



STATIC ENERGY ANALYSIS OF LOW LEVEL PROGRAMS

INDEPENDENT STUDY THESIS

Presented in Partial Fulfillment of the Requirements for
the Degree Bachelor of Arts in the
Department of Mathematics and Computational Sciences
at The College of Wooster

by
Patrick May
The College of Wooster
2024

Advised by:

Professor Drew Guarnera (Computer
Science)



THE COLLEGE OF
WOOSTER

© 2024 by Patrick May

ABSTRACT

This work presents the theory and approach of a static energy analysis tool that semi-automatically predicts energy cost of compiled programs within the ARM v8 assembly architecture. It discusses various background information that is required to understand the methodology of modern software, including compilers, computer architecture, etc. It also presents the concept of cost relations and upper bound static cost analysis specifically, as well as sampling alternative semi-static approaches. A Raspberry Pi 4 Model B is used as a testbench for dynamic energy benchmarking and testing the validity of the static tool.

To friends and family; and a more sustainable future

ACKNOWLEDGMENTS

Acknowledging everyone who has influenced and helped me on this journey is a Sisyphean task, but I shall do my best. First, to my advisor, Professor Drew Guarnera for his suggestions and guidance throughout the IS process, and for serving a crucial role in my entire computer science education at the College of Wooster. He has been truly a wonderful mentor that I firmly believe is the standard for excellence as an instructor. An additional thank you to Professors Heather Guarnera and Daniel Palmer, for believing in me and my ability to complete this project. To my family and friends, for constant support with a senior who was dead-set on doing everything everywhere all at once. A named acknowledgement to Taemour Zaidi, for being an amazing friend, motivator, and confidant through IS woes. To Alex White, for being a wonderful friend who empathized with my procrastination of this work. To my roommates Lyonel Fritsch and Aiden Lentz, for being wonderful people and tolerating my esoteric computer science ramblings. A further mountain of gratitude towards all my peers that have influenced my undergraduate education in innumerable ways. Thanks to my coworkers and manager at Web, for keeping me around even as a college student, aiding in my professional development while also giving me enough space to excel in the tail end of my undergraduate career. And a final special thanks to Dr. Samir Genaim for sharing direct access to their PUBS solver executable.

CONTENTS

Abstract	iii
Dedication	iv
Acknowledgments	v
Contents	vi
List of Figures	viii
List of Tables	ix
List of Listings	x
CHAPTER	PAGE
1 Introduction	1
2 Background	6
2.1 Compilers	6
2.1.1 At a Glance	6
2.1.2 Beneath the Hood	8
2.2 Assembly Language	10
2.2.1 Assembly Languages	11
2.2.2 Tokenizing ARM	13
2.2.3 Parsing ARM	15
2.2.4 Abstract Syntax Trees	16
2.2.5 Type Systems and Semantic Analysis	17
2.2.6 Intermediate Representations and Control Flow Graphs	20
2.3 Modern CPU Complications	21
2.3.1 Running a Program	21
2.3.2 Caching	23
2.3.3 Pipelining	25
2.3.4 Branch Prediction	28
2.4 Energy Analysis	30
2.4.1 Static vs. Dynamic	31
3 Program Cost Analysis	34
3.1 Recurrence Relations	34
3.2 Cost Relations	36
3.2.1 Cost Relation Extraction	42

3.2.2	Automatic Upper Bound Inference	47
3.2.2.1	Upper Bound Node Counting	51
3.2.2.2	Upper Bounding Individual Node Cost	53
3.2.2.3	Automatic Upper Bound Issues	55
3.3	Practical Upper Bound Solver (PUBS)	57
3.3.1	Prolog Overview	57
3.3.2	PUBS Interface	61
3.4	Partial Dynamic Cost Analysis	64
4	Implementation And Hardware Benchmarking	66
4.1	Static Energy Analyzer Process	66
4.2	Hardware Choice	69
4.3	Energy Testing Background	70
4.4	ARM Testing Methodology	73
4.5	Experimental Energy Per Instruction Results	76
5	Discussion	80
5.1	Example SEA Workflow	80
5.2	A Non-SEA	87
5.3	Challenges	88
5.3.1	Static Analysis	88
5.3.2	Hardware Testing	89
6	Conclusion	92
6.1	Future Work	92
6.2	Closing	94
	References	98
	Index	106

LIST OF FIGURES

Figure		Page
2.1	Levels of Programming Languages	8
2.2	Broad Steps of a Compiler [51]	9
2.3	'Null' C Program	12
2.4	ARM ASM from Figure 2.3	12
2.5	Simple Label Regular Expression	14
2.6	3 Dimensional Table of Type Systems of Various Programming Languages	19
2.7	32 and 64 bit register structure in ARMv8	22
2.8	Cache Heirarchy Model of Modern Processors	23
2.9	Generalized 4 Stage Processor Pipeline	26
3.1	Simple Foo() Method in C++	38
3.2	ARM ASM from Figure 2.3	38
3.3	Common Programming Paradigms [31]	58
3.4	Programming Languages Colored By Paradigm over Time [39]	59
3.5	ArrayReverse() Java Method Code [4]	61
3.6	Cost-Equation System in PUBS syntax of Figure 3.5	61
3.7	PUBS Executable Usage	62
3.8	PUBS ArrayReverse() Partial Output	63
4.1	Semi-Automated Static Energy Analyzer Process	68
4.2	Raspberry Pi Testbench	69
4.3	Energy Reading Multimeter	69
5.1	C Code for PowerOf function	81
5.2	PowerOf Compilation and Flags	81
5.3	ARM Assembly of PowerOf function	82
5.4	Control Flow Graph for powerOf Assembly Code	84
5.5	Example of Possible PowerOf PUBS Input [4, 3]	86

LIST OF TABLES

Table		Page
3.1	PUBS/Prolog Syntax	62
4.1	Static Current Consumption at Different Clock Speeds of Pi	77
4.2	Estimated Energy Per Instruction (pJ), integer operations	77
4.3	Estimated Energy Per Instruction (pJ), bitwise logic operations	77
4.4	Estimated Energy Per Instruction (pJ), floating point operation	78
4.5	Estimated Energy Per Instruction (pJ), move, compare, load, store operations	78

LIST OF LISTINGS

Listing		Page
4.1	Benchmark Instruction Test File Structure	74
4.2	add with out read-after-write	75
4.3	add with read-after-write hazards	75

CHAPTER 1

INTRODUCTION

The field of software development is rapidly evolving, as more and more industries begin to utilize and rely upon technology to simplify, manage, and modernize existing business functions. A large amount of developer effort targets improving existing software along various benchmarks. Software systems are optimized for *speed*, *memory utilization*, and sometimes *code size* itself [26]. Another metric that software systems can be evaluated upon is that of *energy efficiency*. Energy efficiency has a multitude of reasons for why it should be studied. For mobile devices or any technology system that pulls power from a finite battery resource, energy efficiency will improve overall device uptime. If the hardware stays the same but with more energy efficient software, new life is breathed into a device with its additional runtime without requiring hardware changes. On embedded and low-level systems, energy usage may be limited by the amperage the circuit board can handle, leading to a desire for minimizing average energy cost as well as overall energy usage. Large companies that have warehouses of servers providing cloud computing power must give real consideration to energy costs that arise from day-to-day operations. Server farms that utilize more energy cost more from the increased energy used, and also in additional resources being required to cool high-power servers. In an enterprise situation, energy-efficient servers provide a

different benefit compared to mobile devices through reducing running costs. All different styles of computers use electricity as the energy necessary to run. This electricity is generated through various means that does not have a positive or neutral environmental effect [35]. Hence, the desire for energy optimization. If one can better understand the energy required to perform various computations, then alternative approaches can be attempted to reduce the power draw. All else held equal, energy-efficient software is better for the environment.

Producing energy-efficient software is one of the goals of Green Software Engineering. Green Software Engineering is a growing sub-field within technology that aims to more directly consider the effects of software on its surroundings – the physical environment and society as it interfaces with it. Unfortunately, green software engineering exists only on the fringes of modern-day development considerations, which typically are centered around the speed of development, interoperability, flexibility, and cost. Much of the work that has been done in the green software engineering field focuses on evaluating fundamental blocks of the development lifecycle. Some examples of existing research include comparing different programming languages for energy efficiency [41], examining common areas of energy leaks within simple scripts [40], or demonstrating a design framework that holds energy usage as one of the main considerations [49]. Many further works are exploratory, but generally, energy efficiency is a non-concern for the end-developer. Instead, energy optimizations are left to compiler engineers, systems architects, and other specialized developers that are separated from the end developer through layers of abstraction. For example, enterprise developers need not concern themselves with the ongoings within a compiler. Instead, the build and compilation process is abstracted as a conversion of their work to a machine-runnable program. This abstraction allows for the people developing the compiler to meddle with the specifics as long as the end result remains invariant. As computational theory

has advanced, various optimizations have been created for programs that strictly improve the end result, through processes such as dead code elimination, loop unrolling, in-lining, etc. These layers of abstraction reduce the amount of necessary knowledge for any one individual developer, which allows for code to be created quicker without one needing tedious knowledge of every single occurrence between writing code and having a computer execute it. However, these abstractions also serve as a barrier between areas of expertise which makes it more difficult for an end-developer to improve their software design for the underlying technologies it is built upon. Optimizing code for any specific reason is difficult, and is even more difficult if one does not know where to start. Development tools such as profilers and debuggers aid a developer in understanding more precisely what occurs within a program. There is a distinct lack of energy efficiency tooling that increases the difficulty for an end-developer.

Developing energy-aware software is particularly challenging because of how non-standard it is. Many applications are profiled to find spots to increase speed. These speedups generally can come from algorithmic changes that would likely have the same effect on the software regardless of hardware platform. Going from bubble sort to mergesort for large sets of data would give any program executed on almost any platform a hardware speedup. Power usage is not as simple. Energy is inherently more tied to hardware - the CPU manufacturer, the CPU architecture, (non)existence of multithreading, and a multitude of other factors are at play when trying to get a physical energy reading of running software on a computer. The same program may take different amounts of energy on different computers, and even the same computer, depending on caching and branch prediction within a CPU, making a program run faster after multiple sequential runs [8]. Out of order instruction execution may cause a program to perform unexpectedly when observed at a very low level. Different operating systems or always-running background processes

may have a hidden power draw that is not expected when analyzing a computer for power usage. Multiple different computer architectural pieces are moving all at once when trying to understand the energy usage of any given software.

A computer science goal of this thesis is to better understand many of these moving parts that are essential when operating in an energy-aware development mindset. A software development goal of this thesis is to develop and demonstrate the steps of a Static Energy Analyzer (SEA). A SEA serves as a program that aims to predict overall energy costs of running a compiled program. At its most simple, a SEA should be able to take in a source code project in its compiled, assembly language form, and after various analysis steps, will output a predicted energy 'cost' of running that program. As a *static* analysis tool, a SEA does not have to concern itself with tracking its own overhead, compared to *dynamic analysis tools*. This is because static analysis interacts with a program at compile time (when a program could be considered text), while dynamic analysis observes a program while it is running. *Dynamic analysis* is a common software profiling technique that involves running additional software that tracks various parts and pieces of the target program as it runs. While dynamic profiling can easily capture the broad strokes of large speed slowdowns and memory leaks [23], it is quite difficult to dynamically profile a program for energy usage at anything beyond the total energy usage of a program, due to the fact that constantly measuring energy statistics of a system takes energy that adds overhead to the results.

The SEA implemented as a part of this thesis touches on many various aspects of mathematics and computer science. It begins by parsing ARM Assembly (ARM ASM, or just ARM), which is a computer language problem that involves understanding ARM grammar. Then the SEA steps further into the world of compilers by translating the internal representation into a control flow graph (CFG). From simplified CFG form, I then semi-automatically extract a cost-relation for the program in question.

Then, I attempt to automatically determine a discrete upper bound system for the cost relation system through the use of the Practical Upper Bounds Solver (PUBS). Automatically solving for an upper bound is difficult and has many limitations which are discussed in Chapter 3, so an alternative is to run a program dynamically to manually observe the required constants to solve the cost relation system. Separate from the static assembly analysis portion, I benchmark a small computer to trace its power usage for various small pieces of program execution. These power usage statistics for individual assembly instructions become the basic 'costs' of which to allow for a final answer of the cost-relation system created in a different step. I then join the two experiments by mapping the underlying relations to individual energy costs. Through this process, the static energy analyzer (SEA) estimates a final energy usage for the inputted program.

The SEA's energy estimation for a program can then be tested against empirical test-bench data to determine the accuracy of the SEA compared to real execution.

CHAPTER 2

BACKGROUND

To perform static program analysis, one first must understand everything that goes into how programs work. In this chapter, I introduce background terminology about key computer science concepts within this thesis, such as ARM assembly (ARM ASM), compilers, abstract syntax trees (ASTs), and control flow graphs (CFGs).

2.1 COMPILERS

2.1.1 AT A GLANCE

As a static program analyzer interfaces with *compiled code*, it is valuable to understand how typical compilers work beyond the basics. At their most simple, compilers exist as a translation tool, taking a source language to a target language [51]. One novel implication from this definition is that many software tools we use everyday are also compilers: web browsers translating html to an interactive web page, word processors compiling a rich text document (markdown, word) into another file format such as pdf or html, compression tools targeted towards a language utilized a reduced representation, etc. It is useful to understand the broadest form of ‘compilers’, but these are not the typical compilers that are referred to when working

in a computational science context. The compilers of focus are compilers that convert higher-level language programming code into low-level assembly language and machine code. Every compiled programming language has (at least) one compiler available to it, such as `gcc` and `clang` for C, `rustc` for Rust, `zig build-*` for Zig, etc. Compilers are **not** interpreters, which are alternative software tools that directly execute source code, compared to the intermediate machine code a compiler outputs. Modern programming languages combine various elements of compilers and interpreters, such as Java's workflow of compiling a project to bytecode, which is then interpreted by a Java Virtual Machine interpreter. Other languages leverage a process known as **Just in Time compilation (JIT)** to translate interpreted bytecode into machine code as needed. Various high level programming languages use various pieces of compiler and interpreter methodologies. For creating a Static Energy Analyzer (SEA), the simplest approach is to comprehend the assembly language representation of a program all at once – meaning interpreters, JIT, and other schemes are not of focus for this work.

Compilers do not simply translate code – they also improve code [51]. A compiler may know of specific optimizations for the current system architecture that it can apply. Essentially, a desire to – at no cost to the end developer – make programs faster. The simplest possible example of this is stereotypical C compilers with `-O1`, `-O2`, `-Ofast`, ... optimization flag options. Within energy-aware computing, prior work has been done trying to create compilers that optimize for energy efficiency, to middling results. Modern compilers typically produce machine code, which is extremely difficult for humans to read. A step up in complexity from machine code is assembly, which with the right compilation flags, can be outputted as a side effect during normal compilation. Figure 2.1 loosely illustrates the high to low level programming language spectrum. Modern day development is infrequently done using assembly language, as it is still difficult for humans to

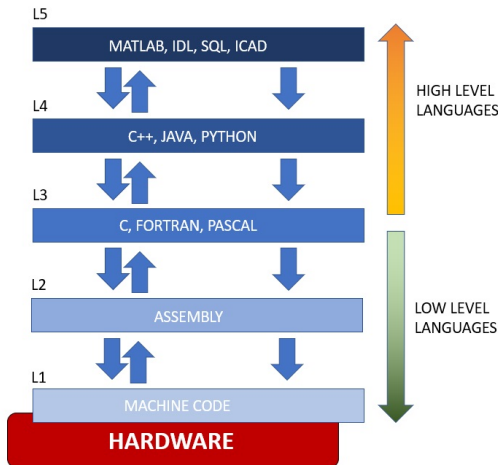


Figure 2.1: Levels of Programming Languages

understand and lacks many higher level programming constructs that stepping one level up the language complexity ladder provides.

As part of generating input for the SEA, multiple simple programs written in various higher level languages (C, to start) will be compiled and have Assembly output from them. Assembly language is still tightly tied to the hardware it runs on, the most common divide appearing between x86 and ARM instruction sets. Further assembly architecture discussion can be found in Section 2.2.

2.1.2 BENEATH THE HOOD

The process of taking a source code project to an executable includes multiple steps, compilation being only one of them. After compilation programs must be processed through an assembler and a linker. An **assembler** consumes assembly code and produce object code, which is a nearly direct machine code translation of assembly code [51]. The object code is still incomplete, as it is missing final memory locations that will be used, as well as connections to prewritten library code that the object file is using. Then, the object file(s) pass through a **linker** that fills in these memory location and library code gaps. Modern ‘compiler’/build systems typically do all of these steps beneath the hood, but a distinction is necessary because a ‘compiled’

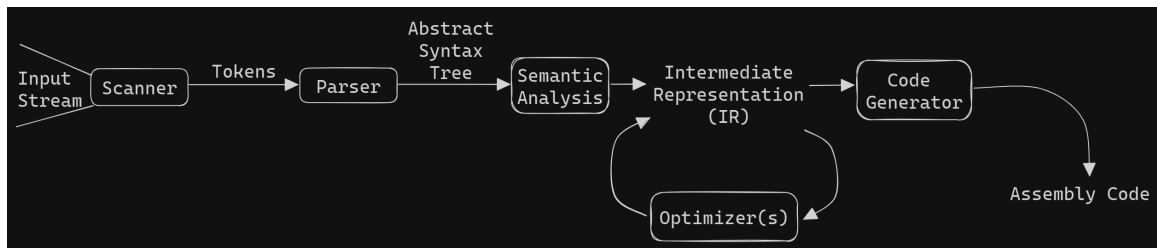


Figure 2.2: Broad Steps of a Compiler [51]

program's asm is different from a 'complete' program's asm, which may mean different things for my SEA. Currently, the SEA is built to work with 'compiled' programs – that is, before linking with builtin libraries and other binaries occur. To access the assembly outputs of modern compilers, various command flags must be used, `-S` being the most common for C & C++. If one instead wants to look at the assembly representation of a complete, linked program, they must instead **disassemble** the executable file itself, usually through the use of a utility such as `objdump` [45].

The 'Compilation' step in the code-to-execution process has various parts within it, as well. **Scanning** the program, also called **lexing** is the process of taking in raw text and identifying tokens from the stream of characters. This is similar to processing words in natural language, however programming languages can have very different rules as to what constitutes a token. After scanning, the sequence of tokens is then fed to a **parser**, which combines sequences of tokens into broader sentences and expressions using the *grammar* of the source language. These expressions are woven together to create an **abstract syntax tree**, an abstract representation of all expressions and grammar from the program. **Semantic routines** then traverse the AST deriving further meaning from expressions, such as type inferences [51], expression expansion [12], etc. After semantic analysis, the code has been transformed into some **intermediate representation** that is different from the source, but not assembly/object/machine code. **Optimizers** are applied on the intermediate representation in order to improve the program on a variety of

dimensions. Typically, optimizers exist to reduce code size, increase program speed, or increase program efficiency. However, optimizers can really do whatever the compiler designer desires. Intermediate representation exists as a reduced form of a high level language but imbued with the constraints of hardware specific concerns. After optimization routines have been run on the IR, the compilation step is complete after passing the IR through a **code generator** to finally produce assembly code [51]. As discussed prior, modern ‘compilers’ also include the further steps of assembly and linking.

A SEA must perform some compiler-like tasks. Instead of translating high level source code to assembly as a typical compiler, the SEA will first **tokenize** assembly language as an input. Assembly is not a superbly difficult language to parse, however the hardware and software specificity of it do mean a true universal grammar is not easily found [18]. After scanning and parsing of assembly, the SEA will have built an ASM AST - assembly language abstract syntax tree. Hence, the justification for an overview of compilers – one could consider the SEA a compiler, as it translates assembly to energy cost relations.

2.2 ASSEMBLY LANGUAGE

Assembly language arguably the lowest level of programming possibly palatable to developers. ‘Assembly language’ is a misnomer – multiple assembly languages exist. The most common assembly languages (ASMs) commonly discussed are x86 and ARM asm, although other assembly languages exist for esoteric hardware designs [42]. Assembly language is good for being the final step before being converted to machine code. Assembly language compiled from a higher level language is not unique, it will differ based on compiler versions, optimization flags, etc. Assembly languages are more readable than machine code as it is not just a stream

of hexadecimal data. Assembly languages are line-oriented [42]. Each language statement is on its own line, with multiline statements only available through special line-continuation character. Each line of assembly code may be **labeled**, and will contain an **operation field**, **operand field**, and an optional **comment field**. Refer to Figure 2.4 for a simplified asm example program.

2.2.1 ASSEMBLY LANGUAGES

x86 assembly is an older assembly language that is paired with most modern computers. Being older, and for the longest time, being the only real architecture option, has led to X86 being quite complex as it tries to maintain backwards compatibility while also introducing new features as new hardware features are released [30]. x86 is classified as a Complex Instruction Set Computing (CISC) assembly language. CISC style assembly has complex instructions that could perform multiple fundamental machine instructions in one assembly instruction. It has less general registers, more complicated addressing modes, and more data types compared to alternative assembly styles [37]. Since CISC is complex, modern processors have a host of specific optimizations on a new level below assembly yet above machine code: microcode [30]. ARM is comparatively a more modern assembly language that is classified as a RISC: Reduced Instruction Set Architecture. RISC is an assembly style that typically has fewer possible instructions that are simpler and more general than instructions in CISC. Instructions in RISC are one word or less, and each can be expected to be completed in one clock cycle. Of interest to the energy-conscious developer is that RISC (and ARM Assembly, by extension) uses less power than CISC processors. In modern contexts, there is no significant difference in instruction counts of a program compiled to ARM or x86. However, x86 architecture allows for more hardware specific micro-optimizations that means



```
1 int main() {
2     return 0;
3 }
4
```

Figure 2.3: 'Null' C Program



```
1 .arch armv8-a
2 .file "null.c"
3 .text
4 .align 2
5 .global main
6 .type main, %function
7 main:
8 .LFB0:
9 .cfi_startproc
10 mov w0, 0
11 ret
12 .cfi_endproc
13 .LFE0:
14 .size main, .-main
15 .ident "GCC: (Ubuntu 9.4.0-1ubuntu1~20.04.2) 9.4.0"
16 .section .note.GNU-stack,"",@progbits
17
```

Figure 2.4: ARM ASM from Figure 2.3

x86 is generally more performant than ARM, but not due to the instruction set itself [9].

ARM is the assembly language of choice for this thesis because it is a RISC-style assembly language. As a more modern development and simplicity compared to a complex instruct set architecture (ISA), it is easier pick up and account for the many constant factors of program execution from an architecture standpoint. Mobile devices are largely ARM based at this point, and (as of 2023) other device families are adopting ARM as a worthwhile platform as well. Apple silicon, such as the M* and X* chipsets which are ARM based, are examples of ARM ISA adoption.

Even within ARM asm, multiple versions exist that have various limitations. The original ARM specified in the 90s that was part of Acorn computers had only about 50 assembly instructions [6]. As processors and hardware has improved, ARM versions have added more operations. For ARMv8 specifically, there are around 354 specific instructions in the specification [6]. This being said, accounting for different variants of operations from single instruction multiple data (SIMD) and scalable vector extension (SVE) sets, this count may be as high as 1300 possible instructions including variants. ARMv8 is still RISC, as the instructions themselves are still simple, there are just more possible simple instructions as hardware has improved. For reference, x86_64 (a very common CISC assembly language) has

at least 981 operation aliases/mnemonics and around 3,700 instruction variants. ARMv8, specifically AArch64, is the instruction set of the Raspberry Pi testbench that is testing the SEA [47]. To extract cost relations and create energy mappings to each instruction in an assembly program, one first must understand what ARMv8 assembly looks like.

2.2.2 TOKENIZING ARM

The Static Energy Analyzer (SEA) reads ARM assembly as input and first must perform the first two steps of a compiler, **scanning** and **parsing**. Tokenizing assembly (asm) is not a complicated process, as discussed prior, there are only 4 main types of tokens that need to be considered:

- **Labels** `main:, LFE0:, ...`

End in a colon, alphabetical first character followed by alphanumerics or specific symbols. Allow for spots in the assembly code to be referred to directly.

- **Operation** `mov, ret, ldp, ...`

Are either 3-4 letter *mnemonics* that have a direct machine instruction, or *assembler directives/pseudo operations* that start with a '.' (dot). Directives do not have a direct machine instruction, instead giving the assembler program specific instructions that are typically hardware specific [42].

- **Operands** `sp, fp, x20, 75, main, ...`

Things that are being operated upon by the operation field prior. Operands can be either *assembler names* (such as labels or registers), *programmer names* (such as variables or constants), and *literals* (explicit values, i.e. 42). Each operation may have 0 to 3 operands, inclusive [42].

- **Comments** comments come after '@' symbol till end-of-line
Freeform text that is ignored by the assembler, human readable text analogous to any other programming language. Some styles of assembly instead use ';' as the comment delimiter

Of the 4 fundamental sections of an assembly program, only 3 are absolutely necessary, as comments can be ignored if desired.

When learning how compilers tokenize input files, the concepts of **regular expressions** and **finite state machines** are used to represent how stream of characters are classified into streams of tokens. Regular expressions are sequences of normal and special characters that allow for representing and matching multiple possible tokens. For example, we could define an ARM ASM label regex as shown in 2.5.

$$[a - zA - Z][a - zA - Z | 0 - 9]* :$$

Figure 2.5: Simple Label Regular Expression

which can be interpreted as an alphabetical first character, followed by any number of alphanumerics and ending with a colon. A regular expression can then expanded into a finite-state-machine, which mutates state based on what character it sees next, and upon reaching desired end characters emits a complete token. Since assembly language is simple and rigorously constructed, abstracted formal lexing techniques are not fully needed, given the lexer is constructed to be allowed some additional information about the file it is tokenizing. For example, assembly is line oriented, so on each line the 4 token fields come in the same order, separated by a whitespace of some form (a tab by convention). With this additional information, one can tokenize based off of both position and typical finite machine style approaches. Some fields, such as the *operand* field, may have a variable number of individual operand tokens. In this case, regular expression representation and FSM scanning is useful.

Each line of assembly may contain each field (label, operation, operand, comment), but typically do not. Not all lines are labeled, nor are all lines commented. Some lines are only comments or only whitespace – to help with readability. With these rules, we can statically transform an assembly language text file to a stream of tokens. Recall that throughout all of this, the SEA has underlying knowledge of how much energy running each assembly instruction takes. Why can we not just multiply each energy cost by the frequency of its appearance in this assembly token stream? We still have not considered some fundamental constructs of programming: recursion, looping, conditional logic, etc. We have a sequence of tokens, essential ‘words’ in assembly, yet we have no comprehension of *what* they mean as a whole. To start looking at ‘sentences’ we have to **parse** sequences of words.

2.2.3 PARSING ARM

The act of **arsing** tokens is analagous to checking if a string of words in natural language makes sense [51]. Parsing a programming language is typically done through the use of a context-free grammar. Context-free grammars are a set of rules that formally describe permissible sequences of tokens from a scanned input stream. They allow recursion and thus are a much more powerful tool than regular expressions in terms of the possibility space of things that they can represent. Context free grammars do have an ordering based on how much complexity they permit. LL(1) grammars can parse an entire sequence just by looking at the current token and one token forward [51]. LR(1) grammars are more powerful as they allow more types of recursion, but this also makes them more complicated to perform by hand. Assembly is a strict and reduced language, so many of the formalities and special features that come from more difficult context free grammars are unneeded. Since the assembly language under consideration is being output from a valid program, errors are impossible barring compiler bugs, so the checking a program

for validity is unneeded. From parsing, operations and their operands can be linked together to form individual blocks of execution. Some of these blocks contain sentences that explain movements that the SEA wants to understand (conditional logic, loops, etc.) in the form of comparisons, un/conditional jumps, etc. A context free grammar representation of ARM is not short, as essentially every mnemonic must have its own defined sequence of allowed operands following it. If we forfeit some of the formality and rigorousness of a proper context free grammar description of ARM, the grammar can be described much more tersely. There is significantly more depth and possibilities for how various programming languages are parsed with various methods, but they are not of interest nor in scope of this work.

Parsing a language can be done with a few different goals in mind, such as simple *validating* the conformance to a certain standard, *interpreting* a program to find an end result, or *translating* a program to something else through the intermediate construction of an **abstract syntax tree**. Parsing a program to attempt to interpret an end result is different from actually executing the program. At the parse step, the program is still being observed statically, and such there are many unknowns to the compiler. Tools exist for static analysis estimation of a program, similar to the idea of a static energy analyzer.

2.2.4 ABSTRACT SYNTAX TREES

Abstract syntax trees (ASTs) are common products of the parsing steps in compilers. Alternatively, sometimes the parsing step yields a parse tree or just checks sentence structure for validity. For higher level languages, an AST is a tree that represents the entire structure of sentences within an entire program. Trees (from a graph theory perspective) are connected, acyclic, undirected graphs. In computer science, trees also have an imbued heirarchical structure, such that each node has one parent and any number of child nodes. Thinking back to Figure 2.2, the static assembly text file

has been *scanned* into a stream of tokens. That stream of tokens has now been *parsed* into sentences of token relationships. The parsing step can do a couple different things with these sentences, but commonly it outputs an AST.

A simple AST is constructed of three types of nodes, declarations, statements, and expressions.

- **Declarations** - something that state the name, type, and value of a symbol. For assembly programmers, literals, labels, and variables are forms of declarations.
- **Statements** - indicate an action to be carried out that changes the state of the program. Language constructs such as loops, conditionals, and returns.
- **Expressions** - are combinations of symbols and values that can be processed and reduce to a value. Some higher level languages can have expressions produce side effects that change overall program state during expression reduction as well.

If one desired to perform the program computation itself, one has to perform a post-order tree traversal tracking the change in value of an expression at each operation step. For assembly, the types of sentences that need to be represented are comparatively limited, with instructions being the primary concern. We can further form AST nodes based off each operation and its immediate operands. Since assembly is line based, we can presume that each instruction block typically follows the preceding block. Here is where considerations for conditional logic and jumps come in. Our AST must branch when conditional logic occurs.

2.2.5 TYPE SYSTEMS AND SEMANTIC ANALYSIS

After an AST is emitted, **semantic analysis** is performed on the AST to ensure that the program follows the rules of its underlying type system. A type system and

its associated programming language have a few axes to distinguish themselves. Commonly, these are safe/unsafe, static/dynamic, explicit/implicit.

Safety from a type system is determined based off of what one could theoretically *do* with a valid program in that language. C is unsafe and allows for arbitrary manipulation of program data. Safe languages have checks that prevent a program from executing unpredictably [51].

State of type checking is another common axis. **Statically typed** languages perform all type checking at compile time, while **dynamically typed** programming languages are able to do type checking during runtime. For compiled languages, this means that after compilation, all type checking has been complete and type information can be discarded in the machine code. For dynamic languages, the type information is available during runtime and is used whenever a variable is interacted with [51].

Inference. Type systems commonly can have either **explicit typing**, where the code and programmer directly explains the types of variables and other program constructs. In an **implicitly typed** system, the compiler (or interpreter, as interpreted languages have type systems too) *infer* the type of variables and expressions as much as possible. Implicit typing allows for more terse code, but allows for a program to operate in undefined behaviour if the inferred type the compiler settles on is different from the type the developer believes is being used [51].

Refer to Figure 2.6 to see a loose breakdown of various programming languages and where their type systems fit on these various axes. Many modern, newer languages fall into the **safe, explicit, statically typed** category. Multiple languages also have support for both explicit and implicit typing, such as C++ after 2011 with the use of `auto` [25], Python after 3.5 with type annotations [59], etc. Unsafe languages do have their purpose, since the possibility space of legal programs

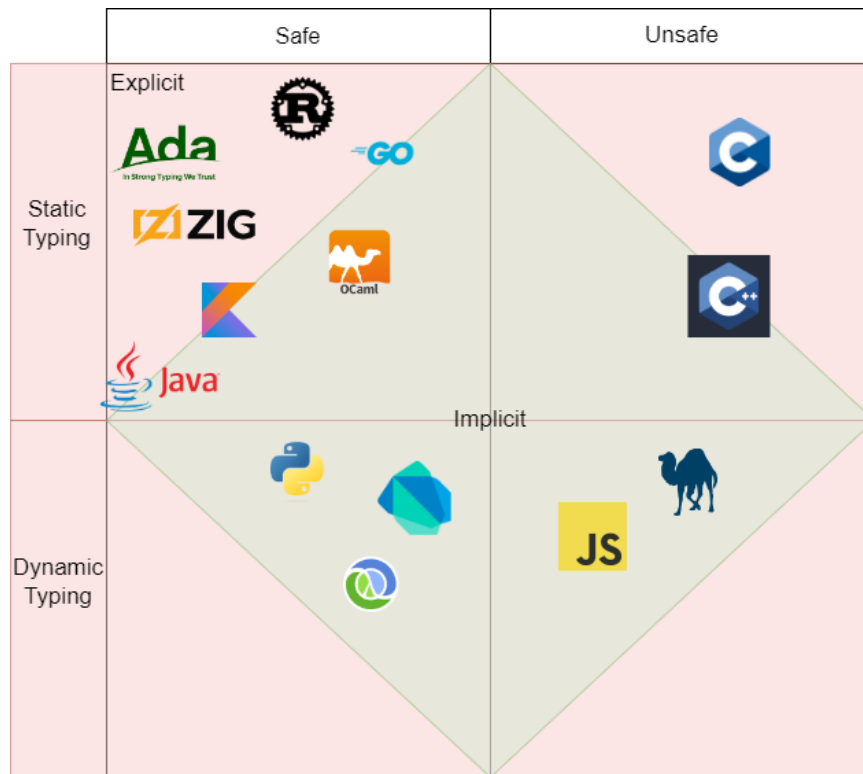


Figure 2.6: 3 Dimensional Table of Type Systems of Various Programming Languages

able to be written is larger than that of a safe language. For example, the ability to perform raw point arithmetic has its uses for various embedded and operating systems level tasks. These operations can be expressed in an unsafe language while the programs' memory ambiguity makes it more difficult to write in a safe language. Unsafe code gives the developer much more control over the underlying system since it is more "hands off". This means that an unsafe language trusts the end-developer to not make mistakes that would cause issues. These unsafe systems are commonly useful in specific technical situations where 'hacky' solutions are desired, for performance, readability, or various other reasons [42, 51, 4].

While semantic analysis is a large step within compilers and where many programming languages make themselves unique from the many thousands of others, it is not the main goal of this paper. Assembly language is by definition an unsafe language, and typically contains *no typing* at all. Fear not, since the outputted

assembly is coming from a compiler, we don't need to do any semantic analysis on our ARM assembly abstract syntax tree.

After semantic analysis in the compilation process comes intermediate representations and possible optimizations associated with IR.

2.2.6 INTERMEDIATE REPRESENTATIONS AND CONTROL FLOW GRAPHS

Semantic analysis is an important step in the compilation for higher level programming languages to lower levels, however it does not significantly alter the underlying representation of the parsed code, it may still be in the form of an AST. Optimizations are an immensely broad step within a compiler, however they are by no means the focus of a simple analysis tool. For posterity's sake, the modern optimization approach uses various modules that apply different optimizations on a program's intermediate representation (IR), ideally where the optimizations are applicable in any order. After all the optimizations permitted by the compiler (and compilation flags), the IR is finally emitted as assembly code.

Of particular interest to us is the **intermediate representation** of a program. Many types of IRs exist, one of particular renown is the Low Level Virtual Machine (LLVM) toolchain's IR [20]. Abstract Syntax Trees are quite powerful representations of a program, but they are bloated with significant information that is not immediately important for optimizations.

A common intermediate representation form is that of a **control flow graph** (CFG). Control flow graphs are more broad than that of an AST, as they represent a high level structure of a program through the medium of a directed graph that may or may not be cyclic. Each node of the graph is a **basic block** that consists of sequential statements. Each edge of a graph represents a possible flow between basic blocks. Based off these rules, a CFG is able to better represent the difficult constructs that an AST cannot encode well: conditional and looping constructs.

Conditionals result in branches in the graph, while loops are represented with backedges. A CFG imbues further structure compared to an AST. For example, a looping program in a CFG connects nodes in the loop sequentially with a backedge at the end of the loop, while an AST would just contain all nodes as direct children [51].

Given the construction of a CFG from an assembly representation of code, we can start to expand upon the types of programs we can provide an estimated energy cost for to programs that include conditionals, loops, recursion, etc. Prior to further SEA development, consideration must be given to all the computer hardware that sits between a program and its execution, and all the additional hardware and processor level optimizations that end up making a program execute not as one would initially expect.

2.3 MODERN CPU COMPLICATIONS

Taking a step away from ARM as a platform, there are hardware processes in modern computers could have an effect on the accuracy of a SEA. While some of the following examples discuss arm specifically, some other sections are more applicable to CISC architectures instead.

2.3.1 RUNNING A PROGRAM

Assembly code output from a compiler is quite close to its final machine code, only missing small pieces of information that give context as to where the executable program fits into the computer at large. The addition of these minor parts does not significantly change the assembly code, so from ASM one can trace exactly how a program executes. When a program is executing, it is contained in the operating

system in various segments, typically thought of as the text section, data section, the stack, and the heap.

The `.text` pseudo operation in an ARM assembly file is used to signify the text segment of an executing program [42]. Assembly code functions in an environment that has a few general purpose registers to use. For 64-bit ARMv8, there are the 16 32-bit registers as well as 32 64-bit registers that reference the same location in memory, but just more of that memory. For example, `w1` and `x1` are 32 bit and 64 bit registers that are stored in the exact same location, just with `w1` missing the additional 32 bits accessible by the `x1` register, as seen in Figure 2.7.

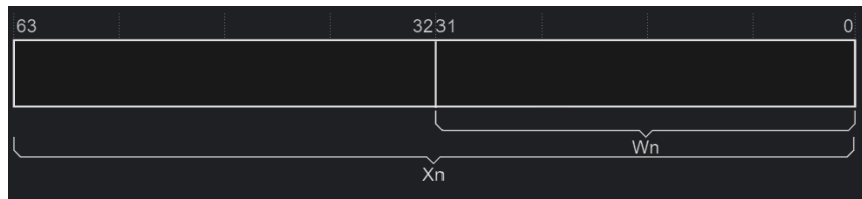


Figure 2.7: 32 and 64 bit register structure in ARMv8

Without getting too sidetracked into exact details of how assembly language operates and constrains within the system, the primary goal here is to understand that in running a program, an operating system allocates space for the program, which the program operates from when executing. During execution, the program flows line by line while jumping to various code sections based on current registers and execution mnemonics [6]. X86 platforms function the same way, just with possibly more complex individual instructions being executed.

The crucial point is that **operating systems** could exert control over how a program executes that the developer is not considering. When doing dynamic (run-time) analysis, perhaps other parts of the OS interfere with how much additional memory the program is able to use, force its process to be scheduled poorly, etc. These effects are not a concern for a SEA energy cost estimation, but could effect empirical test bench data.

A simple form of an energy analyzer would be to count the number of instructions that occur during execution, and then multiple each instruction by an estimated average energy cost of that instruction. Adding this all up, we could theoretically get an energy prediction for a program that minimizes the noise of background processes [28]. However, some factors can change the actual amount of energy consumed by executing a sequence of machine instructions that can be thought of as assembly code.

2.3.2 CACHING

Cache models have been a part of computers since the 1960s [13]. Caches are hardware constructs, typically within the CPU itself that serve allow for significant speedups. The methodology behind caching is to store information that a program may want to reuse so that there is not a delay from fetching the information from memory. Cache existence and configuration differs drastically by hardware, however industry trends exist. Cache solutions usually exist in **cache heirarchies**, as seen in Figure 2.8.

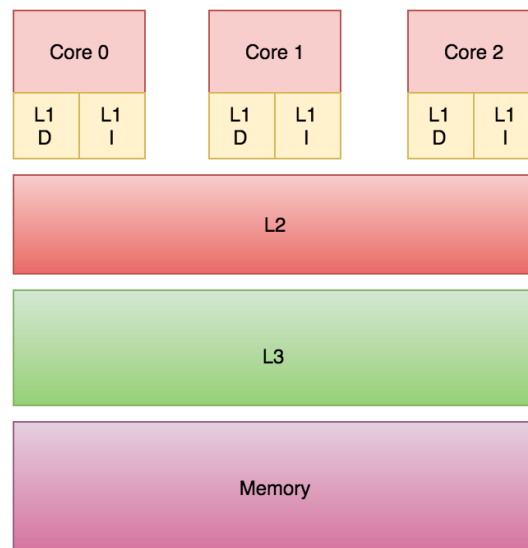


Figure 2.8: Cache Heirarchy Model of Modern Processors

Practically all modern processors have L1 cache, which is sometimes split into L1 D and L1 I portions, for data caching and instruction caching, respectively. Increasing in scope, most computers also have an L2 cache that has additional space to store more data than the L1 D cache, but is physically further from the processing core and thus is slower than L1. L3 cache is a further step from level 2. Some modern computers do not have L3 cache, and some specialized computing devices have additional caches beyond L3. All of these caches are checked prior to finally reaching out into regular RAM (called ‘memory’ in Figure 2.8) for the information needed. A cache is a small storage of data or processes determined to be useful for the program in the future. Modern computers are incredibly fast, so much so that waiting for data from memory can be a non-trivial slowdown to computation. Caches alleviate this by keeping a curated amount of pertinent data in a place physically closer on the hardware, which is also more accessible through higher bandwidth channels than RAM. Each level of cache gets slower to access, but also larger in the amount of things it can store. Algorithms for how to choose which information is important to store in a cache are beyond the scope of this work. These speedups caches allow mean a computer is commonly checking if certain data is stored in a cache, but as programs are easily larger than cache sizes, it is possible that *cache misses* occur, which is when a cache is searched for an item that is not found, so instead the computer slows down to search the next level of cache (or regular memory). Cache misses are one metric for analyzing performance of programs that is based on understanding and optimizing a program to minimize reaching up and beyond its caches.

In the context of constructing a SEA, cache models introduce two potential problems. When trying to benchmark individual assembly instructions during runtime, how much caching should we expect the computer to use? If the testing software is written to maximize cache misses, then the average energy cost

per associated instruction may be a significantly higher worst-case bound than desired, if the goal is to compute an “average expected energy” cost per instruction. Conversely, if the test assembly program utilizes *too much* cache, it could make the end instruction cost estimate naively optimistic. This poses a general question in the realm of static analysis, can cache hits and misses be predicted? Cache models and computer hardware engineering are growing increasingly complex as a way to extract performance improvements from nothing, however this complexity makes compile-time inference incredibly difficult to impossible. Therefore explicit considerations for caching are not a part of this project because proper cache systems are beyond the scope of this work and a “normal” amount of cache misses will be implicitly included in the instruction energy cost estimations.

2.3.3 PIPELINING

Modern computer processors have *pipelines*. The length of the pipeline differs based on architecture style (RISC/CISC) and individual hardware [36]. Each machine instruction can be broken down into different stages of execution. While pipeline stages differ depending on any number of factors, a simple pipeline could consist of 4 stages: fetch the instruction, fetch the operands, execute, write results. For example, a simple pipeline attempts to build up to instruction flow capacity such that all 4 stages of instruction execution are happening in parallel, with each step corresponding to a different instruction, as seen in Figure 2.9 [37]. This pipelining allows for instructions to be ready to execute as soon as a prior instruction finishes execution, compared to having to wait for fetching of the instruction and operands first. One could envision pipelining as splitting a processor into parts equal to the number of stages in the pipeline, and then executing the various stages of different instructions in parallel.

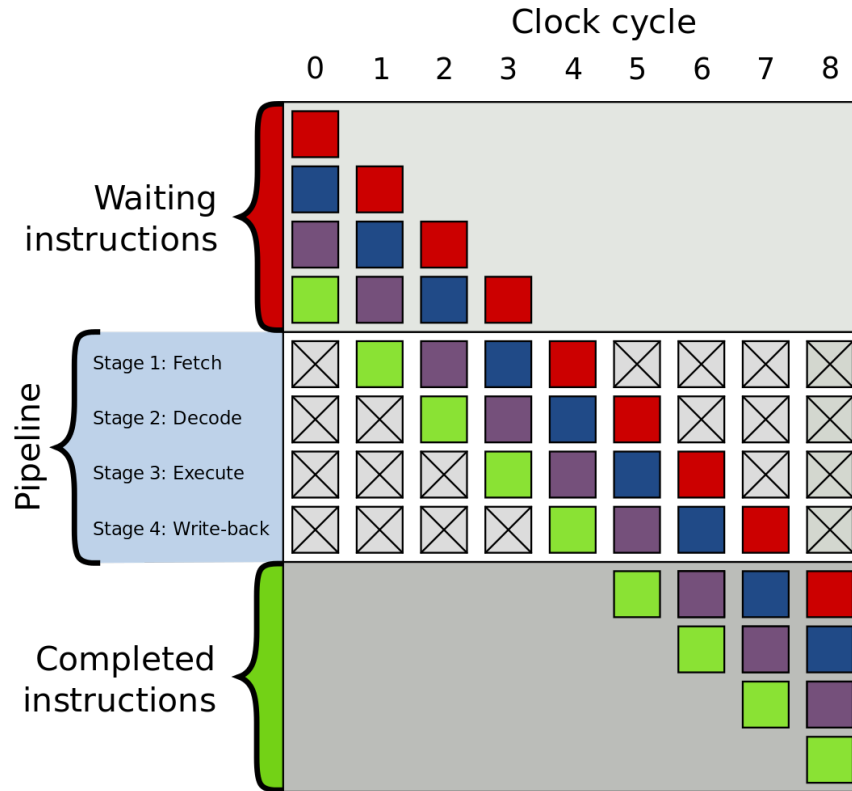


Figure 2.9: Generalized 4 Stage Processor Pipeline

The classic RISC pipeline is broken down into 5 sequential stages: Instruction fetch, instruction decode and operand fetch, execution, memory access, register write back [37]. A full pipeline typically implies a *faster* processor, as instructions can generally be executed faster if all parts of the processor are being used at any moment. Alternatively, *non-pipelined* processors, sometimes referred to as multi-cycle processors, carry out individual instructions from start to finish (and then begin carrying out the following instruction) [37]. Nearly all modern day CPUs have pipelines of various length due to obvious benefits of pipelines such as faster execution, reducing duplicate work, and allowing for various optimizations [46, 21]. The additional effort required to control the steps of the pipeline does increase circuit complexity, but execution speed is increased significantly to more than offset increased energy usage for a pipeline [37]. Thus, pipelining is a nearly ubiquitous

in modern chip design. Similar to caching, pipelining is something that is worth knowing exists, but its fine tracking and accounting for in this project is not a focus.

Nearly all modern computers have some form of pipelining, so another consideration is understanding how pipelines affect energy consumption based on the programs that are being fed through them. Pipelined processors are at their best when they are performing the preparation stages for incoming instructions while executing current instructions. A hidden assumption in most underlying models of assembly code is that each instruction is completed before the next instruction executes. For pipelined processors, this is not true, and can lead to the rise of **hazards**. Hazards are issues that arise when the following instruction in the pipeline cannot be executed in the next clock cycle [36]. Hazards are broadly split into three possible categories: data, structural, and control.

- **Data hazards** occur when upcoming instructions depend on the data of prior instructions within the pipeline. This hazard is analogous to a *race condition* in multi-threaded programming, where different parts of the processor (“threads”) are attempting to interact with the same underlying register (“data”) [36].
- **Structural hazards** occur when multiple pipelined instructions are attempting to use a shared processor resource (or structure). For example, if multiple instructions were entering the execution phase at the same time but all attempting to use a singular Arithmetic Logic Unit [36].
- **Control hazards** occur when a pipeline pre-emptively follows an incorrect control flow that means resultant computation has to be ignored. These often arise from incorrect branch prediction [36].

Many approaches exist to deal with hazards, but the most simple is the concept of **bubbling**. Bubbling is an action performed by the control logic unit on a processor.

During the initial stages of a pipeline, control logic checks if the inputted instruction could lead to a hazard occurring. At this point, control logic will buffer the pipeline with nop (no-operation) instructions such that hazards are avoided. This does *stall* the pipeline, which causes a program to take longer to execute, as the computer is not operating with a filled pipeline, instead being filled with “bubbles” [36]. Data hazards can be resolved through various means, some of which increase latency (such as bubbling [36]), some of which attempt to avoid needing bubbling (such as out-of-order execution or operand forwarding [11]). The specific architectures behind these different approaches and their effects on energy consumption is not a focus of this thesis, but it is important to understand that certain parts of programs lead to the rise of data hazards, which generally slow the processor down, causing longer execution time and increased energy usage. Alternatively, given lucky branch prediction and out-of-order execution, some programs may be computed non-trivially faster than expected. Anecdotally, programs with a high degree of possible branches will perform significantly better on processors with branch prediction after executing a couple times, as branch prediction caches adapt to how the execution actually occurs. For testing purposes on a computer with branch prediction, consideration to avoid unrealistic real world performance can be achieved through running different tests in sequence, instead of repeating an individual test multiple times [50, 11].

2.3.4 BRANCH PREDICTION

Given a long enough pipeline, **branch prediction** is a desirable feature for a processor to perform. ARM processors have comparatively short pipelines (4-6 stages [47]), compared to typical desktop computers with X86 architecture that have longer pipelines, typically in the 14-20 stage range. One other specification of modern computers are processors considered to be “superscalar”, allowing for

multiple instructions to be executed in parallel. These processors have quite long pipelines to manage synchronicity of the multiple instructions in one cycle[48]. The length of the pipeline will affect how much of a slowdown bubbling introduces, the longer the pipeline, the longer the processor must stall before resuming normal execution. In an attempt to lessen the need for bubbling the pipeline, when a branching construct appears in the program (such as an `if` statement), the processor will follow some of the overall possible branches until it is certain about the outcome of the `if` statement (generally when the branch conditional instruction exits the pipeline). If the chosen branch was correct, the processor can continue executing as normal. If the predicted branch was *incorrect*, the processor must instead throw away all computation after the faulty prediction and begin with an emptied pipeline on the correct branch (typically through the use of a process called *pipeline flushing* [36]).

When attempting to analyze branch prediction and branching behaviour from a static analysis standpoint, a few approaches exist, with varying levels of optimism. Static estimations range from assuming perfect branch predictions to absolutely imperfect branch predictions. Generally, the variance from branch prediction accuracy is a relatively tight band, as pipeline length is nearly never comparable to total instruction count. For example, a 10,000 instruction program and a 30 stage pipeline that gets invalidated 10 times is, at worst, 300 wasted instructions, which means pipeline flushed wasted only 3% of overall instructions. Now consider that branch prediction is surprisingly accurate, and pipelines are usually much shorter than 30 stages. Hence, the “only” difference in a pipeline stall is a pipeline flush at each failed branch [11]. From an energy analysis standpoint, branch prediction accuracy is yet another complication that effects energy estimations of a program. Since ARM processors have comparatively short pipelines compared to their CISC style counterparts, many ARM processors *do not* support branch prediction [6].

Thus, when developing a SEA, additional consideration should ideally be given at branches for forced slowing of the pipeline, but the increased effort of pre-traverse a decision tree determining branch prediction costs [11] is beyond the scope of this work. One other general solution to certain hazards in a pipeline is through **out-of-order execution**, which some ARM processors support [6]. Out-of-Order execution allows some data hazards to be avoided by reordering their execution sequence. CISC superscalar processors do experience performance speedups from out-of-order processing, but this does come at an increased energy cost due to necessary control logic [56]. However, out-of-order execution is not a main concern for this work as it does not significantly effect an energy estimation for a typical ARM program due to the testbench's ARM architecture which does not benefit significantly from out-of-order execution.

2.4 PROGRAM ANALYSIS

After modern software is written, a large portion of it's lifecycle is spent in maintenance. To perform various upgrades, optimizations, and changes on an existing system, the program is analyzed with a specific goal in mind. A different goal will effect the style of program analysis that is performed. The broad, applied goals of program analysis are for either program *optimization* or *correctness* [60]. In simple terms, existing code is typically analyzed to *increase* performance or *fix* existing bugs. For example, the algorithms within programs are commonly analyzed for asymptotic efficiency represented with Big-O notation as an analysis tool [15]. In addition to amortized computational time, memory utilization is another axis upon which programs are commonly analyzed [15]. These results of these analytical methods can then be applied to help optimize programs. Alternatively, for extremely important code such as that being run on potentially dangerous vehicles (airplanes,

power grids, etc.), programs are analyzed with a goal of verifying that no issues are possible during runtime. One step closer to computer programs themselves lives a smaller field of research exists around the concept of Worst Case Execution Time (WCET) that looks for estimating practical execution time limits of a given program. As one descends toward real-world program analysis based on practical implementation rather than the theoretical field of algorithm analysis, assumptions can be made that asymptotic models cannot. Most program analysis techniques spawned as ways of verifying properties about production software [60]. Significant difficulty appears based on the *rigor* that some techniques desire because of two findings in computer science, **Rice's Theorem** and **the Halting Problem** [60, 54, 34]. Further discussion about the problems these two findings create is discussed in Chapter 3. One approach to account for these theoretical problems is to tie the analysis directly to the real-world operation of the program, by analyzing a program during runtime.

2.4.1 STATIC VS. DYNAMIC

Dynamic analysis deals with understanding a program during its runtime. Also referred to as “profiling”, dynamic analysis aims to further understand how a program operates by integrating itself into the running program or through running in parallel to the program in question and polling operating-system level channels. Regardless of how a dynamic profiler interacts with an executing program, it *is* interacting with a program during runtime. Although it can be minimized, dynamic methods will *always* have some form of overhead. These additional costs, when analyzing for execution speed or memory utilization, can typically be waived as trivial compared to the executing program. Unfortunately for the aspiring energy aware developer, any other program running in the background (such as a theoretical dynamic energy analyzer) takes energy itself to run. Hence, an energy

analyzer running within the program under analysis feeds into its own sample data, entangling real results with noise.

A simple solution for energy analysis is then to remove the energy reader from the execution system itself, and instead place a medium (typically a multimeter or other instrument) in between power source and computer to get a more accurate reading. In the test data collection phase of this thesis, there are processes that must be done dynamically. During this phase, a separate hardware reader approach will be used (as described in Chapter 4). Some computer energy analysis approaches further expand on the separate sampling hardware approach by attaching multiple sensors to various parts of the hardware under investigation (as opposed to one meter sitting between the power supply and computer itself). These approaches show a further breakdown of energy consumption on a per component basis [28]. Within the context of a static energy analyzer, provided all background software is minimized, component level granularity of sampling is not necessary for this work as the goal is overall program energy usage estimates.

Static program analysis refers to methods for understanding a computer program without running the program. Typically static analysis is considered to be done at 'compile' time. Some static analysis methods are statistical observations about the program, such as comment density, cyclomatic complexity, dependency counts, etc. Comment density, necessary dependencies, etc. can all be found relatively trivially by observing the underlying source code to a program. Static analysis can be more involved, as even though the program being analyzed is not being run, one can use nearly arbitrary processing and computation in the analysis phase. Since static analysis does not have access to the additional information about the runtime of a program, it utilizes alternative assumptions to account for the theoretical issues with program analysis.

Another form of static analysis technique is that of **data flow** analysis. Data

flow analysis attempts to determine run-time information about a program *without* running it [60]. This is commonly done through constructing some abstract form of a program that encodes the possible flows of information with a control flow graph. Then, **control flow paths** within the CFG can be explored to find worst-case execution time (WCET) of a program. Control flow paths are difficult to follow explore, as CFG models can easily allow for infinite paths from non-determinability of a static algorithm. In this region lies the area of termination analysis, where a program is statically observed with the goal of detecting if it ever terminates. It has been mathematically proven that it is impossible to arbitrarily determine if a computer program terminates (or ‘halts’) [54]. However, a nontrivial subset of programs can be statically analyzed for termination with various methods, typically utilizing the concept of a *loop invariant*, a way to represent if a loop terminates.

PROGRAM COST ANALYSIS

This section discusses methods for program analysis. An emphasis is put on automatic complexity analysis, as that is the static approach used by the SEA. A discussion of the field of recurrence relations in mathematics is introduced, as well as the concept of *cost relations*. Explanations about control flow graphs and approaches for extracting cost relations from an abstract program representation are provided. After discussion of cost relation extraction, I further explain the process to find closed forms of cost relation systems and re-applying cost expression contexts for a final “total energy cost” result.

3.1 RECURRENCE RELATIONS

For many programs and sequences, it is beneficial to have tools to model them *recursively*. In mathematics, a **recurrence relation** (RR) is simply an equation that defines a sequence based on a formula that references prior terms in the formula. The simplest example of a recurrence relation is the Fibonacci Numbers. As a

recurrence relation, it is defined as:

$$F(0) = 1, F(1) = 1, F(n) = F(n - 1) + F(n - 2)$$

This system for representing recurrence relations can encode much more complex systems. Recurrence relations have an *order* equivalent to the furthest back term that the formula references. For example:

$$R(n) = 3R(n - 1) + \frac{R(n - 2)}{2} - R(n - 5)$$

is a recurrence relation with order 5. *Homogeneity* of a RR is determined from the existence of a non-recursive term. For example,

$$R(n) = 2R(n - 1) - f(n)$$

$$\text{Homogenous} \iff f(n) \equiv 0$$

Linearity of a RR is an attribute determined by if any recursive terms are being operated upon non-linearly.

$$R(n) = \sqrt{R(n - 2)}$$

is nonlinear, as a recursive term, $R(n - 2)$ is being square-rooted. RRs are very powerful and have a multitude of applications in applied and higher math. Of importance is that some special forms of recurrence relations can be translated into

a *closed (discrete)* form, removing all recurrence terms. For example,

$$\begin{aligned} R(1) = 4, R(n) &= 2 * R(n - 1) + 3 && \text{Recursive} \\ &= 2^{n+1} + 3(2^{n+1} + 1) && \text{Closed Form} \end{aligned}$$

is a linear, homogenous, first-order recurrence relation that has a discrete formula. How closed-forms of RRs are extracted differ by the order, (non)linearity, and (non)homogeneity of the RR. Analytical approaches for determining closed forms are beyond the scope of the paper. Computer algebra systems (CAS) are utilized for more complex recurrence relations as they can better handle complex logic processes for finding closed forms of certain types of RRs. Not all recurrence relations have simple closed forms, such as one closed form representation for Fibonacci numbers.

$$\begin{aligned} F(0) = F(1) = 1, F(n) &= F(n - 1) + F(n - 2) && \text{Recursive} \\ &= \frac{1}{\sqrt{5}} \left(\frac{1 + \sqrt{5}}{2} \right)^n - \frac{1}{\sqrt{5}} \left(\frac{1 - \sqrt{5}}{2} \right)^n && \text{Closed Form} \end{aligned}$$

For the purpose of this project, I presume that some recurrence relations are not able to be translated to a closed form.

3.2 COST RELATIONS

The execution cost of computer programs can also be modeled using “recurrence relations”. For a simple program that imperatively executes various assembly instructions, no recurrence is necessary, we can instead model the execution cost as a direct sum of all the instructions in addition to various constant energy consumption

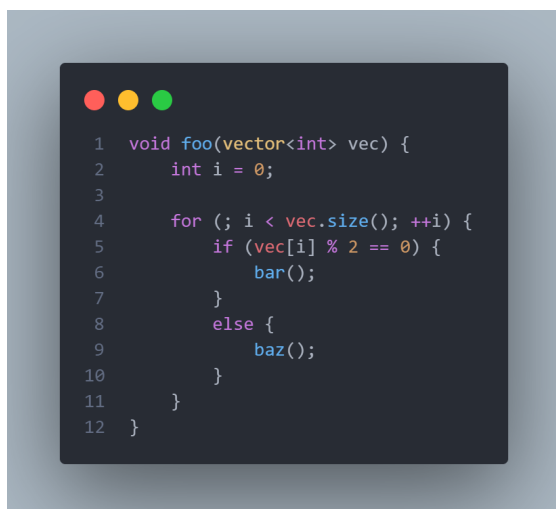
values (discussed in chapters 2 and 4). Nearly all programs are not as simple, instead with various control flow constructs such as loops, function calls, etc. Recurrence relations can model these constructs, especially if their definition is expanded slightly to a related concept of **cost relations** (CR). These cost relations are used to compute a *cost* as a primary goal. Recurrence relations serve as a way of expressing a reliance on prior information.

As an example, the cost relations for a simple function `foo()`, which can be seen in code and control-flow graph from in Figures 3.1, 3.2, are as follows:

$$\begin{aligned}
 (a) \quad C_{foo}(v) &= k_1 + C_{for}(vec, 0) && \{v \geq 0\} \\
 (b) \quad C_{for}(v, i) &= k_2 && \{i \geq v, v \geq 0\} \\
 (c) \quad C_{for}(v, i) &= k_3 + C_{bar}() + C_{for}(v, i + 1) && \{i < v, v \geq 0\} \\
 (d) \quad C_{for}(v, i) &= k_4 + C_{baz}() + C_{for}(v, i + 1) && \{i < v, v \geq 0\}
 \end{aligned}$$

where i, v are the counter variable and vector length, respectively. Then $C_{foo}(\dots), C_{bar}(), C_{baz}()$ represent the costs of executing their respective methods. $k_{1..4}$ are constant costs that are a part of each equation. The $C_{foo}(\dots)$ block is split into a ‘for’ loop C_{for} , and the declaration and initialization of the counter variable i . In the above cost equation, k_1 accounts for the cost of creating i . Equations (c) and (d) account for the costs of the branches created by the if statement contained within the loop. Since this is a static model, it is impossible to know the actual element values of vec , so the conditions for (c) and (d) are the same, even if only one will execute per iterations [4]. A static analysis for the worst case execution time of `foo` is thus:

$$C_{foo}(v) = k_1 + v(k_2 + \max(k_3 + C_{bar}(), k_4 + C_{baz}()))$$



```

1 void foo(vector<int> vec) {
2     int i = 0;
3
4     for (; i < vec.size(); ++i) {
5         if (vec[i] % 2 == 0) {
6             bar();
7         }
8         else {
9             baz();
10        }
11    }
12 }

```

Figure 3.1: Simple Foo() Method in C++

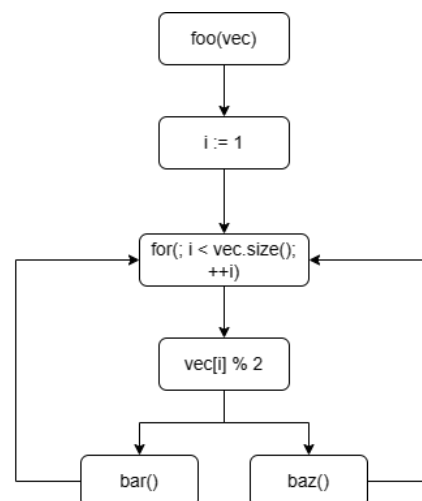


Figure 3.2: ARM ASM from Figure 2.3

Which one sees is a *closed* form of the input cost relation system, with all recurrences removed. If $C_{bar}()$ and $C_{baz}()$ had additional nesting and logic within them, the process of conversion to a closed form will propagate throughout all parts of the program representation until a fully closed form is reached, or it is determined a closed form for the cost relation system is impossible with current methods. This is only a simplified version of CR analysis, where program scope is trivial enough to extract CRs manually as well as find a closed form intuitively. Formal approaches are discussed further in this chapter.

As seen in the example above, CRs have syntactic similarities with recurrence relations. Cost Relations are *systems* of equations that may reference themselves, like recurrence relations. CRs have *bounds* that can define different behaviour. RRs similarly have bounds, with simple RRs generally being bounded by $n > 1$, as the few terms in the recurrence relation are defined statically, such as $R(1) = c$. While CRs can be represented with syntax congruent to RRs, they can differ enough that most RR methods are not applicable to CR solving.

- **Non-determinism** of cost relations. Systems of cost relations can reference themselves in ways that recurrence relations do not. CRS can represent *non-halting* problems. If the underlying programming language that is being

represented by CRs is deterministic, the levels of abstraction that are required to represent a program statically can introduce non-determinism. Since arguments can only be observed by their *sizes* instead of underlying values, this *size abstraction* can cause a cost relation system to not terminate [4].

- **Imprecise constraints** due to the possibility of needing to encode non-trivial data structures as parameters. In the simple example, a vector is a linear structure that can be sized simply by the number of elements within it. For example, if instead of a vector, consider a CR system that is being built to model a program traversing a graph or tree. It is difficult to predict the number of elements in each structure, and thus also difficult to predict the amount of elements to reduce by at each step. In the vector example, each step reduces by one element. Given an unbalanced binary tree, representing the reduction in search space at each step by a cost relation precisely is impossible [4]. This issue pushes many CR analysis systems to constrain themselves to Worst-Case analysis, as then more information can be inferred about worst-case size reduction at each step. For example, a cost-relation modeling searching for an element within a binary search tree. Statically, one cannot predict how the actual tree is balanced or where the search query is when the program is running. Due to the fundamental structure of a binary search tree, one can always know that at worst, stepping towards one node removes at worst one element from the search.
- **Multiple Arguments** of cost relations. CRs very commonly depend on several arguments that can move in multiple different directions. This means that the number of times a relation recurs is a combination of multiple of its arguments [4]. For a simple recursive representation of a loop, that recursive equation is iterated n times, where n is the size of the loop. For automatically constructed

cost relations for a program, the underlying code will likely be more complex than a simple loop, instead having flow dictated by multiple variables. A combination of all of these variables is then able to be used to infer how many times a cost equation is ‘called’ in a static analysis of the program. In terms of recurrence relations, this means that many cost relations are **not primitively recursive**. Primitive recursion means that a recurrence relation has a direct translation to an iterative construction. A recurrence relation that is also not primitively recursive is the famous **Ackermann Function** [2].

Hence, the methods of solving RRs differ from the methods of solving CRs. Some dimensions of further complexity in RRs exist that cannot occur within CRs, such as polynomial coefficients to recursive calls. More complex recurrence relations and closed form discovery methods are not a focus of this work, aside from their relevance and applications to cost relations.

An influential framework presented in the 1970s by Wegbreit for automatic cost analysis splits the analysis task into two parts [58]. First, a computer program, along with an associated **cost model**, is statically analyzed to create a system of cost equations (a cost relation) [58]. Sometimes systems of cost relations are wholly referred to in the singular, in the context of “A program and its associated CR”. A CR is not useful for automatic analysis, since the possibility of recursion within the system makes analysis quite difficult. Second, the CR must be converted to a *closed form* representation. As this conversion is happening statically, the assumptions that are used generally result in the static, closed form of a CR to be an approximation. The **worst-case** bound is the most common approximation, as pessimistic estimations are preferable to naïveté [58, 4]. However, different assumptions in the closed form construction step will lead to different estimations.

Cost relations need a fundamental **basic cost expression** to construct more

overall cost equations that accumulate an overall cost. The actual values and meanings of these basic cost expressions can change, which change the resulting CR. Some common “costs” that are chosen for programming include the number of numerical comparisons [58, 53], heap allocations [4, 5], or bytecode instructions [4]. Formally, a basic cost expression is produced according to the following grammar [4]:

$$\begin{aligned} \text{exp} := & r \mid \text{nat}(l) \mid \text{exp} + \text{exp} \mid \text{exp} * \text{exp} \mid \text{exp}^r \mid \\ & \log_n(\text{exp}) \mid n^{\text{exp}} \mid \max(S) \mid \text{exp} - r \end{aligned}$$

Where $r \in \mathbb{R}_+$, l is a linear expression, S is a non-empty set of basic cost expressions, nat is a function defined from \mathbb{Z} to \mathbb{Z}_+ by $\text{nat}(v) = \max(v, 0)$. Note that this definition means that a basic cost expression *must be positive*, both by the definition above and by intuition. Since the goal of this analysis is to produce a predicted *cost*, the existence of actions that come at a *negative cost* is impractical. In the most minimal case, an action will have a cost of 0, but not negative. Allowing for negative cost expressions is nontrivial as it invalidates many presumptions that further analysis is constructed upon.

Basic cost expressions are an abstract representation of the costs accumulated throughout the system in question. For an energy aware programmer, the actual *cost* that is being accumulated is energy. Connecting electrical consumption to program execution are machine instructions. Since assembly is more human readable version of machine code, *assembly instructions* are the basic cost expressions of the SEA implemented as a part of this project. After worst-case upper-bounds analysis, a closed form expression in terms of assembly instructions is produced, if possible. This expression is complete as is or can be condensed into a single energy estimation

cost through mapping assembly instructions to energy values through dynamic benchmarking. For the static energy analyzer presented here, the basic cost is an assembly instruction. However, the amount of energy consumed by a computer while it runs a program can not be attributed absolutely to just the assembly instructions of the program being executed. Some additional considerations to the overall energy cost are necessary, such as processor level complications (branch prediction, caching, out-of-order execution) discussed in Chapter 2 and system level considerations (operating system, background energy consumptions, etc) discussed in Chapter 4.

In the CR example illustrated by Figures 3.2 and 3.1, the associated CR was given. For simple programs, cost relational systems can be inferred, but the automatic construction of the CR is (unsurprisingly) involved.

3.2.1 COST RELATION EXTRACTION

Extracting cost relations from existing code has been demonstrated for various programming languages of different paradigms, such as functional, logical, and imperative. Cost relation systems are arguably easier to extract for higher level programming languages, due to the layers of abstraction that they encode allowing for an easier understanding of program control flow. However, the cost relation system of a higher level language is less useful for static cost analysis as the fundamental “cost” is messy. The same abstractions that make finding a representative CRS also mean that the most basic cost blocks are highly abstracted. These high level basic cost blocks are difficult to test for a specific cost individually. Therefore, the goal is instead to extract cost relations of a lower level language representation of a program that is close enough to machine execution to allow for individual testing of the pieces of the program, but still with enough information to allow for cost relations to be extracted in the first place [3].

While assembly does not follow any particular programming paradigm, it is most closely modeled by an imperative paradigm. The steps required to extract the cost relations of an imperative program are to translate the code into a control flow graph (CFG), perform intermediate flattening upon the CFG to find an intermediate recursive representation, infer size relations based on initial input calls, statically analyze approximations of rules from the intermediate recursive representation, and finally combine the prior steps to form a complete *cost relation* [3].

A control flow graph is a powerful intermediate data structure used in some compilers as a representation of program execution. The following theoretical approach utilizes one possible formal definition of a CFG that tracks information in a way that is helpful for cost relation construction. Alternative styles of CFGs exist, although their general concept is the same. To generate a CFG from corresponding imperative source code, one first identifies all the possible blocks within the source code. Each *block* is a node in the CFG, where a block is a 4-tuple of the following.

$$\langle id, G, B, D \rangle$$

- *id* a unique label or identifier to a block
- *G* is the “guard” of a block that contains conditions that must be satisfied for the block to be executed
- *B* is a sequence of imperative instructions that are presumed to all be executed if the block executes
- *D* is a list containing all possible successors to the current block. Multiple successors may exist in the case of conditional control flow such as loops, function calls, recursion, or jumps.

In this case, B and id are comparatively trivial objects to construct, but determining guards G and successors D is more involved. In the context of assembly, a large portion of mnemonics have exactly one successor (which means they can be combined into one overall block). For this project, the main structure that could introduce multiple entries to a block's successor list D are that of *branching*. In assembly, various mnemonics allow for arbitrary jumping to other portions of the source code execution. These jumps in ARM assembly can be unconditional, such as the mnemonics b (branch) or bl (branch and link). Alternatively, a whole family of conditional branching mnemonics exist as well, such as the family of $b.\langle condition \rangle$, where $\langle condition \rangle$ could be anything such as $EQ(=)$, $NE(\neq)$, $GE(>=)$, $LE(<=)$, $GT(>)$, $LT(<)$, etc. For a better reference on ARM mnemonics, refer to Appendix A. These conditions then could then contribute to the next block's guard clause. Further constructs for control flow exist within assembly, but they are beyond the scope of this work.

The next step in automatic cost-relation system construction is removing all iteration from the program and replacing it with recursive links in the CFG. Additionally, we need to flatten the operand stack such that all its contents are represented through local variables to each block. In each block we condense the possible stack variables into parameters of the current block to limit chances of exceeding stack size limits [3]. There are further optimizations and theory into how the process of this translation, but the general goal is to remove all global information in this imperative environment and have all information needed by a block passed to it by the prior in the CFG. This allows the control flow graph to be represented as a system of recursive equations.

After various reductions to the control flow graph, the next step is to infer *size-relations* at different points of the computation. When limited to static analysis, one is unable to know the underlying value of variables. However, it is possible

to understand *sizes* of variables. In higher level languages, this could be things such as the length of a collection, numerical constraints, term-depth, etc. In the context of ARM with the limited scope of this project, the primary size-relation under consideration is that of constraints on numerical variables. For example, a loop with a condition to loop *until* as certain value is too small allows for a size relation to be created for the looping variable. Before inferring size relations, we need to convert the recursive representation to contain *calls-to size-relations* between variables in the head and variables used in subsequent calls.

Definition 3.2.1 (calls-to size relations). Let \mathcal{R}_m be the recursive representation of a method m , where each rule takes the form $p(\bar{x}) \leftarrow G, \bar{B}_k, (q_1(\bar{y}; \dots; q_n(\bar{y}))$. The calls-to size-relations of \mathcal{R}_m are triples of the form

$$\langle p(\bar{x}), p'(\bar{z}), \varphi \rangle \text{ where } p'(\bar{z}) \in \text{calls}(\bar{B}_k) \cup \{\text{p_cont}(\bar{y})\}$$

which describes the size-relation between \bar{x} and \bar{z} when $p'(\bar{z})$ is called, where $\text{p_cont}(\bar{y})$ refers to the program point immediately after \bar{B}_k . The size-relation φ is given as a construction of linear constraints $a_0 + a_1v_1 + a_2 + \dots + a_nv_n \text{ op } \mathbf{0}$, where $\text{op} \in \{=, \leq, <\}$, and each a_i is a constant, and $v_k \in \bar{x} \cup \bar{z}$ for each k [3].

With this definition, the process of inferring size-relations is done through first reducing blocks into the linear constraints they impose upon variables, and then using a bottom-up approach from a *fixpoint* algorithm. Block reduction to linear constraints is done by abstracting guards and underlying operations into their effect on variables [3]. Fixpoint algorithms are beyond the scope of this work, but the main idea is that we utilize a fixed point in the recursive representation as an anchor and iterate upon that point until a target condition is satisfied.

Finally, to get a *cost relation system* (CRS) for the imperative source program.

The main idea is to combine information from the recursive representation of a control flow graph and additional information yielded from typical static analysis methods to get an optimal cost relation system for a program. The primary goal of this additional step is to detect what arguments in the recursive representation system are unnecessary for the cost relations. Given a block $Block_{id}$ in a control flow graph, represented by one recursive rule with no distinction between local and stack/global variables, the *cost* function for $Block_{id}$ is of the form $C_{id} : (\mathbb{Z})^n \rightarrow \mathbb{N}_\infty$. The minimization of the count n arguments is done using a few rules. Sometimes, arguments in recursive representations can be dropped because they do not affect overall program control flow. For example, consider a passed variable that is just an accumulating parameter of some kind. More generally, the process of determining an approximation of variables is a common static analysis problem that involves tracing data dependencies against control flow and a fixpoint process. There are more complex and precise reduction processes available, but since the primary goal here is removal of redundant variables (as opposed to redundant operations), more simple approaches suffice. As each recursive equation representation is equivalent to a block in the control flow graph, one can observe the unconditional instructions B in that block and all variables it interacts with in the process of finishing the block, and drop the unused variables. To get the desired cost relation (which is a system of cost equations) for a given $Block_{id}$, the main process is to create one cost equation that defines the total cost of sequential statements B in the block and the cost to the next block (call it p_cont for continuation), and another cost equation which sets the cost of p_cont as the cost of any other block with a satisfied guard. The continuation is a separate equation because it allows for more flexibility in possible alternative paths [3]. A rigorous mathematical representation of these cost relations exist, but is omitted as the extreme specifics are not of focus here. After all this, there will be a (likely sizeable) set of cost equations for the overall program

execution. These cost relations can then either be solved for dynamically (with inaccuracies due to measurement) or statically (with losses due to approximation) to allow for a final cost to be determined. It is worth noting that this framework of *cost* is flexible, as a ‘cost’ is any positively accumulating value that is trackable. In the case of this project, the goal is tracking energy consumption per instruction, where groups of instructions have different energy costs. Alternative cost systems exist, such as having every instruction cost ‘1’ unit to estimate overall computation time, or tracking only memory allocations to estimate memory usage cost.

3.2.2 AUTOMATIC UPPER BOUND INFERENCE

This is a highly mathematical summary of the methods being used by the PUBS solver [4]. Proceed with caution.

To perform significant transformation upon CRs, a more rigorous definition is first required. A **Cost Relation System** S is defined as a finite set of all equations of the form:

$$\langle C(\bar{x}) = \mathbf{exp} + \sum_{i=1}^k D_i(\bar{y}_i), \phi \rangle \quad k \geq 0$$

Where C and D_i are cost relation symbols, $\bar{x} \cup \bar{y}_i$ are distinct variables, and ϕ is a set of linear constraints over $\bar{x} \cup \text{vars}(\mathbf{exp}) \cup_{i=1}^k \bar{y}_i$ [4]. Informally, this is a terse way of expressing all possible equations that construct a full cost relation system. For any specific equation ϵ from the cost relation system S , to determine an exact cost of the program, begins with the value of the base cost expression \mathbf{exp} and then accumulates the cost of each step in the summation, where cost relations are explained by D_i , variables by \bar{y}_i , and associated linear constrains described by ϕ .

At each iteration of the summation, there may be alternative choices that change the future cost relations $D_{i+1}, D_{i+2}, \dots, D_k$. This means that *branches* can appear in the

summation during recursive evaluation. Thus, ϵ can be represented by an associated **evaluation tree**. An evaluation tree is a data structure that encodes how a high level expression is interpreted by branching at each operation. The associated evaluation tree for a specific ϵ thus represents all the ways in which the program could execute. By summing over all the nodes to the evaluation tree, a final cost of ϵ is collected[4]. The formal definition for the set of trees for an entire cost relation system is complex and detracts from the comprehensibility of this work. Given the set of all trees for a cost relation system defined by $Trees(C(\bar{v}), \mathcal{S})$, the *solutions* to a Cost Relation System are provided by:

$$Answers(C(\bar{v}), \mathcal{S}) = \{\text{Sum}(T) \mid T \in Trees(C(\bar{v}), \mathcal{S})\}$$

Where each tree represents one possible evaluation of an initial ϵ .

To get the worst-case cost of program, consider a $MaxCost(\bar{v}) := \max(Answers(C(\bar{v}), \mathcal{S}))$. Unfortunately, $MaxCost()$ is flawed. Turing proved that it is impossible for one computing machine to determine if another computing machine is “circle free” in a finite number of steps [54]. This finding is renowned as the **Halting Problem**, which shows it is impossible to determine if an arbitrary program terminates statically. For the evaluation tree set $Answers$, the trees could be *infinitely deep, infinite in quantity*, both, or neither. Thus, $MaxCost()$ is flawed because $Answers$ could likely be an infinite set. Additionally, many more computer programs are terminal but highly combinatorial. Hence, some evaluation tree sets may be finite, but still not computable in any legitimate timespan. The existence of these mathematical and computational limits does not mean this approach is worthless. Sometimes, infinitely many trees *can* be reduced to a finite upper bound solution. While infinitely

many evaluation trees exist, they are not always distinct, some infinitely numbered trees have a finite set *Answers*. The overall goal is *not* to iterate through an evaluation tree that follows one possible execution of a cost relation, but to find a **closed form**. An immediate challenge to a constraint based closed-form translation of *Answers()* is that many cost relation systems produce infinite trees due to imprecise constraints, ϕ . For example, in equation 3.2 of the cost relation example at the beginning of this section, i is constrained as $i \geq v$, where v is the size of an input vector. Although theoretically infinite choices for values of i exist in the evaluation tree expansion, the result is always only k_2 , a constant representation a basic cost expression.

Recurrence relations are generally single functions, that may have alternative logic based on values of the recurrence. Cost relation systems (CRS) allow for more interaction since they tend to be systems instead of individual functions. CRS are *composable*, meaning a single cost equation may be made up of not only itself, but other cost equations as well. This allows for the solving of upper bounds to be done by focusing on individual equations. If one equation has a reference to another CR, temporarily shift focus to that CR. Importantly, this compositionality only works well if recursions are *direct*. If cycles of recursion exist, compositionality breaks down.

$$D(x) = 10 + C(x - 1) \quad \{x \geq 0\}$$

$$C(x) = 11 + D(x + 1) \quad \{x \geq 0\}$$

For example, when trying to decompose the CRS above with an entrypoint of $D(x)$, one quickly observes that it is impossible to continually break down the system.

The decomposition would be as such.

$$D(x) = 10 + C(x - 1)$$

$$D(x) = 10 + [11 + D((x - 1) + 1)]$$

$$D(x) = 21 + D(x)$$

$$D(x) = 21 + [10 + c(x - 1)]$$

and so on...

While this is a technical constraint to this upper bound solving approach, it is also an unlikely issue in practice since the CRS systems modeling programs of interest are less likely to be so trivially infinite. For non-cyclic cost relation systems, eventually through repeated substitution one can reach a non-recursive (or *stand alone*) cost relation which is constructed entirely of basic cost expressions.

This approach of substituting sequential cost relations for a prior reference can also be thought of as trees, which is the evaluation tree approach discussed prior. **Evaluation tree approximation** is a common static analysis approach, where an evaluation tree is evaluated by their size, depth, and node count [4]. There are two types of nodes within an evaluation tree constructed from a CRS: *internal nodes* with recurrent references and *leaves* of stand alone cost relations. A common approach is to then count the number of leaves and number of nodes separately and multiply each by an upper bound cost for the base and recursive step. This approach is still nontrivial, as one still must perform internal and leaf node counting, as well as inferring a tight upper bound multiplier for both.

3.2.2.1 UPPER BOUND NODE COUNTING

For the evaluation tree approximation method, one needs upper bound counts for both $internal_+(\bar{x})$ (internal nodes) and $leaf_+(\bar{x})$. The approach presented in [4] finds the bounds on the *branching factor* b and *height* $h_+(\bar{x})$ of all trees related to x . Then one can compare the height and branching factor of the unknown tree to an existing, *complete* tree with the same height and branching factor to obtain an upper bound. Trivially, an incomplete tree is upper bounded by a complete tree of the same height and branching factor. To infer the upper bound of height, h_+ , one starts with any evaluation tree $T \in Trees(C(\bar{v}, \mathcal{S}))$. Consecutive nodes in any branch of T represent consecutive recursive calls that would occur in the evaluation of $C(\bar{v})$. From this, the bounding tree height can be reduced to bounding consecutive recursive calls during the evaluation of $C(\bar{v})$. To model consecutive calls, the concept of *loops* is useful even within cost relation systems.

$$\varepsilon = \langle C(\bar{x}) = \mathbf{exp} + \sum_{i=1}^k C(\bar{y}_i), \varphi \rangle$$

So, the loops on a certain cost relation C , is found as $Loops(C) = \bigcup_{\varepsilon \in def(S, C)} Loops(\varepsilon)$ [4]. Automatic termination analysis typically find an upper bound on the number of iterations of a loop exists by finding a function f within the loop's arguments to a *well founded* partial order, such that f decreases in any two consecutive calls. If a partial ordering can be found to show "progress" towards end bounds of a loop, then infinite loops are impossible (and thus, termination will occur). These

functions are called *ranking functions* [4]. Automatic termination analysis typically only aims to prove the *existence* of a ranking function, as the mere existence is enough to show that the process in question terminates. For upper bound solving, we need to discover a *precise ranking function* f .

$$f : \mathbb{Z}^n \implies \mathbb{Z}_+$$

$$\langle C(\bar{x}) \implies C(\bar{y}, \varepsilon) \rangle \text{ if } \varepsilon := f(\bar{x}) > f(\bar{y}).$$

The above is a formal definition for a ranking function. Of note is that ε captures information in which the value of variables change iteration to iteration, and ε captures information about guards of the loop such that it can detect when to terminate [4]. A ranking function can be said to be a function for a cost relation C if it can be used to rank all loops in $Loops(C)$. A cost relation ranking function is thus usable to find evaluation tree height because the ranking function decreases incrementally and is non-negative. The most common ranking functions are linear, which are the main types used within the PUBS solver [4]. Some loops may have further, tighter bounds, such as non-linear loops. If every difference in consecutive calls of a ranking function is guaranteed to be some constant $\sigma > 1$, then $\lceil \frac{f_c(\bar{x})}{\sigma} \rceil$. Other cases exist, such as when a loop ranking function decreases exponentially between steps, which is handled by a logarithmic ranking function. The logic behind such bound reductions is not the focus of this work, but worth mentioning as nonlinear loops occur within programs someone commonly. This shows that for an evaluation tree, its *height* h_+ , branching factor b can be observed, as well as using ranking functions to derive upper bounds for loops represented within the evaluation tree.

3.2.2.2 UPPER BOUNDING INDIVIDUAL NODE COST

Recall the overall method for an upper bound cost on an evaluation tree is to combine the cost per node with a bound on the number of nodes to reach an end bounded cost prediction for the execution of a given evaluation tree. The prior section discusses automatic inference upon the number of nodes of an evaluation tree, while this section discusses inference upon the cost per node. A tree can have internal nodes and leaves (external nodes), which must be handled differently. This section discusses, in brief, derivation of $costr_+(\bar{x})$, a function to calculate the cost of any internal node \bar{x} , as well as $costnr_+(\bar{x})$, which calculates the cost of an external node. The reason behind the separation between $costr_+(\cdot)$, $costnr_+(\cdot)$ is that external nodes have less reliance on unknown variables since they are composed of basic cost expressions [4]. Within each individual node, we can split the cost expressions into two solvable subproblems, *invariants* and *upper bounds of cost expressions*.

At the static analysis level, it may be impossible or unfeasible to track individual variable information. However, these values can be approximated from a superset of possible values using existing static analysis methods. To compute an *invariant* from a linear constrain perspective, we utilize $Loops(C)$. A linear constraint psi holds for the first call of a cost relation, $C(\bar{x}_0)$, and the arguments of any latter recursive call instance, $C(\bar{x})$ then there is a loop path that can be applied to reach one step further than the path needed to reach $C(\bar{x})$, call it $C(\bar{y})$. The math behind deriving loop invariants is beyond the scope of this work. The general approach is a proof through induction, to show that at any level of a possible loop, the next level can be reached with the same invariants. Loop invariants could be problematic since they are defined over all evaluation trees. The set of all evaluation trees for a cost relation system is very large or infinite, so this is incomputable in general. The approach used to compute invariants trades exactness for an answer, giving approximate

invariants through utilizing a convex-hull operation and a guarantees termination operation [4].

Finally, we need to determine the upper bounds on cost expressions themselves. Each node in an evaluation tree is composed of different instances of cost expressions. More generally, one can infer the upper bounds on the cost expressions in the initial cost relation system, and then apply those bounds to compute upper bounds for every internal and external node in the evaluation tree. This upper bounding technique is done using automatic linear constraint tools that follow a two step process of computing ϕ for a new variable y , and then syntactically searching in ϕ for an expression that can be rewritten to a reduced version of y . There are multiple further steps of optimization and reductions to tighten the upper bound as presented in [4], however the specifics are beyond the scope of this thesis. Combining prior explanations and after applying various optimizations, the cost of any initial call $C(\bar{v})$ of a cost equation is given by the following definition:

Definition 3.2.2. Let $S = S_1 \cup S_2$ be a cost relation where S_1 and S_2 are the sets of non-recursive and recursive equations for C . Let

- $\langle C(\bar{x}_0) \implies C(\bar{x}), \psi \rangle$ be a safe approximation of the loop invariant I_c .
- $E_i = \text{ubexp}(\text{exp}, \bar{x}_0, \varphi, \psi) \mid \langle C(\bar{x}) = \text{exp} + \sum_{j=1}^k C(\bar{y}_j, \varphi) \in S_i, 1 \leq i \leq 2$
- $\text{costnr}_+(\bar{x}_0) = \max(E_1)$ and $\text{costr}_+(\bar{x}_0) = \max(E_2)$

Then, for any call $C(\bar{v})$ and for all $T \in \text{Trees}(C(\bar{v}), S)$ it holds that

- $\forall \text{node}(_, r, _) \in \text{internal}(T). \text{costr}_+(\bar{v}) \geq r$
- $\forall \text{node}(_, r, _) \in \text{leaf}(T). \text{costnr}_+(\bar{v}) \geq r$

This allows for actual hard upper bounds to be substituted into non-recursive cost relations. Following this process allows for a static evaluation tree upper bound

approximation to be completed. Further improvements to this technique exist, however they will only be mentioned in brief here. The node-count upper bound approach discussed does not tightly bound *divide and conquer* programs. This is because the branching factor of divide and conquer programs is much larger than one, which is a partial presumption in the prior approach. To achieve a tighter bound, automatic upper bound solvers utilize a *level-count* upper bound approach. Additionally, the presented approach requires direct recursions, however it is entirely possible that a cost relation system contains indirect recursions (cycles involving more than one function). It is possible, however, to automatically transform a CRS into its *direct recursive* form by replacing all intermediate relations with final definitions through an unfolding process. Unfolding involves replacing a recursive call with its definition in a target relation. This process may be non-terminating, such as in the simple direct cyclical cost relation system presented prior. Cost Relation Systems can be further sorted by something known as their *Binding Time Classification* which encodes if their unfolding terminates. The high level process involves finding residual relations within a cost relation systems which must remain in the system, and working to eliminate the remaining unfoldable relations [4]. The algorithm behind this transformation involves an involved partial evaluation process that will not be summarized here.

3.2.2.3 AUTOMIC UPPER BOUND ISSUES

There are weaknesses to this automatic upper bounding for a cost relation system. Directly recursive cost relations do not have a cover point, meaning they are essentially infinite/non-terminating loops, are unable to have an upper cost bound derived. Finding ranking functions for loops within evaluation trees is difficult and uses a linear ranking approach. This inference method does not cover all possible

ranking functions, but is “good enough” for realistic use. Methods exist to infer more complex ranking functions, however they were not implemented in the tool this SEA utilizes. Finding invariants is difficult, and the linear constraint approach is fallible. For example

$$\langle C(n, m) = m, \{n = 0\} \rangle$$

$$\langle C(n, m) = C(n', m'), \{n' = n - 1, m' = 2 \times m, n > 0\} \rangle$$

which means the base case of $m = 2^n \times m_0$. Different methods exist for inferring more complex invariants, however they were also not implemented in the tool used within this static energy analyzer.

In this section, I summarize the main ideas behind cost relations and the process behind determining upper bounds for a system of cost relations. The CRS is broken into a (possibly infinite) set of evaluation trees, where we perform an upper bound approximation on an evaluation tree by combining the incremental recursive cost and the per-node cost. The incremental recursive cost is determined through the tree’s branching factor and height, bounded by a complete tree. Loops are expressed within the evaluation tree and performing some complex mathematics, allow for tighter bounding. The individual node cost is extracted through detecting invariants and upper bounding individual cost relations. Some cost relations may have complex recursion, which can sometimes be simplified using a partial evaluation unfolding algorithm [4].

3.3 PRACTICAL UPPER BOUND SOLVER (PUBS)

The approach of automatically solving for upper bounds is, as surveyed in the prior section, non-trivial. The research team that presented the above approach also released a free tool to practically solve cost relation systems. It is called the Practical Upper Bounds Solver (PUBS), which is primarily accessible through a web endpoint¹ PUBS is a Prolog based program rolled with the Parma Polyhedra Library for linear constraint handling. It reads a cost relation as input and outputs various upper bound, closed forms of the equations as output. Due to practical limitations, the only local version of PUBS accessible for this thesis is an x86 executable, which is not ideal as the programs of interest that are being analyzed are ARM assembly files. Since this is a static process, this is not an issue, as the ARM program can be given to PUBS running on a laptop of the appropriate architecture for the analysis, and then have the ARM still run on the original ARM system.

3.3.1 PROLOG OVERVIEW

Prolog is a *Logic* based programming language, a programming paradigm somewhat removed from more commonly used paradigms of *imperative*, *functional*, or *object-oriented*. Programming languages are categorized in many different ways. In Chapter 2, I discussed differences in semantics and individual languages' types systems. Programming languages can also be separated by paradigm, an abstract, core idea of any programming language. A language's paradigm will change how a programmer expresses the program they want to write. Imperative languages require the programmer to *instruct* the computer *how* to do what they want it

¹<http://costa.ls.fi.upm.es/pubs> [4]. PUBS was presented as a part of a research paper that is 13 years old as of 2024, which makes the web interface an undesirable endpoint for the SEA. Luckily, the research team, when contacted, shared a local executable version of PUBS for x86 based machines. This does limit the existing implementation of the SEA such that it analyzes an ARM assembly while on x86 architecture, however this is not problematic since the analysis is *static*.

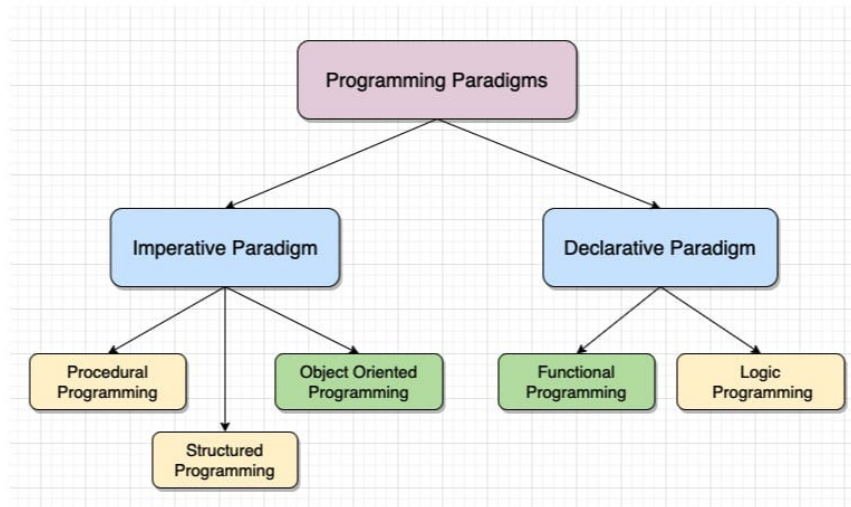


Figure 3.3: Common Programming Paradigms [31]

to do. Object oriented programming within the imperative category, but adds some additional ideas about containing data and actions to interact with said data together in objects. From the existence of objects comes the allowance for the tenets of OOP: encapsulation, abstraction, polymorphism, and inheritance. Even with all the additional abstractions objects allow within a system, one is still instructing the computer imperatively within OOP. An alternative umbrella style is declarative programming, which instead focuses on expressing a desired result and allowing an existing system to "figure it out". Within the declarative paradigm lies Prolog, a logic programming language.

When developing a Prolog system, the methods of "programming" are different from the typical imperative-esque approach. A Prolog developer must understand the formal relationships between objects within a problem, and which relationships should be considered "true" [14]. For a complicated mathematical system, such as the one specified in the prior section for automatic upper bound inference, this makes Prolog an ideal medium to express the constraints and successful relationships. If we constrain the program to declarations about the existing system, we allow for the computer to infer new facts from existing ones and perform various hidden optimizations [14].

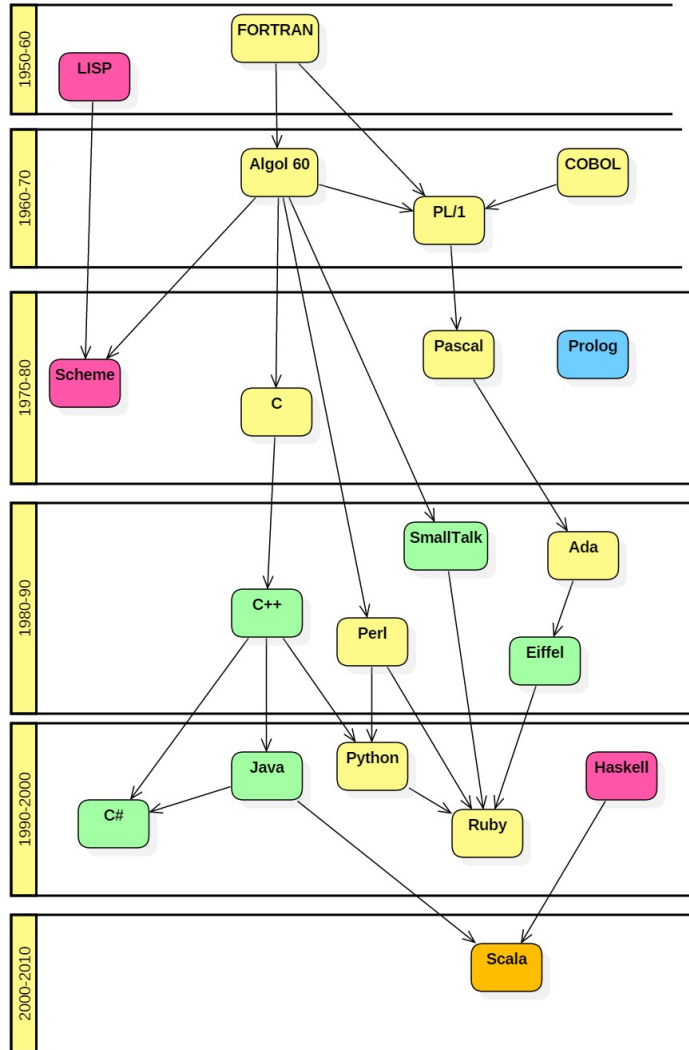


Figure 3.4: Programming Languages Colored By Paradigm over Time [39]

The basics of Prolog programming involves interacting with *facts*, *rules*, and *questions*. Within Prolog, everything is considered an “object”, but a disjoint idea from an object oriented notion of an object. This is a weakness of English as a communication medium, but not the point of this thesis. As seen in 3.4, Prolog emerged separately to object oriented languages, and as such the two ideas of objects are separate, just unfortunately sharing the same name. Objects interact through *relationships*, the other conceptual block with which Prolog logic systems are constructed. The actual modeling of a system in Prolog is done through specifying

facts and *rules* about objects and then asking *questions* about those objects and relationships. Prolog dynamically constructs containers for facts and rules about objects and their relationships as it interacts with the developer.

Facts are ways of defining one instance of a relationship between objects. Standard Prolog syntax is as follows:

```
relationship(object1, object2, ...).
```

where the relationship comes first, followed by all objects that the fact dictates are part of the relationship are surrounded by braces and in a comma separated list. Variable names, as `relationship`, `object1`, `object2` and so on are variables, must all begin with lowercase letters. To Prolog, the order in which the objects are listed has no inherent meaning, but it will attempt to follow the order provided, so it is important to be consistent. Questions asked to a Prolog system are done in a similar form of facts. Instead of defining objects, relationships, and a binding between them to be stored in a logical database, one instead *queries* the existing database to see if a statement is *true* or *false* [14]. It can become tedious asking true/false questions to a Prolog system, as every fact in question has to be fully written out. To account for this, one can use *variables* and allow for Prolog to solve for what a variable could possibly be. Capital letters are commonly used as ‘always variables’, meaning unknowns that the system will attempt to solve for. Beyond this, one can also define conjunctions between facts, such as *and*, *or*, etc. The final fundamental piece of Prolog is the ability to write *rules*, which are constructed in *if-then* like structure

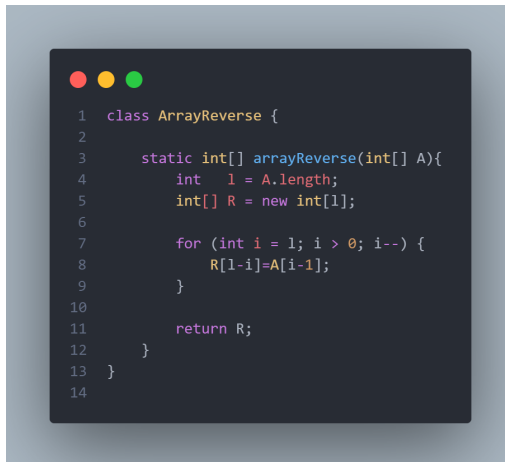
```
if_relation(a, b) :- result(a, b).
```

This is a very brief overview of Prolog and its declarative, logical paradigm [14]. The fundamentals of Prolog are very basic, which allows for unique logic systems

to be constructed on top of it. In our case, PUBS uses standard Prolog and a critical dependency of the Parma Polyhedra Library [10].

The Parma Polyhedra Library exposes numerical abstractions that are useful in many highly mathematical contexts, however it is aimed at abstracting various difficult steps within complex system analysis and verification. Static analysis for an automatic upper bound, the use of PUBS, falls within a subset of complex system verification. Of specific interest to the PUBS system is the ability to perform various steps required for automatic synthesis of linear ranking functions and parametric integer programming [10].

3.3.2 PUBS INTERFACE



```

1  class ArrayReverse {
2
3      static int[] arrayReverse(int[] A){
4          int l = A.length;
5          int[] R = new int[l];
6
7          for (int i = l; i > 0; i--) {
8              R[l-i]=A[i-1];
9          }
10
11         return R;
12     }
13 }
14

```

Figure 3.5: ArrayReverse() Java Method Code [4]



```

1  eq(arrayReverse(A),0,[m2(A)],[]).
2  eq(m0(A),2,[m1(A)],[]).
3  eq(m3,0,[],[]).
4  eq(m4(A),0,[],[]).
5  eq(m5(B),10,[m3,m0(A)],[B+ -A=0]).
6  eq(m2(A),0,[m5(A)],[]).
7  eq(m6(B),12,[m0(A)],[A>=0,B+ -A=1]).
8  eq(m1(A),0,[m6(A)],[]).
9  eq(m1(A),0,[m4(A)],[]).

```

Figure 3.6: Cost-Equation System in PUBS syntax of Figure 3.5

One example of a simple method to infer a static upper bound for can be seen in Figure 3.5, which is Java code for a method that simply reverses an array. For now, I hand wave the various transforms and assumptions used to transform the code to a cost-relations system, but one can see in Figure 3.6 a cost-relation system for the associated java ArrayReverse() code.

The various permissible syntax for the PUBS solver can be seen in Table 3.1.

PUBS Object	Definition
<i>Equation</i>	$:= \text{eq}(\text{Head}, \text{CostExpr}, \text{ListOfCalls}, \text{ListofSizeRel})$
<i>Head</i>	$:= \text{Name} \mid \text{Name}(\text{Arguments})$
<i>Arguments</i>	$:= \text{Variable} \mid \text{Variable}, \text{Arguments}$
<i>ListOfCalls</i>	$:= [] \mid [\text{Head} \mid \text{ListofCalls}]$
<i>ListOfSizeRel</i>	$:= [] \mid [\text{SizeRelation} \mid \text{ListOfSizeRelations}]$
<i>SizeRelation</i>	$:= \text{Variable} \text{ Oper } \text{LinearExpr}$
<i>LinearExpr</i>	$:= \text{RatNum} * \text{Variable} \mid \text{RatNum} * \text{Variable} + \text{LinearExpr}$
<i>Oper</i>	$:= >= \mid <= \mid =$
<i>CostExpr</i>	$:=$ As seen in section 3.2
<i>ListOfCostExpr</i>	$:= [] \mid [\text{CostExpr} \mid \text{ListofCostExpr}]$
<i>RatNum</i>	$:=$ Rational Number

Table 3.1: PUBS/Prolog Syntax

```

$> ./pubs_static {name of PUBS executable}
    -file <INFILE> {path to .ces file}
    -entry <ENTRYPOINT> {override default entrypoint}
    -show_asym <yes|no> {show asymptotic bounds}
    -output xml {output as XML instead of plaintext}
    -computebound <OPTIONS BELOW> {how UB is found}
        ubnormal {default node-count approach}
        ubnormal_withlevelcountenabled {level count}

```

Figure 3.7: PUBS Executable Usage

Running the PUBS solver on the `ArrayReverse()` cost-relation system produces a large amount of text output, which an abbreviated version of can be seen within Figure 3.8. The output from PUBS, even without additional logging and output

```

CRS '$pubs_aux_entry$(A) -- THE MAIN ENTRY
* Non Asymptotic Upper Bound: 12+14*nat(A)
* LOOPS '$pubs_aux_entry$(B) -> '$pubs_aux_entry$(C)
* Ranking function: N/A
* Invariants '$pubs_aux_entry$(A) -> '$pubs_aux_entry$(B)

  entry : []
  non-rec: [A=B]
  rec    : [0=1]
  inv    : [1*A+ -1*B=0]

CRS arrayReverse(A)
* Non Asymptotic Upper Bound: 12+14*nat(A)
* LOOPS arrayReverse(B) -> arrayReverse(C)
* Ranking function: N/A
* Invariants arrayReverse(A) -> arrayReverse(B)

  entry : []
  non-rec: [A=B]
  rec    : [0=1]
  inv    : [1*A+ -1*B=0]

```

Figure 3.8: PUBS ArrayReverse() Partial Output

flags enabled, is nontrivial. The goal of this example use of PUBS was to compute an upper bound for the array reverse function, which the solver found to be $12 + 14 * nat(A)$, where A is the size of the array passed into the method. If the set of cost-equations were instead built with the fundamental cost of assembly instructions, this PUBS result would give the desired output for an equation that would estimate the amount of energy required to run the code given an array of a specific size. Thus we see the basics of interacting with PUBS, the specificity in which the cost-equation systems must be writing and the basics of Prolog, the system of which PUBS was built within.

3.4 PARTIAL DYNAMIC COST ANALYSIS

The goal of automatic upper bound cost inference is to remove the self references within a static system and reach an end, non-recursive “cost” of a program. An alternative, older approach is that of allowing for *dynamic* execution profiling [53]. Following the above example of inferring the cost of an array reversal program presented in Figure 3.5, instead of translating the code into a system of cost relations, one must identify the fundamental blocks within assembly. These blocks are sequences of instructions that do not follow any conditional logic. In this instance, the assembly within the for loop is a basic block, the instructions required prior to the loop are a block, and the instructions after the loop are a block. Then, one runs the program in question, keeping track of how many times each block executes. With this additional information gleaned at runtime, the end cost of execution can be found through multiplying the total cost of each observed block by the number of times it executed.

This approach is simpler than the process used in static automatic upper bound cost inference, however it fundamentally changes the software observation process by requiring information that is only available at runtime. The use of this additional information does allow for significantly less complexity in the analysis process, but has a few limitations. Some programs are not able to be “run” trivially. In a high-performance computing setting, where one wants to estimate the cost of executing a calculation that may take hours, days, or weeks of compute time, running the program as a prerequisite to estimating the amount of energy it would consume while being run is nontrivial. More broadly, this approach still utilizes some static analysis techniques in identifying basic blocks of code execution, but then switches to dynamic analysis techniques in finding coefficients for an end solution. If one is already running the program as a step in getting an energy estimation, a fully dynamic analysis approach would be to use a run of the program

as the test to analyze. Existing profiling tools can do this, through different means of instrumentation or statistical methods [24]. These fully dynamic energy analysis methods are not the focus of this work.

CHAPTER 4

IMPLEMENTATION AND HARDWARE BENCHMARKING

In this chapter, I discuss the steps and associated software pieces implemented for this work. It also explains the reasoning behind using a Raspberry Bi 4 model B as the hardware testbench and the background of energy analysis in the context of modern day computing along with various metrics used in the field of energy aware development. Further, the testing configuration and methodology behind tests being run are explained. Through the hardware testing process, I demonstrate a range and average expected energy usage for many of the common instructions with the ARM instruction set architecture (ISA). These individual instruction energy values used as the smallest explicit cost blocks within the SEA, finally allowing for full static, partially automatic analysis of arbitrary assembly files.

4.1 STATIC ENERGY ANALYZER PROCESS

At its most simple, a SEA is a specialized compiler. Recall back to Chapter 2, where a compiler is presented as a “translation” tool. In this case, the goal of a static energy analyzer is to consume an assembly file and output an expected energy consumption of the program.

The first step in compilation is *tokenization*, where the input text is split into small chunks where each chunk represents a fundamental part of the programming

language in question. As initially detailed in Chapter 2, assembly has relatively simple grammar. Each line is a separate statement, with each line containing some selection of labels, mnemonics, operands, and comments. A parser was originally implemented in Zig, a new programming language targeted primarily for embedded systems. It supports significant interoperability with C [52], hence why it was originally chosen. Aspirations existed to utilize its interoperability with existing portions of the GNU tool chain, however the complexity of interfacing with an existing compiler quickly became out of scope, due primarily to the newness of Zig's documentation. The Zig approach had initial benefits, but the language's newness and lack of official 1.0 release showed itself in its lack of robust and accessible graphing library. Thus, an assembly to control flow graph visualization tool was restarted in Python, with said language choice bringing tools such as matplotlib, networkx, and first class string support to the table. This is an intermediate step chosen to allow for better visualization of an underlying ARM assembly, targeted towards allowing manual creation of a program's cost relation. Existing GNU utilities can output a program's control flow graph in a specific format, which is an ideal starting point instead of duplicating work through re-parsing and reconstructing a CFG from generated assembly. Unfortunately, these utilities output control flow graphs of programs in higher language forms than assembly. Invoking `-fdump-tree-cfg` on a gcc instance on a `.c` program functions as expected, but a similar invocation with an intermediate assembly program `.s` fails.

The SEA that was implemented for this project is semi-automatic. Therefore, wrapping the PUBS solver in a more accessible command line script was unnecessary. This project uses PUBS as the static upper bound cost solver for a program.

The other portion of data required for a full static energy analyzer is hardware data to convert the hypothetical static model into a final energy consumption prediction. Further detail on how individual assembly instructions were benchmarked can

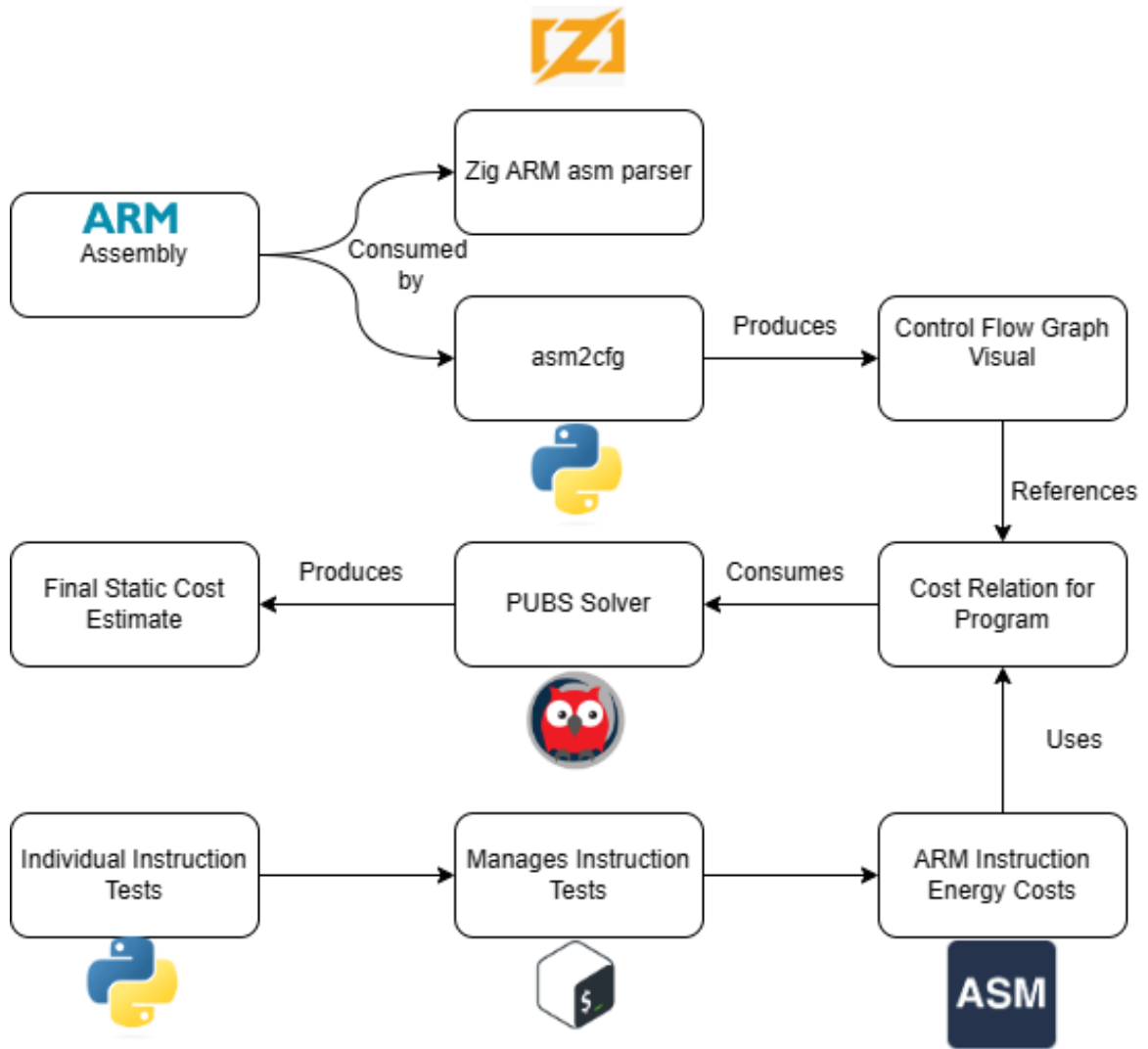


Figure 4.1: Semi-Automated Static Energy Analyzer Process

be found in Chapter 4. It involved manually writing a small amount of assembly and using a python script to “unroll” that into an assembly file of many individual repeated instructions. The testing of assembly was aided by the use of a bash script that managed compilation and processor speed throttling. A visual representation of the main steps of the static energy analyzer and the software languages/tools utilized can be seen in Figure 4.1.

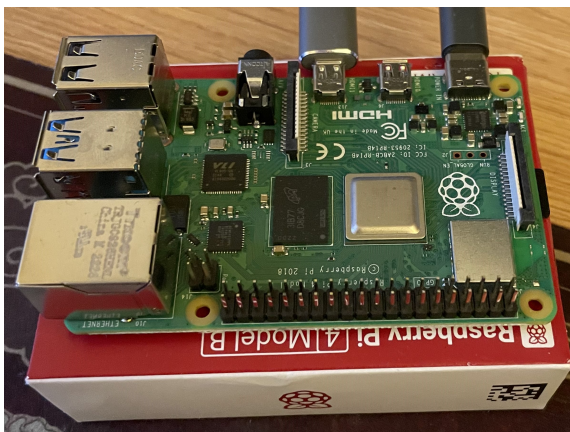


Figure 4.2: Raspberry Pi Testbench



Figure 4.3: Energy Reading Multimeter

4.2 HARDWARE CHOICE

The Raspberry Pi 4 model B (further referred to as just the “Pi”) is the hardware testbench of choice for this project. Raspberry Pis are easily accessible, fully-functional computers and microcontrollers sold by the Raspberry Pi Foundation. They all are powered by an ARM architecture. The Pi was the testbench of choice due to its ARM architecture and Raspberry Pis having a wealth of tooling and documentation. The specific Pi used has a quad-core Cortex-A72 64 bit processor with a clock speed of 1.5 Gigahertz. It has 4 GB of RAM memory, as well as support for wifi, bluetooth, and more. The Pi has two micro HDMI portas as video output [47]. Its operating system, a Raspberry Pi foundation distribution of Linux, runs on it off of an SD card. It does not have active cooling, as passive air cooling is enough to keep its operating temperatures under control. The processor, an ARM Cortex-A72 has L1 and L2 cache subsystems as a buffer before regular random access memory [6]. The L1 cache is split into an instruction cache and a data cache.

The Pi is powered by a USB-C cable and the energy reading device is a multimeter that sits in between the USB-C power cable and the Pi testbench. A picture of the tool can be seen in Figure 4.3. The multimeter displays both voltage (Volts) and

current (Amps). To get *power* consumption, one must multiply the two based on the equation [55]:

$$P = I \times V$$

The multimeter has additional functionality, such as displaying changes in amperage or voltage over time. For this project, the readings are manually checked during execution of tests. Synchronizing a logging multimeter with the testbench without introducing communication overhead fell outside the scope of this project, but is an area of improvement for future experimentation. Alternative energy tracking approaches involve utilizing advanced multimeters or using existing sensors on the computer's motherboard itself. Interacting with onboard power consumption registers is challenging and may not result in significant accuracy improvements, as the interaction to read the energy register also consumes energy. Additionally, the specific Pi model being used as a testbench is not supported by more robust energy tracking tools [47] that its Intel x86 and AMD counterparts do have, such as *perf*, *rapl*, *Intel PowerLog*, etc.

4.3 ENERGY TESTING BACKGROUND

The energy consumption of processors is an interesting topic for a multitude of reasons. One estimate projects that servers are approximately 1.5% of all global energy consumption [17]. Beyond the energy required to run a server farm, an even larger proportion of energy and other resources, such as water or other liquids [33] are consumed in cooling server farms [22]. One analysis finds that a processor contributes up to 30% of a computer's overall energy consumption [38]. Hence, understanding more about the inner workings of a processor allows for these costs to be reduced. Refer back to Chapter 2 for a "software developer" perspective on

computer processors, where possible complications with modeling a processor are discussed. This section will serve as an electrical engineering perspective of how processors work. This bottom-up approach is applicable to hardware testing as individual instruction profiling is closer to an electronics problem than a software engineering task.

Processors can be thought of as extremely complex circuits composed of transistors. For this project, consider a transistor to simply be the basic building block of a processor that has two states, ON or OFF [57]. The state of a transistor allows or prevents current from passing through it. Logic gates are constructed through stringing transistors together in various ways to achieve various outputs based on energy “input” to the gate. Whenever a transistor changes state, it must discharge or recharge an amount of electricity known as its *inherent capacitance* (C). A large complex circuit of transistors has two ways in which energy is consumed. Static energy is energy that is always being consumed, regardless of activity within the circuit. The amount of static energy consumed depends on supply voltage and other characteristics of the circuit such as materials and gate length [57]. For this work, static energy consumption is presumed to generally be a characteristic of the circuit in question that will not be affected by external changes such as power supply, temperature, etc. Dynamic energy is the other type of power consumption of a circuit that is a summation of the energy used in changing transistor state. A circuit could theoretically stay in one state, meaning all energy consumption would be static. For modern computers to perform computation, the state of the circuit is changing often. The amount of energy used dynamically by each transistor is dependent on various transistor properties such as inherent capacitance, but also the number of state changes desired and the voltage of the system [57].

One defines energy usage E of a transistor with voltage V at frequency f with

inherent capacitance C and activity a as the following equation [57]:

$$E = \frac{1}{2} \times C \times V^2 \times f \times a$$

which demonstrates two important relationships. Dynamic energy has a linear relationship to inherent capacitance (C), frequency (f), and activity (a), and a quadratic relationship to voltage (V). This equation can be used to model more than an individual transistor if activity and inherent capacitance for a larger system are known. Unfortunately, hardware manufacturers rarely release information on the inherent capacitance of an entire system because as its computation is quite involved and it is arguably proprietary information. Additionally, knowing the activity a for an entire system is incredibly complicated, as it requires a simulation of the entire computation to count every state change that would occur. The relationships are important, but the equation itself is not easily usable when testing a processor.

Many processors have metrics that allow for comparison on various axes. For example, many processors have performance metrics that attempt to quantify total computing power such as clock speed, GigaFLOPS, etc. Processors are incredibly complex systems that make various tradeoffs (for example, a slower clock speed processor with superior architecture may be faster than a higher clock speed outdated circuit). Instead of processor metrics, a common way of comparing processors is to testing the practically on various benchmarks and compare the results. Energy analysis metrics exist as well, such as Thermal Design Power (TDP). TDP is a metric of a processor provided by the manufacturer to explain its cooling needs by showing the maximum power usage (and thus, heat production) [57]. TDP is a metric useful for knowing an upper limit, but does not provide further insight into inferring the amount of dynamic energy consumed by a processor. A useful approach for better inference of energy consumption and efficiency are correlation metrics. By taking

readings of a processor and correlating that with its current activity, one obtains a better overall model of a processors energy usage. One common energy correlation metric is Energy Delay Product (EDP) and other weightings of variable influence such as ED^2P [57]. Energy delay product is a value calculated based on the amount of energy consumed and the time taken to finish computation. For this project, a correlation approach is used to correlate running a specific instruction with an energy consumption reading.

4.4 ARM TESTING METHODOLOGY

This section details the methods used and some complications that arise when attempting to extract assembly instruction level energy values for the Pi. ARM v8 is a reduced instruction set (RISC) architecture, with many individual mnemonics for “instructions” however these instructions can generally be separated into various groups of very similar instructions. The approach used in [20] for profiling the low-level virtual machine intermediate representation (LLVM IR) instruction set architecture involved splitting the instructions into 4 groups, memory M , program flow B , division D , all other instructions G . The ARM v8 instruction set architecture can similarly be split into group of instructions. Following the approach presented in [57], these instruction groups are branching/control flow, integer/logic, floating point, register movement, compare/test, and all other instructions and mnemonics. These groupings allow for testing a reduced amount of the total instruction set space without sacrificing much accuracy. The general approach to correlating an energy usage value to an individual instruction is to repeat that instruction many times while tracking power consumption of the computer during the benchmark process execution. Then power usage per individual instruction is found by dividing the amount of power used by the number of times the instruction in question

```

1      .arch armv8-a      ; various pseudo-ops
2      .text
3      .align 8
4      .global main
5      .type main, %function
6  main:          ; function entrypoint
7  .LOOP_START:
8      .cfi_startproc
9      <TESTOP>      <TESTOPERANDS> ; repeated ~2000
10     b.al      .LOOP_START ; branch always to loop start
11     .cfi_endproc
12  .LOOP_END:
13     ; various end-of-file pseudoops

```

Listing 4.1: Benchmark Instruction Test File Structure

was run. Some considerations are given to background energy consumption not attributable to the program being executed. To make the Pi repeat one ARM instruction repeatedly, a large loop of repeated assembly instructions are used. Each individual instruction test is generated with the aid of a python metaprogramming script. The outputted assembly is about lines of repeated assembly instructions that infinitely loops back to the beginning of the unrolled loop block. An rough outline of the script can be seen in Listing 4.1. After the test assembly is written and verified by hand, it then needs to be assembled with the aid of GNU's `as` tool [43] and linked with `ld` [44], which can alternatively can both be done simply by invoking `gcc` on the assembly `.s` file. After generating the test, assembling, and linking, it is then run for a significant enough time to ignore initialization, background processes, operating system overhead, etc. While running the benchmark, voltage and current are tracked. After stopping the benchmark, energy per instruction (EPI) can be computed statistically. While many assembly instructions are very similar, additional complication is introduced by the fact that one instruction may

```

1 add    x1, x2, x3
2 add    x4, x5, x6
3 add    x7, x8, x9

```

Listing 4.2: add with out read-after-write

consume a different amount of energy based on its operands. For example, consider a possible implementation of an add mnemonic test which does not have read-after-write hazards [57]. As discussed in Chapter 2, processor pipelines provide speedups through parallelizing as much work as possible, but hazards appear if one instruction relies directly upon the result of the prior. Thus, an alternative test approach is to purposefully write the repeated operation such that it encounters as many hazards as possible:

```

1 add    x1, x2, x3
2 add    x4, x1, x6
3 add    x5, x4, x7

```

Listing 4.3: add with read-after-write hazards

Due to the limited resources available for this project, testing every assembly instruction for both optimal no-hazards and worst-case all-hazards is unfeasible. Additionally, since the goal of this hardware testing is to determine an EPI cost for every assembly instruction, having multiple costs for the same instruction complicates things. When attempting static upper bound analysis, determining if and when an instruction creates a data hazard requires practically entire program simulation. An in-depth simulation of an entire program as a “static” method is questionable, at best. Static analysis tools generally aim to be helpful at the compile step, where the computation time required for a whole program simulation is not usually accessible. Exploring differences in EPI based on data hazards in the handwritten assembly is one possible extension for this work.

4.5 EXPERIMENTAL ENERGY PER INSTRUCTION RESULTS

Clock speed is important in determining individual energy per instruction values. The Pi comes with a 600 Mhz minimum clock speed and a 1.8 GHz (1,800 MHz) maximum clock speed. The governance of when processor frequency changes is one variable that should be controlled, as different frequencies may have different static and dynamic energy consumption rates [57]. Forcing clock speed limits and minimums on the Pi was accomplished through the aid of `cpupower` [7]. Highly unrolled loops of test instructions were observed at 600, 1200, and 1800 MHz clock speeds. For all testing, the multimeter kept a voltage reading in the 4.8 - 4.9 range.

EPI values were computed with the formula:

$$EPI = V \times (A_{ins} - A_{static}) / (C * \sigma)$$

where A_{ins} is the amperage of the Pi during a given instruction's benchmark test, A_{static} is a baseline amperage of the Pi at the given operating frequency, V is voltage presumed to be ~ 4.85 , C is the number of cycles per second, and σ is a tolerance to account for execution of instructions other than *ins*. Results for dynamic energy consumption for instructions at various frequencies were found to be in the same magnitude as other works [20, 57], however the imprecision of the manual logging system likely introduced significant human error into the mix. Static energy consumption was checked by running minimal background processes on the testbench, with the exception of powering a monitor, the terminal session, and the software test harness to run the individual instruction test. Results can be seen in Table 4.1.

Following the approaches presented in [57, 20], a smaller subset of assembly instructions was tested since many instructions perform very similar operations

Clock Speed (MHz)	600	800	1000	1200	1400	1600	1800
Idle Current (A)	0.27	0.3	0.32	0.32	0.34	0.32	0.33

Table 4.1: Static Current Consumption at Different Clock Speeds of Pi

with slight differences in areas such as the operands, minor logic swaps, etc. The main instructions tested fell into categories of integer arithmetic, boolean logic, floating point arithmetic, data movement, and data load/storage. All energy per instruction (EPI) values in the following results tables are in pico-joules (pJ).

		Proc. Freq. (MHz)		
		600	1,200	1,800
Instruction				
	add	596	234	340
	sub	383	243	374
	mul	170	72	142
	div	391	140	116

Table 4.2: Estimated Energy Per Instruction (pJ), integer operations

The integer operations show some variance, surprisingly the division and multiplication operations having lower EPIs at higher frequencies that addition and subtraction.

		Proc. Freq. (MHz)		
		600	1,200	1,800
Instruction				
	nop	162	255	156
	and	230	102	204
	orr	230	106	199
	eor	196	238	247
	bic	255	149	213

Table 4.3: Estimated Energy Per Instruction (pJ), bitwise logic operations

Of note in Table 4.3 is the instruction nop, with is “no operation”. The common logical operations of and, exclusive or, inclusive or all have similar EPIs across clock speeds. Bitwise clear bic also demonstrated similar EPI.

A similar trend is seen in Table 4.5, where it appears that the EPI is reduced at a “median” clock speed compared to the slowest limit. One thing worth noting is

Instruction	Proc. Freq. (MHz)		
	600	1,200	1,800
fadd	221	128	213
fadd (db)	238	157	216
fsub	247	115	193
fsub (db)	255	170	204
fmul	204	115	201
fmul (db)	289	132	201
fdiv	196	119	162
fdiv (db)	213	119	196

Table 4.4: Estimated Energy Per Instruction (pJ), floating point operation

that the floating point arithmetic operations have different precision for the same mnemonic based on their operands. Entries marked as (db) in Table 4.5 were tested on double precision operands, while those without invoked the single precision version of the operation.

Instruction	Proc. Freq. (MHz)		
	600	1,200	1,800
mvn	213	115	173
mov	196	89	170
mov (db)	255	111	182
fmov	255	115	213
cmn	247	106	176
cmp	264	128	216
tst	247	234	235
b	230	85	150
ldr	511	170	199
str	230	204	233

Table 4.5: Estimated Energy Per Instruction (pJ), move, compare, load, store operations

Other instructions of interest can be Table 4.5. `mov` and `mvn` copy an argument (register or immediate) into another register either directly or negated, respectively. `(db)`, like with floating point arithmetic operations, signifies moving of a double precision float, while `fmov` is an instruction to move a single precision floating point. The other instructions of specific interest are those of `ldr` and `str`, for loading and storing of register values, respectively. For operations interacting with

memory, primarily loads and stores, the unrolled loop section was modified such that the number of instructions per iteration was a power of 2. This consideration is modeled after the reasoning of better memory utilization as presented in [57]. The mnemonic `ldr` had a higher EPI than its peers at lower frequencies. Testing branching mnemonics is questionable as branches themselves are rarely a large proportion of a compiled program [57].

Testing memory loads and stores is additionally complicated by the fact that time to execute one instruction may be stalled by waiting on responses from various levels of cache or other memory. Binary logic did not show many differences between instructions. `nop` has the lowest EPI at a lower frequency, but a surprising high EPI at the middle frequency of 1.2 GHz. This is curious, as `nop` should theoretically have extremely low effect on the amount of energy consumed as it is *not doing* any new computation. Generally, these initial experimental findings do not show significant differences between instructions and dynamic energy consumption, instead clock speed appears more correlated to EPI.

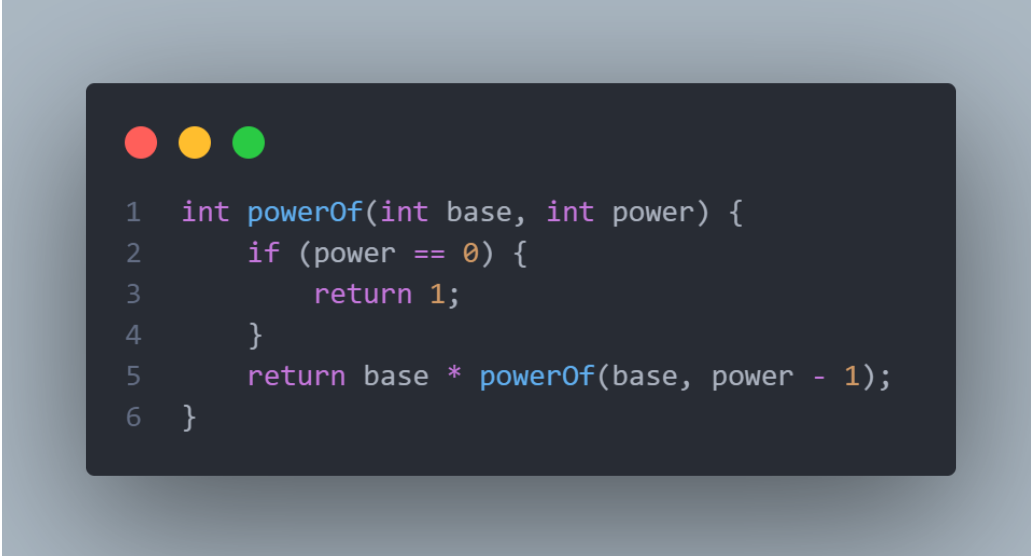
CHAPTER 5

DISCUSSION

This chapter brings together the hardware testing results and SEA methods and demonstrates how one would perform static energy analysis on ARM programs. Additionally, I discuss the various challenges faced in this work for both the SEA implementation and hardware testbench profiling.

5.1 EXAMPLE SEA WORKFLOW

A prerequisite to analysis is having something to analyze. This section will follow the simple example of wanting to estimate the static upper bound cost of a recursive computation of a number to a power. A simple implementation of this, in C code, is provided in [Figure 5.1](#).

A terminal window with a dark background and three colored window control buttons (red, yellow, green) at the top left. The code is displayed in a light blue font with line numbers 1 through 6 on the left side. The code defines a recursive function named powerOf that takes two integers, base and power, and returns the base raised to the power. The function has a base case where power is 0, returning 1, and a recursive case where it returns base multiplied by powerOf(base, power - 1).

```
1 int powerOf(int base, int power) {  
2     if (power == 0) {  
3         return 1;  
4     }  
5     return base * powerOf(base, power - 1);  
6 }
```

Figure 5.1: C Code for PowerOf function

C code is not the desired input format for the static energy analysis process presented in this work, so first it must be compiled into ARM assembly. Depending on platform, a cross compilation tool may be necessary. For example, `aarch64-linux-gnu-gcc` is an arm compiler on the x86 platform, instead of just `gcc` when a computer that is already ARM using architecture. PowerOf is then compiled on the command line, as seen in Figure 5.2.

A terminal window with a dark background showing a single command in light blue font: `$>gcc powerOf.c -S -fverbose-asm`.

```
$>gcc powerOf.c -S -fverbose-asm
```

Figure 5.2: PowerOf Compilation and Flags

The `-S` compilation flag tells `gcc` to emit assembly, as opposed to a complete executable file. The `-fverbose-asm` flag tells the compiler to attempt to automatically comment the produced assembly, which helps with readability [19]. The verbose flag is not necessary, but does help with manually constructing a control flow graph of the assembly code and translation into a recursive representation. The main section of the produced assembly for `powerOf` can be seen in Figure 5.3.

```

1  powerOf:
2  .LFB0:
3      .cfi_startproc
4      stp x29, x30, [sp, -32]!    //,,,
5      .cfi_def_cfa_offset 32
6      .cfi_offset 29, -32
7      .cfi_offset 30, -24
8      mov x29, sp //,
9      str w0, [sp, 28]    // base, base
10     str w1, [sp, 24]    // power, power
11     // power.c:3:    if (power == 0) {
12     ldr w0, [sp, 24]    // tmp94, power
13     cmp w0, 0 // tmp94,
14     bne .L2 //,
15     // power.c:4:    return 1;
16     mov w0, 1 // _3,
17     b .L3 //
18     .L2:
19     // power.c:6:    return base * powerOf(base, power - 1);
20     ldr w0, [sp, 24]    // tmp95, power
21     sub w0, w0, #1 // _1, tmp95,
22     mov w1, w0 //, _1
23     ldr w0, [sp, 28]    //, base
24     bl powerOf //
25     mov w1, w0 // _2,
26     // power.c:6:    return base * powerOf(base, power - 1);
27     ldr w0, [sp, 28]    // tmp96, base
28     mul w0, w1, w0 // _3, _2, tmp96
29     .L3:
30     // power.c:7: }
31     ldp x29, x30, [sp], 32 //,,,
32     .cfi_restore 30
33     .cfi_restore 29
34     .cfi_def_cfa_offset 0
35     ret
36     .cfi_endproc

```

Figure 5.3: ARM Assembly of PowerOf function

The C programming language and the GCC compiler have a long history and robust tooling at this point, however if one desired to analyze a program from a different source language, the compilation flags would look different. The next step in the static analysis process of an imperative language is to construct a control flow graph of the program. Some tools exist that can do automatic control flow graph construction of assembly, however it is much more common for higher level languages to support easy control flow graph visualizations. In the compilation process, the produced assembly code typically exists after the control flow graph has been created and has optimizations applied to it, thus there is little reason to reconstruct a CFG for the final steps of assembling and linking. For more information on the compilation process of a program, refer back to Chapter 2.

A manually constructed CFG for the assembly for `powerOf` can be seen in Figure 5.4. The various pseudo-operations were stripped for this control flow representation. The CFG starts at the top block, where the stack pointer is moved and the arguments base, `power` are stored in certain registers. Then, the next three instructions `ldr`, `cmp`, and `bne` constitute the original if statement. The value `power` is loaded, compared to 0 (as if the power is 0, any number raised to 0 is 1), and if the prior comparison is not equal, the program branches to the recursive step. If `power` is 0, then the processor moves 1 to a register and always branches to a cleanup block where the function's value is returned. In the recursive portion, values are manipulated until the recursive call is executed through the `b1` (branch and link) instruction, which links to the initial method label. After linking, the result from the recursive call is loaded, the multiplication is performed, and then the result is returned.

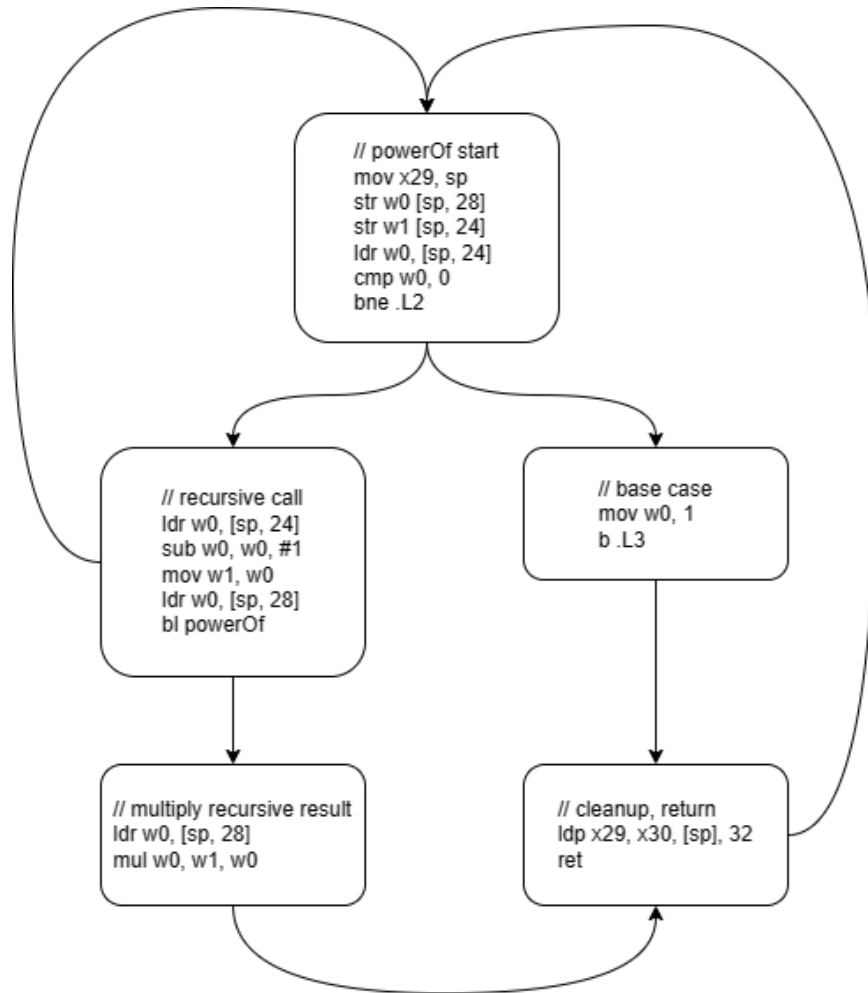


Figure 5.4: Control Flow Graph for `powerOf` Assembly Code

The next step is to flatten the control flow graph into a recursive representation, as well as construct guards to each of the blocks in the control flow graph. A more detailed explanation of this process can be found in Chapter 3. The step of detecting appropriate guard clauses for assembly is non-trivial because of how little additional information is present in assembly at compile time. Since this step is manual, multiple valid representations likely exist. If done automatically, an additional intermediate step is to translate the recursive representation of a program into a representative cost relation. The recursive representation and translation into cost expressions can be done at the same time manually for simple programs. Many optimizations exist upon how representations and costs are found, however

manually applying all of them is well beyond the scope of this work. One example of a possible cost relation representation for the `powerOf` function is as follows:

$$\begin{aligned}
 (a) \quad C_{powerOf}(A, B) &= k_1 + C_{powerOf}(A, B) && \{B \geq 0\} \\
 (b) \quad C_{powerOf}(A, B) &= k_1 + C_{base}(A, B) && \{B \geq 0\} \\
 (c) \quad C_{recur}(A, B) &= k_2 + C_{powerOf}(A, B - 1) + k_3 + C_{cleanup}(A, B) && \{B \geq 1\} \\
 (d) \quad C_{cleanup}(A, B) &= k_4 && \{B \geq 0\} \\
 (e) \quad C_{base}(A, B) &= k_5 + C_{cleanup}(A, B) && \{B = 0\}
 \end{aligned}$$

$$k_1 := \langle MOV \rangle + 2 \times \langle STR \rangle + \langle LDR \rangle + \langle CMP \rangle + \langle BNE \rangle$$

$$k_2 := \langle LDR \rangle + \langle SUB \rangle + \langle MOV \rangle + \langle LDR \rangle$$

$$k_3 := \langle LDR \rangle + \langle MUL \rangle$$

$$k_4 := \langle LDP \rangle + \langle RET \rangle$$

$$k_5 := \langle MOV \rangle + \langle B \rangle$$

Each of the lettered equations represents a single cost equation. The entire system is the overall cost relation for the original `powerOf` function shown in Figure 5.1. A, B are variables to represent the parameters `base` and `power` respectively. Each of the k_i variables represent basic cost expressions that are a summation of the sequentially executed assembly in that block. What they expand to, in terms of individual instructions, are shown as well. These individual instruction cost values will vary depending on the chosen method of determining energy per instruction. In this work, instructions were broadly categorized and then a sample from each category was tested. Alternatively, one could have a comprehensive energy-per instruction list that would allow for simple 1-to-1 instruction to energy value conversion. As discussed later in the challenges section, assembly has a non-trivial amount of


```

1 eq(power(A),0,[m0(A)],[]).
2 eq(m1,0,[],[]).
3 eq(m0(A),0,[m2(A)],[]).
4 eq(m3(A),2,[m1],[A=0]).
5 eq(m4(B),8,[m1,power(A)],[A>=0,B+ -A=1]).
6 eq(m2(A),2,[m5(A)],[]).
7 eq(m5(A),0,[m4(A)],[]).
8 eq(m5(A),0,[m3(A)],[]).

```

Figure 5.5: Example of Possible PowerOf PUBS Input [4, 3]

complexity in how the same instruction may operate differently based on various factors, making this comprehensive approach not feasible.

Finally, the last step is to translate the cost relation into the required Prolog syntax for the PUBS solver. In this step, one replaces all the instruction cost placeholders with actual EPI values. An example cost equation system in proper PUBS syntax can be seen in Figure 5.5.

It is important to note that the powerOf PUBS input in Figure 5.5 was provided by the original authors of PUBS, thus it is not the product of the manually generated cost relation system constructed above. After creating a valid PUBS cost equation system, the logic program is then run to automatically infer a static upper bound of the program, if possible. The PUBS result for Figure 5.5 is:

$$4 + 10 \times \text{nat}(A)$$

The constants 4 and 10 will change depending on how basic costs are chosen.

If one would like to compare static results to dynamic values, one must simply dynamically test the program in question with various inputs and compare the

amount of energy consumed compared to the consumption predicted by the static model.

5.2 A NON-SEA

Creation of a static energy analyzer is extremely difficult. The approach presented in [3] for translating imperative programs into cost relations, followed by automatic upper bound cost inference in [4] is incredibly difficult. Working with assembly provides a common ground for all instructions, but it is not a target of general tooling. Manual writing of assembly is slowly becoming more niche as compilers have become so optimized that writing in a higher level language is not a downside for performance. In terms of creating a SEA, this means assembly is not a good choice as analysis medium, in terms of making use of existing frameworks and tools. The creation of robust assembly analysis tools tailored for static analysis lies much more in the realm of a potential Ph.D. thesis or enterprise funded research. For example, a nontrivial amount of static analysis has been used in military applications, such as static analysis of linear programs [16]. This thesis presents the main pieces necessary in a static energy analyzer, without the some of the connective tissue required for a fully automatic static energy analysis tool. Immediate further work would involve developing an automatic assembly to cost-relation system framework, which is the largest non-automatic portion of the static analysis process presented in this thesis. The general process of static cost analysis of a program is presented at length in Chapter 3. The final step is to replace the basic cost instructions with values of individual assembly instructions, where are tested for using the approach presented in Chapter 4. Alternatively, some tools for more complicated processors allow entire program retracing, in which case one could run a program and find all of the individual instructions executed with a retracing tool [32]. This approach would

be similar to the partial dynamic method presented at the end of Chapter 3, where it uses some dynamic analysis techniques to remove the need for complex upper bound analysis.

5.3 CHALLENGES

5.3.1 STATIC ANALYSIS

The overarching challenge of static analysis is in its immense complexity. The methods presented in Chapter 3 primarily follow the work of a specialized research group in Madrid. When performing dynamic program analysis, it is easier to accomplish an immediate result, through simply running the program in question. Many further extensions to dynamic analysis exist, typically through various tooling that observes various different parts of interest on the computer. Regardless, the time and effort between initial analysis and a result is less than that of static analysis. If one lets dynamic analysis be oversimplified to “running” the program with some extra bits and pieces, static analysis does not have an easy comparison. Static program analysis must interact with a static intermediate representation of a program, then perform significant mathematical operations upon it to achieve the desired conversions. The math required is incredibly complex, with many pitfalls that can easily lead to nonterminal programs unable to be analyzed statically. This type of mathematical representation of a program and interacting with it is a foreign concept to many developers, the researcher included. Thus the type of programming and logic required to automatically convert a control flow graph to a reduced recursive representation composed of basic cost expressions fell beyond the scope of this work. For simple programs, manual construction of a cost relation is more manageable than developing an automated conversion tool.

Control flow graphs are an intuitive representation of how a program moves

throughout its runtime. In the case of ARM assembly, its rigid structure may lead one to believe CFGs of assembly are easy to create. This unfortunately turned out to not be the case, as assembly has a multitude of hidden pitfalls one could encounter in trying to generate an accurate CFG. Jumps form edges between assembly blocks, but jumps do not always go precisely to labeled sections of code. For example, partial execution and varying register sizes (relevant assembly optimizations) means the same branch instructions may function differently. Unconditional, arbitrary jumps are incredibly powerful because they allow assembly to express nearly any possible computation, but following these jumps statically is nontrivial.

5.3.2 HARDWARE TESTING

The largest challenge in working with the Pi testbench arose from the hidden complexity of testing ARM assembly. Some works similar to this thesis reduce the necessary tested instruction set through grouping instructions into more broad groups than even this work does [20]. While ARMv8 is RISC, this means more so about the complexity of its individual instructions, not the complexity of the instruction set itself. Assembly, regardless of actual specification (MIPS, ARM, WASM, etc.) is hard. The amount of subtle variance between mnemonics is large. This presents a challenge of determining when to group an instruction with an existing benchmark and when to create a new benchmark. Manually writing an unrolled loop to repeat an instruction infinitely while also not allowing for processor optimization is non-trivial. The differing length of processing pipelines for different types of instructions makes forcing or avoiding data hazards prohibitively complex. Additionally, the combinations of different registers and addressing modes for the same instructions makes determining the thoroughness to which the ISA has been tested confusing.

Modern computers are incredibly fast, such that determining EPI relies on

converting a larger measurement down to an individual instruction level. This conversion relies on a few assumptions, for example, the accuracy of a precise clock speed not varying throughout the time sample. Furthermore, the tolerance σ may be a different value or be a different type of effect. For example, perhaps the minimal background processes only use a constant amount of instructions regardless of processor frequency. Background polling through `top` showed all benchmark tests consuming 100% of the CPU, with all other processes consuming less than a fraction of a percent. It is possible the assumptions used in conversions for this thesis are incorrect. Additionally even on a RISC architecture, some instructions may take more than one cycle to complete [57]. Instructions taking more than a cycle to execute would reduce the total number of instructions executed in the sample, thus increasing the EPI.

All of these hardware challenges are easy to discuss, but difficult to track. The Pi testbench is relatively limited in its access to common dynamic analysis tools due to its architecture and intended purpose. Unlike some more specially designed systems with extra registers that show important metrics such as power consumption, the Pi has only a few registers that are useful. Primarily, the Pi does have a way of checking its current processor frequency, but lacks support for more power focused utilities such as `ptop`, `perf`, or other trackers for processor stalls.

The limitations and challenges of this work are many, as at nearly every step of implementation, there are still a few layers of abstraction to follow until getting an appropriately “accurate” result. With all these limitations known, there is a real tradeoff between complexity of the tool versus its accuracy. To properly model a full ISA test and account for variance, one would need to simulate the entire computation to track the additional constraints and effects happening at the gate level. If less rigor is permitted, testing difficulty decreases. This work attempted as

much granularity as possible with the limited resources available in the instruction set testing modeling and static energy analysis tool.

CHAPTER 6

CONCLUSION

A wrap up chapter, one step further removed from the results and challenges presented in Chapter 5. In this chapter, possible extensions and future work are discussed as well as a more broad conclusion to this independent study in general.

6.1 FUTURE WORK

This work serves as a starting point for static energy analysis. It presents significant background material on compilers, computer architecture, and program analysis. Additionally, it summarizes the concept of cost relations and the process of inferring static cost upper bounds, as presented in [3, 4, 58]. This theory is applied through the presentation of how to perform static cost analysis of the chosen source language, ARM assembly. The process presented here is semi-automated, with a fully automatic energy analysis tool being left as a future work. The PUBS solver is incredibly helpful for static cost analysis provided a cost relation in proper formatting. Finding an optimal cost relation that represents an arbitrary program is nontrivial. Detecting cost relations for higher level languages (regardless of paradigm) is easier than that of low level languages, however this ease is cancelled by the additional difficulty of determining what a “basic” instruction is, and what

an appropriate associated cost would be. For programs written in an imperative paradigm, the process is to reconstruct a control flow graph, remove unnecessary information, reduce to a recursive representation, then union with calls-to-size relations to finally produce an set of recursive cost equations representative of the imperative program [3]. This work accomplished presenting a CFG of ARM assembly, but faltered at the automatic recursive representation conversion step. Thus, an obvious extension is to iterate further to allow for arbitrary automatic cost relation construction of assembly files.

Another axis of exploration is that of choosing an alternative fundamental cost. This work considers an assembly instruction to be the fundamental energy cost that accumulates throughout the lifetime of a program. As discussed in Chapter 5, the energy differences at an individual instruction level for a RISC processor appear to be minimal at the level of observation utilized in this work. Other possible basic costs could be specific instructions (such as dynamic memory allocation), instruction count (regardless of the actual instructions), etc. Any of these approaches would require changing the reduction logic used in the CFG to CR transform to preserve the chosen cost. To achieve an final actual power consumption estimation, some dynamic processor data is required. This data is what couples the static model to real-world hardware. Choosing different fundamental costs will affect the degree of coupling. For example, if a cost system was built around the basic cost being “an instruction”, then one only needs an average energy per instruction estimate, as opposed to instruction specific granularity.

Moving to the hardware, the most obvious extension is experimenting on different hardware. Many other computers exist with the ARM architecture, and the Pi has unfortunate limitations such as lack of good onboard power and performance analyzers. Alternative hardware systems with better onboard power registers may provide alternative ways of measuring power readings throughout instruction

testing. Rerunning the existing instruction testsuite on the Pi with a more robust energy tracking tool may yield more precise results. Reworking of the various instruction benchmarks to better induce or avoid read-after-write hazards would be worthwhile to see differences in energy consumption. Extending the testsuite to further explore more of the ARM v8 instruction set may highlight high power instructions. Alternatively, ARM has an entire other, more energy efficient mode of execution called THUMB [6]. Exploring THUMB assembly may yield interesting results, especially to the world of extremely memory constrained embedded systems where 64 bit computing is not possible.

Beyond improving existing pieces of this work, the next field of exploration is utilizing an assembly level SEA as a form of compiler comparison. Comparing (compiled) high level programming languages is hard, but since they all pass through assembly as an intermediary between development and execution, it serves as a common ground. Using a program's assembly as a shared format, using a SEA could provide further insight into how different language compilers work. Using a static *energy* analyzer specifically has applications to both high compute fields to save energy costs, and mobile devices to increase battery life.

6.2 CLOSING

The initial goal of this work was to produce a tool that aids in producing more energy efficient software. Another broad goal of this work aimed to peel back as many of the layers of abstraction of modern computing as possible.

This work discusses many of the steps from a "top-down" perspective – where one starts at the top of writing code. When trying to run code one has written, it is passed through a compiler (or interpreted, but that is not the focus of this work) that has various sub-steps of scanning, parsing, semantic analysis, optimization,

until assembly is finally emitted. After assembly generation, it is then put through an assembler and finally a linker to produce the end executable file. Then, when running a program, the processor has various procedures that challenge the initial mental model of sequential instruction execution. These complications are primarily things such as pipelining while avoiding hazards, branch prediction, caching, etc. Compiler optimizations and cache coherence models specifically have significantly more depth than was explored in this work. However their existence and discussion in brief do peel back a few layers of abstraction.

This work also explored a “bottom-up” perspective – where one starts at the electronics level. Considering processors are incredibly complex circuits, this thesis explored some fundamental relations in power usage, as well as making the distinction between static and dynamic energy consumption. The electronics perspective quickly encountered challenges due to complexity, but it served as an anchor to explain power usage metrics.

This thesis illuminates many of the steps performed by modern computers that are shrouded by abstraction. The software analysis pieces associated with this thesis demonstrate the process of parsing assembly, various basic instruction tests and helper scripts, and the PUBS solver. The mathematical processes used in cost analysis are interesting and useful, with many applications in execution analysis and other forms of static analysis. The use of Prolog as a complex system model is a novel application of the sometimes discarded logic programming paradigm. Understanding the algorithm of how one performs static energy analysis is valuable. This work faced challenges (see Chapter 5) in developing an arbitrary, fully automatic static energy analysis tool, but these challenges brought new abstractions and considerations to light. Therefore, this work successfully elucidates many assumptions and abstractions between programming and computer energy consumption.

AFTERWORD

I believe this thesis has parallels to my undergraduate career at Wooster. As a high school student, I chose a college with the perspective of “I make it something meaningful, a degree can come from anywhere”. As an underclassman, I struggled in knowing if I was making the most of my time as a student. As an upperclassman, I feel that my highschool perspective was limited. I would now add an extension to my original perspective. **The people** make it meaningful. Without tracing needless hypotheticals, I could have sought a computer science degree anywhere. Academically, the professors I have interacted with have shaped my experiences, not only through the technical content, but by seeing their experiences and how they engage with the field. Socially, my peers have been a tremendous positive influence that have taught me many things about expressiveness, communication, and emotion. While constrained to a year (and honestly, more like a semester and a half), this thesis has had similar ups and downs to my four years at the College. Excitement in getting started, being exposed to new ideas and topics through background literature. Trudging through coursework and literature that is necessary yet not enthralling. Questioning my progress at intermediate points in the process. Reframing the goal to better understand my accomplishments. For my degree, I take pride in its completion, even if mistakes were made in the process. For this thesis, I also am proud of its completion, even if it took a page out of many academic research endeavours and fell short of its original, overly-ambitious goals. Working through this project has been a true challenge due to its length,

complexity, and independence. Even if the software product and experimental results are not the deliverable I envisioned, the journey has been informative. The world of computer science and technology is incredibly broad and incredibly deep. Tricking rocks into thinking really is quite complex. As fascinating as static analysis is, the black magicry of compilers and the seemingly millions of complications gives me pause for further pursuit of this field. Regardless, I learned about the existence of many challenges in computation and can proudly believe that I am closer to understanding how computers actually work, behind the many abstractions in my original mental model. Thank you for reading.

REFERENCES

- [1] Sarah Abdulsalam et al. “Program energy efficiency: The impact of language, compiler and implementation choices”. In: *International Green Computing Conference*. 2014, pp. 1–6. doi: [10.1109/IGCC.2014.7039169](https://doi.org/10.1109/IGCC.2014.7039169).
- [2] Wilhelm Ackermann. “Zum Hilbertschen Aufbau der reellen Zahlen”. In: *Mathematische Annalen* 99, BF01459088 (1928), pp. 118–133. doi: [10.1007/BF01459088](https://doi.org/10.1007/BF01459088). URL: <https://doi.org/10.1007/BF01459088> (page 40).
- [3] E. Albert et al. “Cost Analysis of Java Bytecode”. In: *Programming Languages and Systems*. Ed. by Rocco De Nicola. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 157–172. ISBN: 978-3-540-71316-6 (pages [42–46](#), [86–87](#), [92–93](#)).
- [4] Elvira Albert et al. “Closed-Form Upper Bounds in Static Cost Analysis”. In: *Journal of Automated Reasoning* 46.2 (Feb. 2011), pp. 161–203. doi: [10.1007/s10817-010-9174-1](https://doi.org/10.1007/s10817-010-9174-1). URL: <https://doi.org/10.1007/s10817-010-9174-1> (pages [19](#), [37](#), [39–41](#), [47–48](#), [50–57](#), [61](#), [86–87](#), [92](#)).
- [5] Elvira Albert et al. “Cost Relation Systems: A Language-Independent Target Language for Cost Analysis”. In: *Electronic Notes in Theoretical Computer Science* 248 (2009). Proceedings of the Eighth Spanish Conference on Programming and Computer Languages (PROLE 2008), pp. 31–46. ISSN: 1571-0661. doi: <https://doi.org/10.1016/j.entcs.2009.07.057>. URL: <https://www.sciencedirect.com/science/article/pii/S1571066109002801> (page [41](#)).

- [6] *ARM Developer Guide*. Accessed on October 15, 2023. URL: <https://developer.arm.com/> (pages 12, 22, 29–30, 69, 94).
- [7] Various Authors. *cpupower Manual*. <https://linux.die.net/man/1/cpupower>. 2024 (page 76).
- [8] Abhinav Bhatele et al. “There goes the neighborhood: performance degradation due to nearby jobs”. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 2013, pp. 1–12 (page 3).
- [9] Emily R. Blem, Jai Menon, and Karthikeyan Sankaralingam. “Power struggles: Revisiting the RISC vs. CISC debate on contemporary ARM and x86 architectures”. In: *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)* (2013), pp. 1–12. URL: <https://api.semanticscholar.org/CorpusID:243246> (page 12).
- [10] Bugseng. *Parma Polyhedra Library*. <https://www.bugseng.com/content/parma-polyhedra-library>. Accessed on: 2/18/2024. 2024 (page 61).
- [11] Andreu Carminati, Renan Augusto Starke, and Rômulo Silva de Oliveira. “On the use of static branch prediction to reduce the worst-case execution time of real-time applications”. In: *Real-Time Systems* 54.3 (July 2018), pp. 537–561. ISSN: 1573-1383. DOI: [10.1007/s11241-018-9306-y](https://doi.org/10.1007/s11241-018-9306-y). URL: <https://doi.org/10.1007/s11241-018-9306-y> (pages 28–30).
- [12] Pohua P Chang and W-W Hwu. “Inline function expansion for compiling C programs”. In: *Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*. 1989, pp. 246–257 (page 9).
- [13] *Chilton Computing - Atlas 50th Anniversary*. Accessed on October 15, 2023. URL: <http://www.chilton-computing.org.uk/acl/technology/atlas50th/p005.htm> (page 23).

- [14] William F Clocksin and Christopher S Mellish. *Programming in PROLOG*. Springer Science & Business Media, 2003 (pages 58, 60).
- [15] Thomas H. Cormen et al. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009. ISBN: 0262033844 (page 30).
- [16] George B. Dantzig and S. M. Johnson. *Upper Bounded Variables in Linear Programming*. Tech. rep. Defense Technical Information Center, 1957. URL: <https://apps.dtic.mil/sti/tr/pdf/AD0605077.pdf> (page 87).
- [17] Jonathan G. Koomey. *Estimating Total Power Consumption by Servers in the US and the World*. URL: <http://www-sop.inria.fr/mascotte/Contrats/DIMAGREEN/wiki/uploads/Main/svrpwrusecompletfinal.pdf> (page 70).
- [18] Maurizio Gabbrielli and Simone Martini. *Programming languages: principles and paradigms*. Springer Nature, 2023 (page 10).
- [19] GNU Project. *GCC(1)*. Accessed: 24th March 2024. Free Software Foundation. 2024. URL: <https://man7.org/linux/man-pages/man1/gcc.1.html> (page 81).
- [20] Neville Grech et al. "Static analysis of energy consumption for LLVM IR programs". In: *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems*. ACM, June 2015. DOI: 10.1145/2764967.2764974. URL: <https://doi.org/10.1145%2F2764967.2764974> (pages 20, 73, 76, 89).
- [21] Joel Hestness, Stephen W Keckler, and David A Wood. "GPU computing pipeline inefficiencies and optimization opportunities in heterogeneous CPU-GPU processors". In: *2015 IEEE International Symposium on Workload Characterization*. IEEE. 2015, pp. 87–97 (page 26).
- [22] *In the Data Center, Power and Cooling Costs More Than the IT Equipment It Supports*. URL: <http://www.electronics-cooling.com/2007/02/in-the->

- [data-center-power-and-cooling-costs-more-than-the-it-equipment-it-supports/](#) (page 70).
- [23] Joseph Ingeno. *Software architect's handbook: Become a successful software architect by implementing effective architecture concepts*. Packt, 2018 (page 4).
- [24] Joseph Ingeno. *Software architect's handbook: Become a successful software architect by implementing effective architecture concepts*. Packt, 2018 (page 65).
- [25] ISO/IEC. *ISO/IEC 14882:2011 - Programming languages - C++*. <https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3690.pdf>. Working Draft, Standard for Programming Language C++. 2011 (page 18).
- [26] Neil E. Johnson. *Code size optimization for embedded processors*. Tech. rep. UCAM-CL-TR-607. University of Cambridge, Computer Laboratory, Nov. 2004. doi: [10.48456/tr-607](https://doi.org/10.48456/tr-607). URL: <https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-607.pdf> (page 1).
- [27] Kazhuu. *asm2cfg: A tool for generating control flow graphs from assembly code*. <https://github.com/Kazhuu/asm2cfg>. 2024.
- [28] Steve Kerrison. *Monitoring the energy consumption of a Raspberry Pi with a MAGEEC Wand*. Aug. 2016. doi: [10.13140/RG.2.2.13289.90725](https://doi.org/10.13140/RG.2.2.13289.90725) (pages 23, 32).
- [29] Steve Kerrison and Kerstin Eder. "Energy Modeling of Software for a Hardware Multithreaded Embedded Microprocessor". In: *ACM Trans. Embed. Comput. Syst.* 14.3 (Apr. 2015). ISSN: 1539-9087. doi: [10.1145/2700104](https://doi.org/10.1145/2700104). URL: <https://doi.org/10.1145/2700104>.
- [30] Daniel Kusswurm. *Modern X86 Assembly Language Programming*. Springer, 2014 (page 11).

- [31] Ario Liyan. *What is a Programming Paradigm?* Accessed: 12 February 2024. 2023. URL: <https://medium.com/@Ariobarxan/what-is-a-programming-paradigm-ec6c5879952b> (page 58).
- [32] LLVM Project. *LLDB Documentation: Intel Processor Trace*. https://lldb.llvm.org/use/intel_pt.html. 2024 (page 87).
- [33] *Microsoft's Datacenter Liquid Cooling Innovation*. URL: <https://news.microsoft.com/source/features/innovation/datacenter-liquid-cooling/> (page 70).
- [34] Bernard M. E. Moret. *The Theory of Computation*. Addison-Wesley, 1998. ISBN: 0-201-25828-5 (page 31).
- [35] Abdeen Mustafa Omer. "Energy use and environmental impacts: A general review". In: *Journal of renewable and Sustainable Energy* 1.5 (2009) (page 2).
- [36] David Patterson and John Hennessy. *Computer Organization and Design*. 4th. Morgan Kaufmann, 2009. ISBN: 978-0-12-374493-7 (pages 25, 27–29).
- [37] David A. Patterson and Carlo H. Sequin. "RISC I: A Reduced Instruction Set VLSI Computer". In: *Proceedings of the 8th Annual Symposium on Computer Architecture*. ISCA '81. Minneapolis, Minnesota, USA: IEEE Computer Society Press, 1981, pp. 443–457 (pages 11, 25–26).
- [38] Mandy Patts. *Microprocessor Power Impacts*. URL: https://tbach.web.cern.ch/tbach/thesis/literature/power_density_Pant-DASS.pdf (page 70).
- [39] Darrell Pearce. *Evolution of Programming Languages*. Accessed: 12 February 2024. 2022. URL: <https://www.cs.sjsu.edu/~pearce/modules/lectures/languages2/concepts/evolution.html> (page 59).
- [40] Rui Pereira et al. "Helping Programmers Improve the Energy Efficiency of Source Code". In: *Proceedings of the 39th International Conference on Software Engineering Companion*. ICSE-C '17. Buenos Aires, Argentina: IEEE Press,

- 2017, pp. 238–240. ISBN: 9781538615898. DOI: [10.1109/ICSE-C.2017.80](https://doi.org/10.1109/ICSE-C.2017.80). URL: <https://doi.org/10.1109/ICSE-C.2017.80> (page 2).
- [41] Rui Pereira et al. “Ranking programming languages by energy efficiency”. In: *Science of Computer Programming* 205 (2021), p. 102609. ISSN: 0167-6423. DOI: <https://doi.org/10.1016/j.scico.2021.102609>. URL: <https://www.sciencedirect.com/science/article/pii/S0167642321000022> (page 2).
- [42] Robert G. Plantz. *Introduction to Computer Organization: ARM Assembly Language Using Raspberry Pi*. <https://bob.cs.sonoma.edu/IntroCompOrg-RPi/intro-co-rpi.html>. Accessed: 09-01-2023 (pages 10–11, 13, 19, 22).
- [43] GNU Project. *AS(1) - Linux manual page*. man7.org. 2024. URL: <https://man7.org/linux/man-pages/man1/as.1.html> (page 74).
- [44] GNU Project. *LD(1) - Linux manual page*. man7.org. 2024. URL: <https://man7.org/linux/man-pages/man1/ld.1.html> (page 74).
- [45] The Linux Documentation Project. *objdump - display information from object files*. <https://man7.org/linux/man-pages/man1/objdump.1.html>. Linux. 2024 (page 9).
- [46] C. V. Ramamoorthy and H. F. Li. “Pipeline Architecture”. In: *ACM Comput. Surv.* 9.1 (Mar. 1977), pp. 61–102. ISSN: 0360-0300. DOI: [10.1145/356683.356687](https://doi.org/10.1145/356683.356687). URL: <https://doi.org/10.1145/356683.356687> (page 26).
- [47] *Raspberry Pi Documentation*. Accessed on October 15, 2023. URL: <https://www.raspberrypi.com/documentation/> (pages 13, 28, 69–70).
- [48] Margaret Rouse. *Superscalar Processor Definition*. Accessed: 1 March 2024. 2019. URL: <https://www.techopedia.com/definition/2897/superscalar-processor> (page 29).

- [49] Cagri Sahin et al. “Initial Explorations on Design Pattern Energy Usage”. In: *Proceedings of the First International Workshop on Green and Sustainable Software*. GREENS '12. Zurich, Switzerland: IEEE Press, 2012, pp. 55–61. ISBN: 9781467318327 (page 2).
- [50] James E Smith. “A study of branch prediction strategies”. In: *25 years of the international symposia on Computer architecture (selected papers)*. 1998, pp. 202–215 (page 28).
- [51] Douglas Thain. *Introduction to Compilers and Language Design*. Independently Published, 2020. URL: [URL](#) (pages 6–10, 15, 18–19, 21).
- [52] The Zig Authors. *Zig Programming Language*. <https://ziglang.org/>. 2024 (page 67).
- [53] Vivek Tiwari, Sharad Malik, and Andrew Wolfe. “Power Analysis of Embedded Software: First Step Towards Software Power Minimization”. In: *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on* 2 (Jan. 1995), pp. 437–445. doi: [10.1109/92.335012](#) (pages 41, 64).
- [54] Alan M. Turing. “On Computable Numbers, with an Application to the Entscheidungsproblem”. In: *Proceedings of the London Mathematical Society, Series 2* 42 (1936), pp. 230–265 (pages 31, 33, 48).
- [55] Paul Peter Urone and Roger Hinrichs. *College Physics 2e*. Houston, Texas: OpenStax, July 2022. URL: <https://openstax.org/books/college-physics-2e/pages/1-introduction-to-science-and-the-realm-of-physics-physical-quantities-and-units> (page 70).
- [56] Hans Vandierendonck et al. “By-passing the out-of-order execution pipeline to increase energy-efficiency”. In: May 2007, pp. 97–104. doi: [10.1145/1242531.1242548](#) (page 30).

- [57] Evangelos Vasilakis. *An Instruction Level Energy Characterization of ARM Processors*. Tech. rep. Foundation of Research and Technology Hellas, Institute of Computer Science, 2015. URL: <https://projects.ics.forth.gr/carv/greenvm/files/tr450.pdf> (pages 71–73, 75–76, 79, 90).
- [58] Ben Wegbreit. “Mechanical Program Analysis”. In: *Commun. ACM* 18.9 (Sept. 1975), pp. 528–539. ISSN: 0001-0782. DOI: [10.1145/361002.361016](https://doi.org/10.1145/361002.361016). URL: <https://doi.org/10.1145/361002.361016> (pages 40–41, 92).
- [59] Collin Winter and Tony Lownds. *PEP 3107: Function Annotations*. Python Enhancement Proposal 3107. Python Software Foundation, 2006 (page 18).
- [60] Wolfgang Wögerer. *A survey of static program analysis techniques*. Tech. rep. Citeseer, 2005 (pages 30–31, 33).

