [20]. It allows for complex concurrency protocols to run and accommodates a lot of peripherals and features. It supports Bluetooth Low Energy BLE, Bluetooth mesh, Thread, Zigbee, 802.15.4, ANT, and 2.4 GHz proprietary stacks. This chip

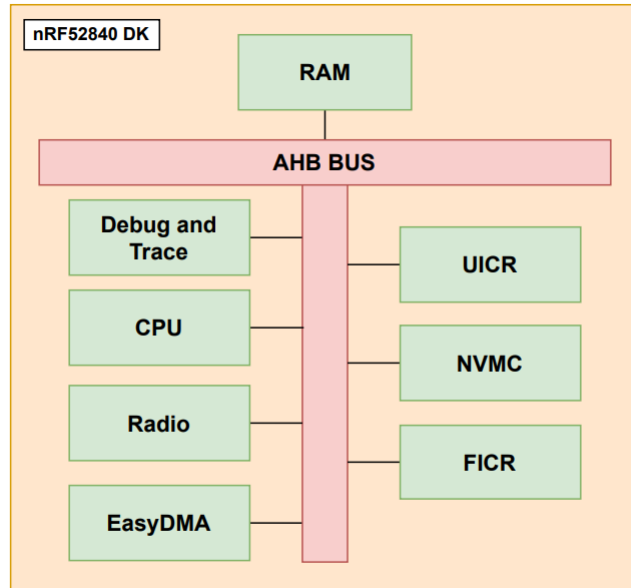


Figure 2.3: nRF52840 Block Diagram

is built with an ARM Cortex M4 CPU.

The nRF52840 development kit which is built on top of this SoC can also accommodate a real-time operating system, which is the Zephyr RTOS. Zephyr RTOS is designed for devices with constrained resources and supports multiple architectures [21].

Figure 2.3 shows the block diagram with the core components alongside the radio module of an nRF52840 development kit. The core components include a main memory, CPU, Non-Volatile Memory Controller (NVMC), Factory Information Configuration Registers (FICR), User Information Configuration Registers (UICR), EasyDMA (Direct Memory Access), Advanced High-performance Bus (AHB) and debug and trace module

Chapter 3

Implementation

The implementation phase can be best split into two sections: understanding the tracing mechanism using JTrace [13] and building the code profilers necessary to understand and analyze the performance of applications. The block diagram in Fig 3.1 represents the implementation as a whole. First, the instruction trace is obtained. Then, after that, three different profilers are created: the Statistical (ST) Profiler, the Deterministic (DT) Profiler, and the Linear (LI) Profiler. However, these generate assembly-level instruction profiles. Thus, we used a subsystem in Pperf [4] that links the assembly instructions to their respective higher-level functions. Finally, these function-level profiles were inputted into a formatter in order to make them more readable and easier for analysis.

3.1 Tracing nRF52840 with JTrace

The first step here is to solder the 20-pin trace interface on the board. Then, the JTrace is connected to both the nRF52840 through a ribbon cable and the host computer through USB. Figure 3.2 shows the pin trace interface that was soldered to the board and the JTrace connected to it. On the host computer, JLink SDK [22] was used to interface with the JTrace through Python scripts.

3.1.1 Debug and Trace Block

The trace and debug module consists of multiple blocks [23]. Figure 3.3 shows the different blocks inside the debug and trace module. The main specifications of this block include:

1. Serial Wire Debug (SWD): is an interface and debugging protocol that could be described as an enhancement and a more compact form than JTAG as it requires fewer pins. It allows memory operations as well as setting breakpoints.
2. Flash Patch and Breakpoint (FPB) unit: This unit allows changing the code temporarily when debugging without changing the original code and allows

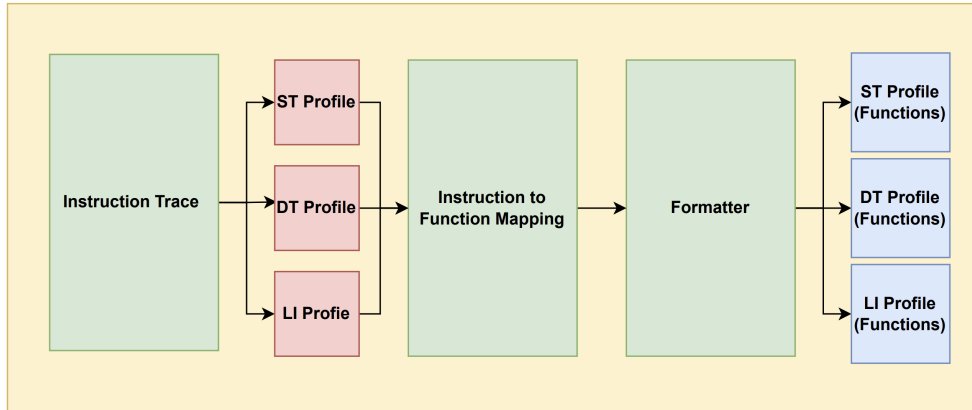


Figure 3.1: Design Block Diagram

setting breakpoints to the program for further debugging. This unit has two literal comparators along with six instruction comparators.

3. Data Watchpoint and Trace (DWT) unit: This unit allows multiple functionalities in the domain of debugging. These include memory monitoring, cycle counting, timestamp generation, and performance analysis. Additionally, this unit has four comparators.
4. Instrumentation Trace Macrocell (ITM): This unit mainly revolves about instrumentation profiling. It allows for real-time debugging and it encapsulates the trace packets and the timestamps generated from the (DWT).
5. Embedded Trace Macrocell (ETM): This unit is responsible for many functionalities such as performance profiling and complex system debugging but the most important functionality is the instruction tracing that it streams to external devices.
6. Trace Port Interface Unit (TPIU): This unit multiplexes tracing information from the two units, the ETM and ITM. It efficiently transmits this information across the trace lines. In other words, all tracing information passes through the (TPIU)

3.1.2 JLink SDK

The JLink SDK [22] was used in order to obtain the instruction trace needed to build the code profilers. Using this SDK, we were able to extract the assembly instruction trace from an input initial address to a breaking point address. Timestamps were also obtained by the SDK but not for every individual instruction. Instead, timestamps were issued every n cycles, where n is a power of 2, and associated with the assembly instruction at hand. In other words, the output of using the Jlink SDK was an assembly instruction trace, some of which had an associated timestamp generated from the ITM.



Figure 3.2: nRF52840 with the 20 Pin Interface Soldered.

3.1.3 Instruction Pipelining and Hazards

Instruction pipelining is the process of dividing the execution of instructions into multiple stages that work in parallel with the aim of achieving a certain speedup [24]. The usual RISC processor is divided into five stages which include: instruction fetch, instruction decode, instruction execute, memory access, and write back.

On the cortex-M4 based Nordic SoC nRF52840, there exist only three stages which include [25]:

- The fetch stage
- The decode stage
- The execute stage

The load use hazard is considered to be a true dependency issue in pipelining. The load use hazard arises when the load instruction is still pending when an instruction at a later stage in the pipeline requests the data being loaded. Due to the simplicity of the Arm Cortex M4 pipeline, the load use hazard is not an issue as the execution of the load happens in the execute stage and there are no further stages in the pipeline. Hence, a load instruction will simply just stall the pipeline until it is done. This information will come in handy when assigning cycles to instructions.

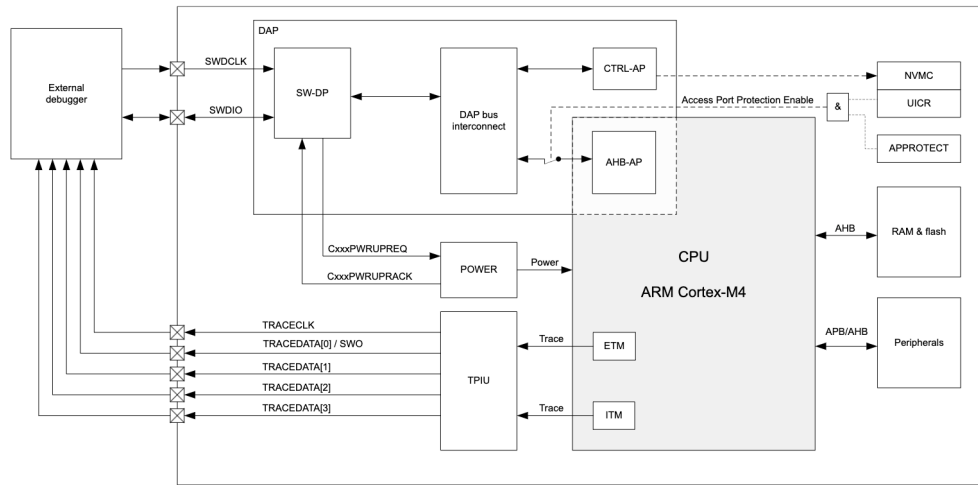


Figure 3.3: Debug and Trace Module [23]

3.2 Building the Code Profilers

The trace that was being recorded from an initial address to the breaking point address follows this specific format:

```

Assembly Instruction 1 - Cycles: X
Assembly Instruction 2
.
.
.
Assembly Instruction N
Assembly Instruction N + 1 - Cycles: Y

```

such that:

$$Y - X = 2^n \text{ where } n \text{ is set by the user} \quad (3.1)$$

While receiving this trace using JLink SDK, three different profilers were created namely the Statistical Profiler, the Deterministic Profiler, and the Linear Profiler. The format above will be used in the explanation of how the profilers were created.

3.2.1 Statistical Profiler

This code profile sheds importance only on the instructions that have timestamps associated with them. In other words, this profiler works based on sampling the trace every 2^n cycles.

The steps for creating this profile include:

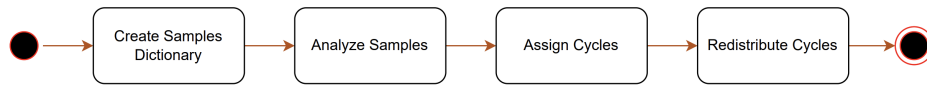


Figure 3.4: DT Profile Flowchart

1. If an instruction has an associated cycle timestamp, save the instruction as a key in a dictionary with whole 2^n cycles assigned to this instruction.
2. If this instruction appears again in the scope of the trace, edit the value field to accumulate the extra 2^n cycles.
3. Lastly, assign a percentage to each instruction based on the following equation:

$$\frac{\text{Total Cycles Associated to the Instruction}}{\text{Total Cycles of All Instructions in the Dictionary}} \quad (3.2)$$

3.2.2 Deterministic Profiler

This type of profiler attempts to assign every assembly instruction the number of cycles it takes based on the analysis of the Cortex M4 instruction set summary [26]. The implementation of the deterministic profiler is quite more complex than the other profiles. Fig. 3.4 shows a flow chart of how the deterministic profile is generated. Each process in this flow chart will be explained in the following paragraphs.

Creating Samples Dictionary: A sample here refers to the list of assembly instructions between two generated cycle timestamps. As per the example format given above, a sample could be a list that begins from Assembly Instruction 1 to Assemble Instruction N. The total number of cycles that the instructions in a sample take is 2^n . The dictionary will thus have the following format:

```

Sample 1    [ (Address 1, Instruction 1), ... , (Address N, Instruction N) ]
Sample 2    [ Address N+1, Instruction N+1), ... ,(Address Z, Instruction Z) ]
  
```

Analyzing the samples: After looking at the Cortex M4 instruction set summary we realize that the number of cycles could be static or might be varying. The variations in the manual are represented by "P" and "N". The symbol "P" is the number of cycles for pipeline refill and could be either one, two, or three cycles. As an average, two cycles will be used in instructions that have the "P" symbol associated with them. The second symbol ("N") represents the number of registers in the register list to be stored or loaded. Other symbols include "B" and "W" which mean the number of cycles for a barrier operation and the number of cycles waiting for an event respectively. However, they are not common and are

thus not taken into account for the scope of this thesis. Another aspect to consider is the special cases mentioned in the load store timing manual [27]. The main important cases include:

1. The store instruction $STR\ Rx, [Ry, \#imm]$ always takes 1 cycle.
2. $LDR [any]$ will be pipelined when possible. Thus, if an LDR is followed by LDR or STR, the whole LDR-LDR or LDR-STR pairs will only take three cycles

Assigning the cycles: Taking into account what has been discussed earlier, a series of if-else statements were formulated to assign the cycles to respective instructions. Regarding instructions with the "N" symbol, the register list size was counted using regular expressions.

Redistributing the Cycles: After distributing the cycles across the instructions in a sample, their sum could either be slightly higher or lower than the 2^n cycles. This is possible due to the assumptions and averages being taken alongside the real-time behavior effect. Another reason might be due to the fact that load instructions stall the pipeline until the transaction is completed. Also, statistical analysis shows that the samples that have a deviation from the 2^n cycles have a predominant number of load instructions. Figure 3.5 shows that the number of LDR instructions is the highest in a sample that encounters this deviation. This makes it a valid reason or a good approximation if we redistribute the extra cycles by either assigning more cycles to LDRs if the sum is higher than 2^n cycles or taking cycles from the LDRs if the sum is lower than 2^n cycles.

The process of redistribution happens by first getting the load instruction count and calculating two metrics which are the *Cycles-per-Instruction* (CPI) and *Remainder-Cycles-per-Instruction* (RCPI). These metrics are calculated using the following equations:

$$CPI = \left\lfloor \frac{Extra\ Cycles}{LDR\ Count} \right\rfloor \quad (3.3)$$

$$RCPI = Extra\ Cycles \% LDR\ Count \quad (3.4)$$

Then, each LDR instruction is revisited through looping, and the following equations are performed:

$$LDR\ Cycles = Previous\ Cycles \pm CPI \pm RCPI \quad (3.5)$$

$$RCPI = RCPI - 1 \quad (3.6)$$

The plus-minus operation is dependent on whether the LDR instructions' cycles are being increased or decreased.

After that, the process is quite straightforward to build the profile by mapping every instruction with the accumulation of cycles obtained from different samples. A dictionary was used with the key being the instruction address and the value being the aggregation of cycles.

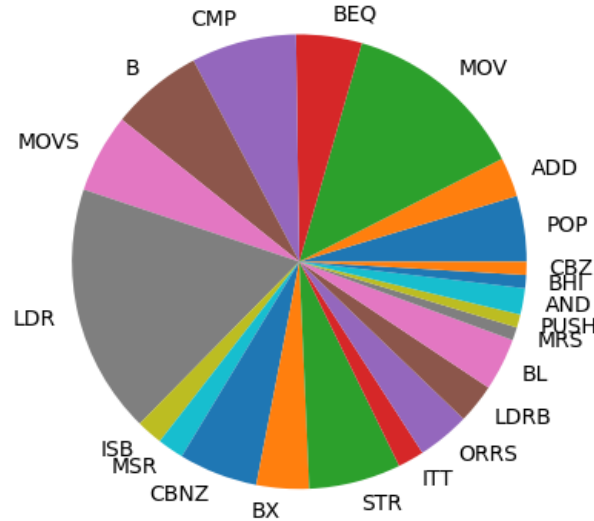


Figure 3.5: Assembly Instruction Pie Chart in a Sample

3.2.3 Linear Profiler

The linear profiler works in a similar manner as the deterministic profile but differs from it by the method of assigning cycles to instructions. This method equally distributes the 2^n cycles of a sample across the assembly instructions. The method of doing this is quite similar to the redistribution process in the deterministic profiler by calculating two metrics which are the *Cycles-per-Instruction* (CPI) and *Remainder-Cycles-per-Instruction* (RCPI). Here the equations slightly differ.

$$CPI = \left\lfloor \frac{2^n \text{ Cycles}}{\text{Instructions Count}} \right\rfloor \quad (3.7)$$

$$RCPI = 2^n \text{ Cycles} \% \text{ Instructions Count} \quad (3.8)$$

Then, every instruction is visited and assigned cycles based on the following equations:

$$\text{Instruction Cycles} = CPI + RCPI \quad (3.9)$$

$$RCPI = RCPI - 1 \quad (3.10)$$

Building the code profile after that requires using a dictionary to store (instruction address, accumulation of cycles) pairs.

3.2.4 Instruction to Function Mapping

After building the three types of code profilers, the output obtained is a list of assembly instructions' addresses and the aggregation of cycles across the application program. Thus, we require mapping the instructions to the higher-level functions in which they are enclosed. To do this, we use a subsystem in Pperf [4]. It works by creating a cache to a binary file and takes as an input the code profiles created above and correlates the assembly instructions' addresses to their functions. This tool gives the following information about every assembly instruction in the code profiles:

- Binary file
- Basic Block
- Parent Function
- Line Number
- Assembly Instruction

This tool was quite straightforward to use but a minor edit was performed which was changing the code in *profileLib.py* at line 259 from "objdump" to "arm-linux-gnueabi-objdump" because the binary ELF files obtained from the nRF52840 are elf32-little which the command "objdump" does not support. A sample output of using this tool is shown below in Listing 3.1

Listing 3.1: Pperf Tool Output

```
pc;binary;function;basicblock;line;instruction;asm;values
0x0003B492;zephyr.elf;process_queue;f1958;378;pop;pop {r4, r5, r6, pc};7
0x0003B490;zephyr.elf;process_queue;f1958;378;mov;mov r0, r4;1
0x0003B484;zephyr.elf;process_queue;f1958;368;mov;mov r4, r0;1
0x00029816;zephyr.elf;chan_send;f1241;279;add;add sp, #12;1
```

3.2.5 Code Profile Formatting

The last step to creating understandable code profiles that are easier to analyze is formatting the output obtained from the Pperf tool. This was done by parsing the CSV files obtained and sorting the functions and their assembly instruction in decreasing order of cycle counts. The only information extracted for every instruction are the address, line number, assembly code, and cycle count. All three code profiles have the same format but differ only in how the cycles are distributed among instructions. A sample code profile is shown below in Listing 3.2.

Listing 3.2: Sample Code Profile

Function: net_buf_simple_tailroom Total Cycles: 81			
Address	Line	Assembly	Cycle Count
0x0003CEFE	1251	ldrh r3, [r0, #4]	16
0x0003CF06	1251	ldr r3, [r0, #0]	15
0x0003CF02	1251	ldr r1, [r0, #8]	15
0x0003CF00	1251	ldrh r2, [r0, #6]	15
0x0003CF0C	1252	bx lr	8
0x0003CF0A	1252	subs r0, r2, r3	4
0x0003CF08	1251	subs r3, r3, r1	4
0x0003CF04	1251	subs r2, r2, r3	4
Function: net_buf_simple_headroom Total Cycles: 12			
Address	Line	Assembly	Cycle Count
0x0003CEF8	1246	ldr r0, [r0, #8]	5
0x0003CEF6	1246	ldr r2, [r0, #0]	4
0x0003CEFC	1247	bx lr	2
0x0003CEFA	1247	subs r0, r2, r0	1

Chapter 4

Experimental Setup

This chapter focuses on putting the designed code profilers in action first to compare their performances and second to give an insight into the running application programs. Before attaching the code profilers to the Bluetooth-related code, we analyze their behavior by attaching them to some samples of the MachSuite benchmark [28]. Then, we trace the Bluetooth-related code that is found in the Throughput Sample offered by Nordic Semiconductor [29].

4.1 MachSuite

MachSuite is a benchmark of 19 algorithms that imitate low-level kernels for hardware acceleration. This benchmark was designed to enable standardization and created an easier path to access the research of hardware accelerators [28]. For the sake of this thesis, we decided to use two algorithms from this benchmark to test our code profilers and analyze their behavior in tracing instructions and assigning CPU cycles to them. The main reason for using these algorithms from MachSuite is their simplicity and ease of analysis even without requiring a code profile. Thus, the process of tracing with the implemented code profilers becomes quite verifiable. However, the main problem was that these benchmarks took a file-based input. To be able to do this on the nRF52840 Nordic development kit, either the algorithms should be greatly edited to make it compatible with the special file system on the nRF52840 or the data should be transferred from input files into variables in the code programs. The latter was chosen mainly due to simplicity and the main goal was not to optimize the algorithms in the Machsuite but to test the implemented code profilers. The two algorithms include FFT-strided and GEMM-blocked.

4.1.1 FFT Strided

This algorithm calculates the fast Fourier transform in an iterative manner. The reason for choosing this algorithm is because of the strided accesses that are found

in the implementation and this helps with further analyzing the load store behavior.

4.1.2 GEMM Blocked

This algorithm is a matrix multiply implementation. The importance of using this algorithm lies in seeing memory locality in action alongside some arithmetic operations in progress. The algorithm uses a blocking factor of 8 and is inspired by the proposal of the algorithm in [30]

4.2 Throughput Sample

The throughput sample, as the name suggests, is a sample application developed by Nordic Semiconductor to test the Bluetooth Low Energy throughput performance. It makes use of the GATT Throughput service. The application shows the interaction and performance shift of the throughput as a function of the following connection parameters

- ATT_MTU: This represents the maximum transmission unit. Increasing this will allow for bigger payloads and hence better throughput.
- Data Length: Increasing this has a similar effect as increasing the ATT_MTU as it allows for more data to be transferred in one packet
- Connection Interval: Increasing this metric may increase the throughput as more packets can be sent in one interval. However, if a packet is lost, the re-transmission becomes longer.
- Physical Layer (PHY) Data Rate: Increasing this value will make the transmissions faster.

Fig. 4.1 shows a flowchart of the throughput sample. The process starts with initialization and the user decides the roles of the two Nordic boards (nRF52840) participating in this throughput sample. After the roles are decided, a connection happens and the devices are paired. If the role of a device is peripheral, then a server is created and the peripheral device waits for the central device to initiate contact. If the device is a central device, a server discovery process starts, and then the MTU parameter is exchanged with the peripheral device. Afterwards, the user is given the choice to edit the connection parameters or run the throughput sample. Once the user decides to run the sample, the data is transferred from the central device to the peripheral device and then read back again from the peripheral device. Finally, the statistics in terms of timing and the total amount of data transferred are displayed.

Although aimed to experiment and measure the throughput of BLE, this application is a good sample to visit the Bluetooth stack including the host and controller. However, this sample has two main issues to deal with before tracing it using the implemented code profilers.

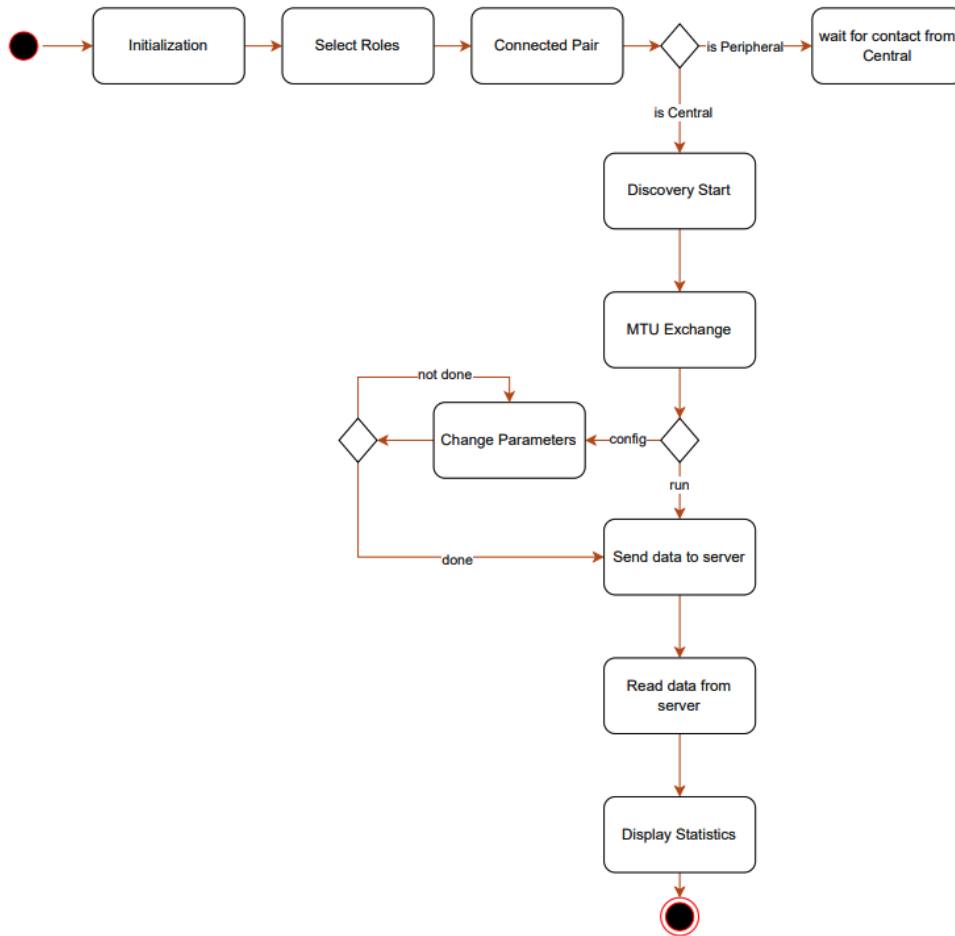


Figure 4.1: Throughput Sample Flowchart

1. Print statements: Print statements trigger a series of functions in the background which would be of no benefit to the overall profile. The throughput sample was thus revisited and the printing statements were eliminated from the code program.
2. Automation: The throughput sample depends a lot on user input such as setting the roles to central or peripheral. Another example would be the user choosing whether to change the parameters or run the program. The problem with user input is that tracing becomes much harder as the program is halted until receiving feedback from the user. Thus, the process was automated to choose the roles and run the program without changing the connection parameters as the important thing here is to see the Bluetooth stack in action and try to optimize that.

Chapter 5

Results

The results section can be mainly divided into two sections: the MachSuite algorithm profiles and the Bluetooth stack profile.

5.1 MachSuite Algorithms Profile

After tracing the two programs, the FFT benchmark and the GEMM benchmark, and obtaining the code profiles, the assigning mechanism of cycles to instructions was further analyzed to compare the performances of the code profilers. Mainly, the instructions were divided into three types:

1. Data Processing Instructions: These include arithmetic, logic, and other data manipulation instructions.
2. Data Movement Instructions: These mainly include moving data from or to the main memory. In other words, they are the load and store instructions.
3. Control Flow Instructions: These include the instructions that decide the control flow of the program such as branching instructions.

Figures 5.1 and 5.2 show the distribution of cycles in the different types of profiles across the different types of instructions. In the statistical (ST) profile of the FFT benchmark, 61% of the cycles were attributed to the data processing instructions, 24% of the cycles were attributed to the data movement instructions, and 15% of the cycles were attributed to the flow control instructions. The deterministic (DT) profile follows a quite similar trend with 54% of the cycles given to data processing instructions, 27% of the cycles given to data movement instructions, and 19% of the cycles given to the flow control instructions. However, the greatest change is noticed in the linear (LI) profile where data processing instructions received more than 75% in cycle attribution, and the data movement instruction received less cycle attribution (9%) than that of the flow control instructions (14%). As for the GEMM benchmark, the cycle attribution distribution followed the same trend as the FFT benchmark with minor changes in percentages across different types of instructions. The respective percentages are shown in Figure 5.2.

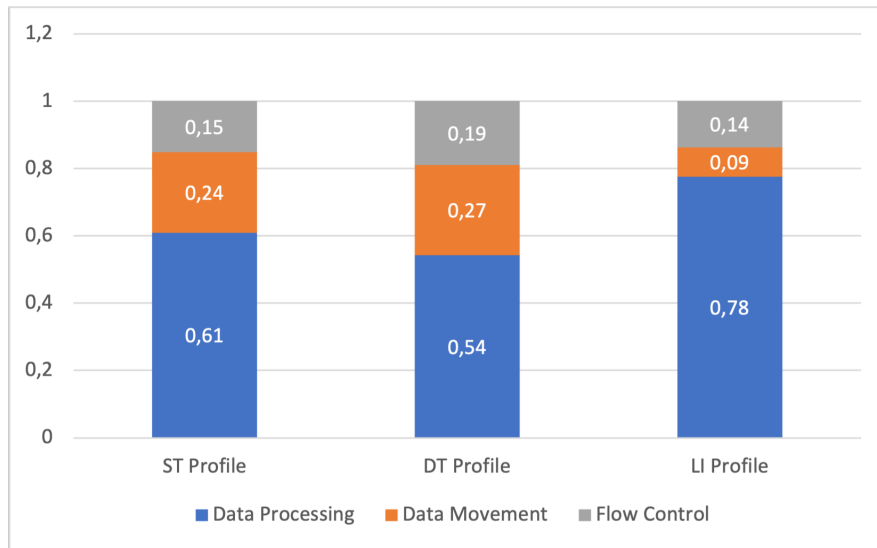


Figure 5.1: Cycle Distribution across Instruction Types in the FFT Benchmark

After inspecting the codes of both GEMM and FFT, we realize that they include a lot of computation instructions which explains why both attribute the largest amount of cycles to data processing instructions irrespective of the type of profiler. This also explains the similarity of Figures 5.1 and 5.2. The statistical profiler is also quite accurate here due to the nature of the algorithms being profiled which encounter a lot of iterations. Thus, statistically speaking, we are covering the program in terms of tracing while giving importance or bias to instructions that take more time to execute in getting attributed more cycles. The linear profiler, on the other hand, performs the worst because it equally distributes the cycles among instructions without taking into account that data movement or flow control instructions take more time to execute than data processing instructions. The deterministic profiler is considered to be the middle ground between both, not requiring a program to be iterative in nature to be accurate nor evenly distributing cycles among different types of instructions. It assigns the cycles based on an informed decision and informed approximations. Additionally, the deterministic profiler is the closest among the implemented profilers in being time proportional.

5.2 The Bluetooth Stack Profile

After running the trace and building the code profiles from the Bluetooth-based sample, the three code profiles show an interesting result with the *memcpy* function taking relatively the most number of cycles. Additionally, the *memcpy* function which is in the "string.c" file from the Zephyr RTOS implementation is directly proportional to the length of the input buffer of the data that will be transferred through Bluetooth. In the original throughput sample implementation, the size of

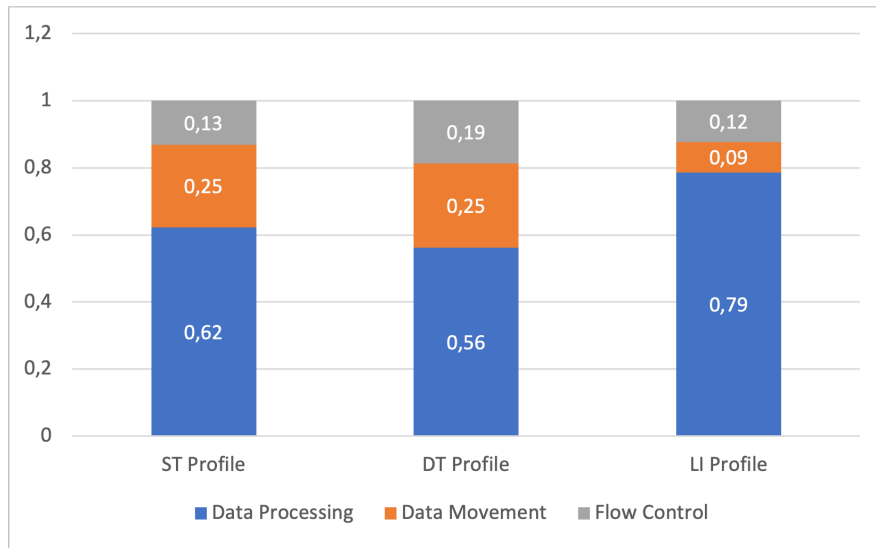


Figure 5.2: Cycle Distribution across Instruction Types in the GEMM Benchmark

the input buffer was 495. Thus, the code profiles of the *memcpy* function with a 495 input size are presented below in Listings 5.1, 5.2, and 5.3 for further analysis.

Listing 5.1: Statistical Profile - While Loop

Function: memcpy Total Cycles: 0.650489137339601			
Address	Line	Assembly	Cycle Count
0x0003A2E6	338	ldrb.w r4, [r1], #1	0.2601956549358404
0x0003A2EE	339	b.n 0003a2e0 <memcpy+0x6>	0.2601956549358404
0x0003A2E2	337	bne.n 0003a2e6 <memcpy+0xc>	0.1300978274679202

Listing 5.2: Deterministic Profile - While Loop

Function: memcpy Total Cycles: 4985			
Address	Line	Assembly	Cycle Count
0x0003A2E6	338	ldrb.w r4, [r1], #1	2002
0x0003A2E2	337	bne.n 0003a2e6 <memcpy+0xc>	992
0x0003A2EE	339	b.n 0003a2e0 <memcpy+0x6>	990
0x0003A2E0	337	cmp r1, r2	496
0x0003A2EA	338	strb.w r4, [r3, #1]!	495
0x0003A2E4	349	pop {r4, pc}	5
0x0003A2DA	299	push {r4, lr}	3
0x0003A2DE	299	add r2, r1	1
0x0003A2DC	299	subs r3, r0, #1	1

Listing 5.3: Linear Profile - While Loop

Function: memcpy		Total Cycles: 4990	
Address	Line	Assembly	Cycle Count
0x0003A2E0	337	cmp r1, r2	1000
0x0003A2E2	337	bne.n 0003a2e6 <memcpy+0xc>	998
0x0003A2EA	338	strb.w r4, [r3, #1]!	996
0x0003A2EE	339	b.n 0003a2e0 <memcpy+0x6>	995
0x0003A2E6	338	ldrb.w r4, [r1], #1	993
0x0003A2E4	349	pop {r4, pc}	2
0x0003A2DE	299	add r2, r1	2
0x0003A2DC	299	subs r3, r0, #1	2
0x0003A2DA	299	push {r4, lr}	2

After inspecting the code, we realize that the *memcpy* function has only one *while loop* and is shown below in Listing 5.4:

Listing 5.4: While Loop memcpy Implementation

```
while (n > 0) {
    *(d_byte++) = *(s_byte++);
    n--;
}
```

After changing the the code to a *for loop* and running the code profilers again, it becomes evident that the *memcpy* function takes less number of cycles and the edited code is shown in Listing 5.5:

Listing 5.5: For Loop memcpy Implementation

```
for (int i = n; i > 0; i--)
{
    *(d_byte++) = *(s_byte++);
}
```

The code profiles for the *for loop* are also shown in Listings 5.6, 5.7 and 5.8.

Listing 5.6: Statistical Profile - For Loop

Function: memcpy		Total Cycles: 0.5890982309794333	
Address	Line	Assembly	Cycle Count
0x0003A2E2	343	cmp r4, #0	0.33136775492593123
0x0003A2E4	343	bgt.n 0003a2e8 <memcpy+0xe>	0.2209118366172875
0x0003A2DA	299	push {r4, lr}	0.036818639436214586

Listing 5.7: Linear Profile - For Loop

Function: memcpy		Total Cycles: 4010	
Address	Line	Assembly	Cycle Count
0x0003A2E0	343	subs r4, r2, r1	673
0x0003A2F0	343	b.n 0003a2e0 <memcpy+0x6>	672
0x0003A2EC	345	strb.w r4, [r3, #1]!	666
0x0003A2E2	343	cmp r4, #0	665
0x0003A2E8	345	ldrb.w r4, [r1], #1	665
0x0003A2E4	343	bgt.n 0003a2e8 <memcpy+0xe>	664
0x0003A2E6	349	pop {r4, pc}	2
0x0003A2DE	343	add r2, r1	1
0x0003A2DC	299	subs r3, r0, #1	1
0x0003A2DA	299	push {r4, lr}	1

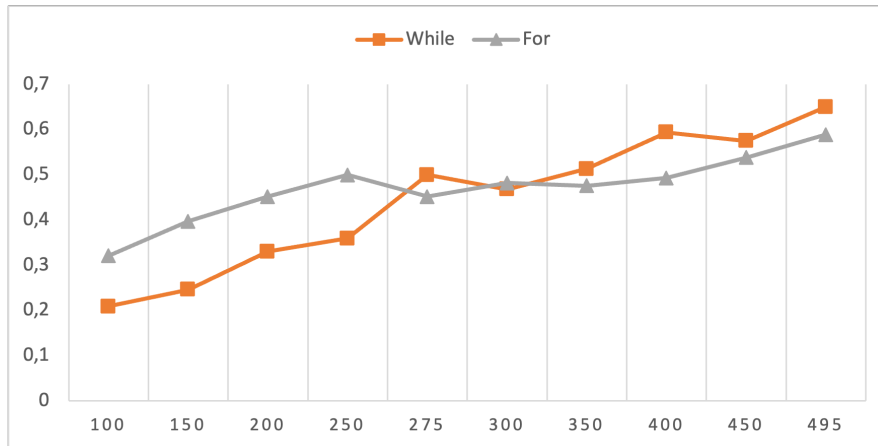


Figure 5.3: For Loop v.s While Loop Graph - Statistical Profile

Listing 5.8: Deterministic Profile - For Loop

Function: memcpy		Total Cycles: 4003	
Address	Line	Assembly	Cycle Count
0x0003A2E4	343	bgt.n 0003a2e8 <memcpy+0xe>	992
0x0003A2F0	343	b.n 0003a2e0 <memcpy+0x6>	990
0x0003A2E8	345	ldrb.w r4, [r1], #1	524
0x0003A2E2	343	cmp r4, #0	496
0x0003A2E0	343	subs r4, r2, r1	496
0x0003A2EC	345	strb.w r4, [r3, #1]!	495
0x0003A2E6	349	pop {r4, pc}	5
0x0003A2DA	299	push {r4, lr}	3
0x0003A2DE	343	add r2, r1	1
0x0003A2DC	299	subs r3, r0, #1	1

Thus, a comparative study between both types of loops was conducted. After generating the code profiles while swiping the value of the input buffer in increments of 50s for both *while* and *for* loops, Tables A.1, A.2, A.3 and Figures 5.3, 5.4, 5.5 were generated.

Figure 5.3 shows the graph of the statistical profile cycle attribution to the *memcpy* function as a function of the size of the input buffer. For input sizes less than 275, the *while* loop executes in fewer cycles than the *for* loop. For values greater than 275, the performance of the *for* loop becomes slightly better or in rare cases performs at the same degree (input size = 300). Both the linear and the deterministic profiles in Figures 5.4 and 5.5 show a more rigid differentiation in performance as the *for* loop performs better for input sizes greater than 275 while the *while* loop performs better for input sizes less than 275. The exact cycle attribution in the case of deterministic or linear profiles or percentage in the case of the statistical profile is found in Tables A.1, A.2, A.3.

From the code profiles shown above, we realize that the linear and deterministic approaches attribute the *memcpy* function an almost equal number of cycles in spite of assigning the individual assembly instructions different number of cycles. Another interesting finding that is in agreement with the Machsuite results is that

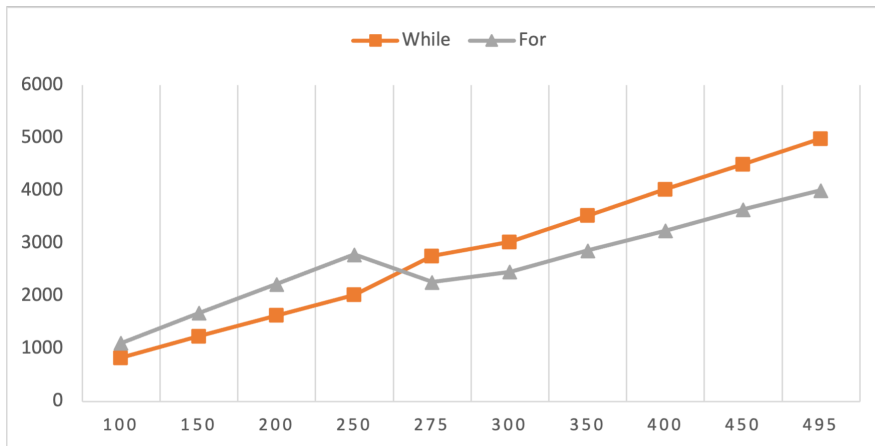


Figure 5.4: For Loop v.s While Loop Graph - Deterministic Profile

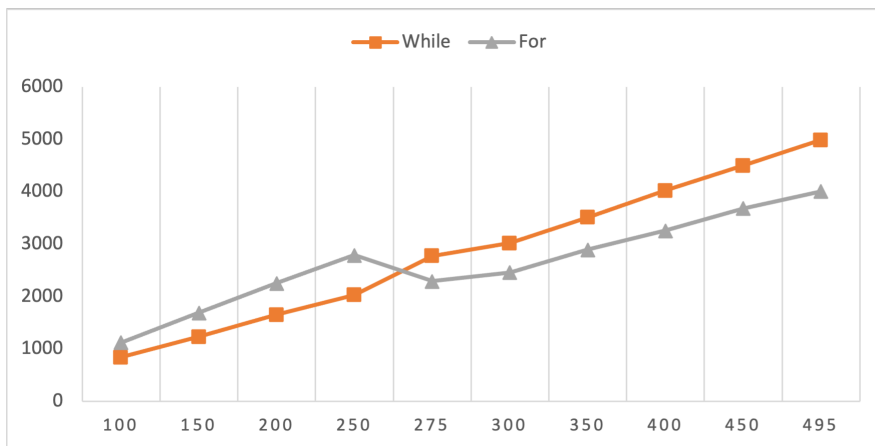


Figure 5.5: For Loop v.s While Loop Graph - Linear Profile

for both the *for* and *while* loop cases, the deterministic profiles give greater cycle attribution to data movement instructions and the linear profiles give equal importance to different instruction types when it comes to cycle attribution. The statistical profiles, however, are quite more random in selecting the instructions sampled and attributing cycles to them. This could be due to the insufficient number of iterations needed to be able to track the cycles and instructions in a more accurate manner.

Thus, from the information analyzed here, a proposed straightforward optimization could be simply checking the input buffer size and deciding whether to use a *while* or *for* loop. The sample code is shown in Listing 5.9.

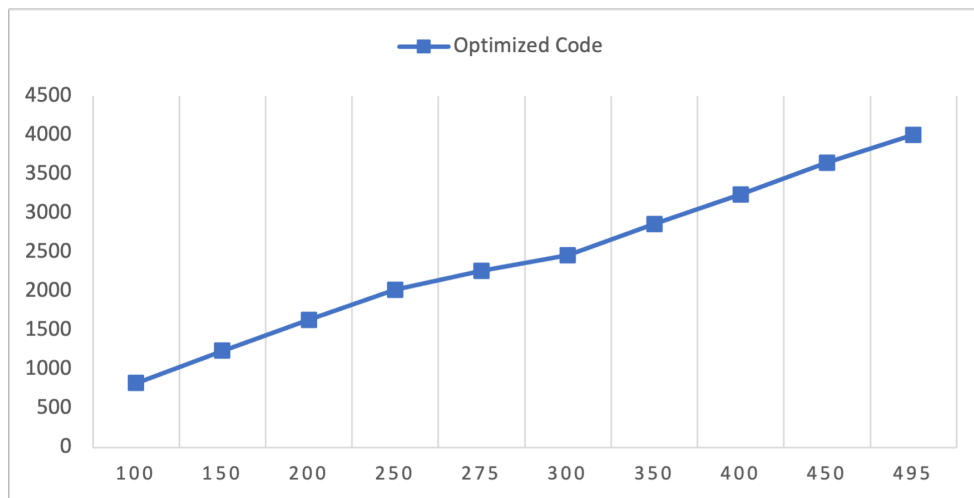


Figure 5.6: Optimized Code Cycle Attribution

Listing 5.9: Proposed Optimized Code

```

if (n < 275) {
  while (n > 0) {
    *(d_byte++) = *(s_byte++);
    n--;
  }
}
else
{
  for (int i = n; i > 0; i--)
  {
    *(d_byte++) = *(s_byte++);
  }
}

```

Taking the deterministic profiler into account as the most accurate based on the findings of this section, the deterministic profile graph of the optimized code in relation to the input buffer size is shown in Figure 5.6. The reason as to why the *for* loop performs better than the *while* loop for higher iterations could be due to many reasons. It may be that in the *for* loop, the number of iterations is known to the compiler and thus compiler optimizations may have been performed. It could also be that the *for* loop has lower overhead than a *while* loop and thus performs better in such scenarios.

Chapter 6

Conclusion

6.1 Conclusion

In this master thesis, we were motivated to achieve energy efficiency on SoCs by providing the necessary tools to detect inefficiencies. Upon looking at profiling techniques, tracing was chosen as the base for our code profiles due to its accuracy and small overhead. We provided an explanation of the mechanism of tracing and also implemented three different types of code profilers: The Statistical Profiler, The Deterministic Profiler, and The Linear Profiler.

We then compared the performances of these profilers by testing them on two Machsuite benchmark algorithms [28]. Through the analysis of the profiles generated, we realized that the deterministic profiler performed the best and the closest to being time proportional due to informed approximations in assigning cycles to instructions while the linear profiler was the worst due to equal assignment of cycles to instructions irrespective of their types. The statistical profiler was found to be accurate in the presence of an iterative application.

These profilers were then used on a case study, the Nordic Semiconductor's implementation of Bluetooth stack [29]. An inefficiency was found and confirmed by all three code profilers. We then edited the snippet of code that was responsible for this inefficiency and achieved a minor speedup.

6.2 Limitations

The main limitations of this master thesis were the following:

1. Time factor: This master thesis was undergone in almost four months and a half. A steep learning curve was needed and the outcomes of this thesis could have been better with more time. However, all the tasks were met in time due to practical time management.
2. Absence of a golden model: A golden model is a code profile that is time proportional and attribute accurately the cycles to instructions. The absence of the golden model made it slightly more difficult to analyze the performance

of the code profilers. However, this limitation was overcome by using easily understandable programs as input to the code profilers.

6.3 Future Work

The list of tasks that can be considered as future work includes the following:

1. A more thorough study when it comes to the approximations and estimations taken for the deterministic profiler. Unit testing could be a good strategy to determine the accuracy of the assumptions taken.
2. A more thorough analysis as to why the for loop performs better than the while loop in the case study. These include testing different compiler flags and isolating the memcpy function from the Bluetooth stack.
3. More case studies could be included to both verify the performance of the code profilers designed and to improve any given inefficiencies in the given case studies.

Bibliography

- [1] P Sancheti, *A Holistic Approach to Energy-Efficient System-on-Chip*, <https://www.synopsys.com/content/dam/synopsys/solutions/documents/a-holistic-approach-to-energy-efficient-soc-design-wp.pdf>, [Accessed 30-08-2023].
- [2] Janani, *What is Code Profiling? – A Detailed Explanation*, <https://www.atatus.com/blog/what-is-code-profiling-a-detailed-explanation/>, [Accessed 13-08-2023].
- [3] *GitHub - Gperftools*, <https://github.com/gperftools/gperftools/tree/master>, [Accessed 16-08-2023].
- [4] *GitHub - EECS-NTNU/PPperf*, <https://github.com/eecs-ntnu/ppperf>, [Accessed 16-08-2023].
- [5] *Linux Perf*, <https://www.swift.org/server/guides/linux-perf.html>, [Accessed 16-08-2023].
- [6] A. Djupdal, B. Gottschall, F. Ghasemi and M. Jahre, 'Lynsyn and Lynsyn-Lite: The STHEM Power Measurement Units,' in *Towards Ubiquitous Low-power Image Processing Platforms*, M. Jahre, D. Göhringer and P. Millet, Eds. Springer International Publishing, 2021.
- [7] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi and K. Hazelwood, 'Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation,' in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, Association for Computing Machinery, 2005.
- [8] D. Bruening and D. Lane, 'Efficient, transparent, and comprehensive runtime code manipulation,' 2004.
- [9] *Clang 7 Documentation*, <https://releases.llvm.org/7.1.0/tools/clang/docs/UsersManual.html>, [Accessed 16-08-2023].
- [10] Z. Ning and F. Zhang, 'Hardware-Assisted Transparent Tracing and Debugging on ARM,' *IEEE Transactions on Information Forensics and Security*, 2019.
- [11] T. Staff, *Telink | The Evolution of Bluetooth® to Becoming a Low-Power Protocol*, <https://www.telink-semi.com/evolution-of-bluetooth-to-becoming-a-low-power-protocol/>, [Accessed 12-08-2023].

- [12] *Bluetooth Low Energy - Nordic Semiconductor*, <https://www.nordicsemi.com/Products/Bluetooth-Low-Energy>, [Accessed 21-08-2023].
- [13] *SEGGER J-Trace Streaming Trace Probes*, <https://www.segger.com/products/debug-probes/j-trace/>, [Accessed 13-08-2023].
- [14] *Setting Up Trace with Ozone - SEGGER Wiki*, https://wiki.segger.com/Setting_Up_Trace_with_Ozone, [Accessed 19-08-2023].
- [15] *Intel® 64 and IA-32 Architectures Software Developer Manuals*, <https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html>, [Accessed 31-08-2023].
- [16] P.J. Drongowsk, 'Instruction-Based Sampling: A New Performance Analysis Technique for AMD Family 10th Processor,' *AMD*, 2007.
- [17] B. Gottschall, L. Eeckhout and M. Jahre, 'TIP: Time-Proportional Instruction Profiling,' in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, Association for Computing Machinery, 2021.
- [18] B. Gottschall, L. Eeckhout and M. Jahre, 'TEA: Time-Proportional Event Analysis,' in *Proceedings of the International Symposium on Computer Architecture (ISCA)*, Association for Computing Machinery, 2023.
- [19] J. Tosi, F. Taffoni, M. Santacatterina, R. Sannino and D. Formica, 'Performance Evaluation of Bluetooth Low Energy: A Systematic Review,' *Sensors*, 2017.
- [20] *NRF52840 - Nordic Semiconductor*, <https://www.nordicsemi.com/products/nrf52840>, [Accessed 30-08-2023].
- [21] *Introduction: Zephyr Project Documentation*, <https://docs.zephyrproject.org/latest/introduction/index.html>, [Accessed 30-08-2023].
- [22] *SEGGER J-Link SDK*, <https://www.segger.com/products/debug-probes/j-link/technology/j-link-sdk/>, [Accessed 23-08-2023].
- [23] *Nordic Semiconductor Infocenter*, <https://infocenter.nordicsemi.com/index.jsp>, [Accessed 23-08-2023].
- [24] *Instruction Level Parallelism - GeeksforGeeks*, <https://www.geeksforgeeks.org/instruction-level-parallelism/?ref=lbp>, [Accessed 20-08-2023].
- [25] *Cortex-M4 - ARM*, <https://developer.arm.com/Processors/Cortex-M4>, [Accessed 20-08-2023].
- [26] *Instruction Set Summary - ARM*, <https://developer.arm.com/documentation/ddi0439/b/Programmers-Model/Instruction-set-summary/Cortex-M4-instructions>, [Accessed 25-08-2023].
- [27] *Load Store Timings - ARM*, <https://developer.arm.com/documentation/ddi0439/b/Programmers-Model/Instruction-set-summary/Load-store-timings>, [Accessed 25-08-2023].

- [28] B. Reagen, R. Adolf, Y. S. Shao, G.-Y. Wei and D. Brooks, 'MachSuite: Benchmarks for Accelerator Design and Customized Architectures,' in *2014 IEEE International Symposium on Workload Characterization (IISWC)*, 2014.
- [29] *Bluetooth: Throughputs; nRF Connect SDK 2.4.99 Documentation*, https://developer.nordicsemi.com/nRF_Connect_SDK/doc/latest/nrf/samples/bluetooth/throughput/README.html, [Accessed 27-08-2023].
- [30] M. D. Lam, E. E. Rothberg and M. E. Wolf, 'The Cache Performance and Optimizations of Blocked Algorithms,' in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, Association for Computing Machinery, 1991.

Appendix A

Appendix

Input Buffer Size	While Loop (Cycles)	For Loop (Cycles)
100	0.209	0.321
150	0.246	0.397
200	0.33	0.452
250	0.359	0.5
275	0.5	0.452
300	0.468	0.482
350	0.513	0.476
400	0.594	0.493
450	0.575	0.538
495	0.65	0.589

Table A.1: Statistical Profile - For Loop v.s While Loop

Input Buffer Size	While Loop (Cycles)	For Loop (Cycles)
100	824	1105
150	1237	1677
200	1632	2225
250	2019	2768
275	2757	2262
300	3026	2460
350	3523	2861
400	4023	3240
450	4498	3644
495	4985	4003

Table A.2: Deterministic Profile - For Loop v.s While Loop

Input Buffer Size	While Loop (Cycles)	For Loop (Cycles)
100	844	1124
150	1233	1692
200	1656	2256
250	2031	2792
275	2778	2294
300	3023	2464
350	3512	2900
400	4024	3259
450	4502	3686
495	4990	4010

Table A.3: Linear Profile - For Loop v.s While Loop



NTNU

Norwegian University of
Science and Technology