TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Erki Meinberg 192563IASM

C-language Parser & Analyzer for Hardware Performance Estimations

Master's thesis

Supervisor: Priit Ruberg
Ph.D.
Co-supervisor: Peeter Ellervee
Ph.D.

Tallinn 2021

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Erki Meinberg 192563IASM

"Riistvara jõudluse hindamiseks loodud C-keele parser & analüsaator"

Magistritöö

Juhendaja: Priit Ruberg
Ph.D.
Kaasjuhendaja: Peeter Ellervee
Ph.D.

Tallinn 2021

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Erki Meinberg

10.05.2021

Abstract

In preceding works, Priit Ruberg et al. devised methodology for estimating the energy consumption and performance of C-code programs. This thesis describes the implementation of a software system which is meant to automate this methodology. The implementation of two key components is described: that of the atomic operations parser and the analyzer. LLVM and clang libraries were used for the implementation of the parser. The product of this thesis is a functional system which can the number of unique operations a given C-code program executes. The implemented system was then tested on both known example programs used by Ruberg et al., and on the Dhrystone and Whetstone benchmark systems.

This thesis is written in English and is 42 pages long, including 7 chapters, 14 figures and 7 tables.

Annotatsioon

Riistvara jõudluse hindamiseks loodud C-keele parser & analüsaator

Eelnevates töödes kirjeldavad Priit Ruberg jt. uut meetodit C-keeles kirjutatud tarkvara programmide jõudluse ja energiatarbe hindamiseks. Käesoleva töö eesmärk on kirjeldada tarkvarasüsteemi, mis automatiseerib selle meetodi rakendamist.

Töö alguses teostatakse ülevaade eelnevatest töödest ja kirjeldatakse nende poolt välja toodud meetodeid. Selle põhjal kirjeldatakse töö käigus arendatud tarkvarasüsteemi tööpõhimõte ja meetodid. Süsteemi töö põhineb hinnatava C-koodi süntaksi parsimisel ja analüüsimisel.

Peale rakendatavate meetodite kirjeldust seatakse paika süsteemi nõuded. Lisaks selle ka süsteemi struktuur. Struktuuri põhiosad, mida selle töö käigus realiseeriti, on operatsioonide parser ja analüsaator programm. Järgnevates töö osades kirjeldatakse mõlema osa tehniline lahendus.

Operatsioonide parser loodi C++ programmeerimiskeeles, rakendades LLVM ja clang teeke. Analüsaator on Python3-s loodud programm. Valmistükina võetud süsteemi komponendiks on gcov ja gcovr koodikatvust hindavad tööriistad.

Koostatud tarkvarasüsteemi tööd võrreldakse eelnevates töödes kasutatud programmidega. Lisaks sellele katsetatakse süsteemi võimekust informatsiooni eraldada tuntumatest jõudluse hindamisprogrammidest, Dhrystone ja Whetstone.

Lõputöö on kirjutatud Inglise keeles ning sisaldab teksti 42 leheküljel, 7 peatükki, 14 joonist, 7 tabelit.

List of abbreviations and terms

AST	Abstract syntax tree.
JSON	JavaScript Object Notation.
UML	Unified Modelling Language.
GCC	The GNU Compiler Collection.
cc1	GCC's C language compiler and preprocessor.
RTL	Register transfer language.
LLVM	A collection of modular and reusable compiler and toolchain technologies.
clang	A C language front end for the LLVM infrastructure.
gcov	A C language code coverage tool.

Table of contents

1 Introduction 11
1.1 Task 11
1.2 Structure 12
2 Methodology & Previous Work 13
2.1 Estimation of performance & energy consumption on embedded systems 13
2.2 Methodology for automating atomic operation analysis 14
2.3 Example of the Application of the Methodology 15
3 Proposed System Architecture 18
3.1 System Requirements 18
3.2 System Architecture Overview 18
4 Atomic Operations Parser 21
4.1 Requirements 21
4.2 Selection of Tooling 21
4.2.1 pycparser 22
4.2.2 GCC 22
4.2.3 LLVM & clang 23
4.2.4 Conclusion 24
4.3 AST Parsing with LLVM & Clang 25
4.3.1 Simple operations 27
4.3.2 Expression types 28
4.3.3 For-loop headers 30
4.4 Parser output format 31
4.5 Conclusion 32
5 Profiling & Analyzer 33
5.1 gcov & gcovr 33
5.1.1 Branches in gcov 34
5.2 Analyzer 35
5.3 Conclusion 37
6 Evaluation and Results 38

6.1 Full Example 38
6.2 Validation of the Atomic Operation Parser 41
6.3 More Complex Benchmarks 42
6.4 Summary and Future Work 43
7 Summary 45
References 46
Appendix 1 – Non-exclusive licence for reproduction and publication of a graduation thesis 48
Appendix 2 – Source code of the tool 49
Appendix 3 – Matrix multiplication benchmark program 50
Appendix 4 – Other benchmark programs 51

List of figures

Figure 1. Overview key actions of the estimation methodology. 13
Figure 2. An example C-code program for power estimation. 16
Figure 3. Proposed system information flow diagram. 19
Figure 4. GCC's compilation path. 23
Figure 5. The main activity diagram of the atomic operations parser. 26
Figure 6. Main classes required to interface with clang's AST and front-end. 27
Figure 7. Example of an arithmetic expression resulting in implicit conversion. 29
Figure 8. An example of implicit operand promotion in arithmetic expression due to other operand type. 29
Figure 9. A for-loop AST example. 30
Figure 10. For statement parsing activity diagram. 31
Figure 11. Example output of the atomic operations parser. 32
Figure 12. gcovr JSON output structure. 34
Figure 13. gcovr JSON output line object structure. 34
Figure 14. The analyzer's main activity diagram. 36

List of tables

Table 1. Comparison of available libraries for the atomic operations parser. 24
Table 2. Overview of atomic operations extracted by the atomic operations parser. 27
Table 3. Example output of the system. 39
Table 4. Overview of modelled microcontrollers. 40
Table 5. Energy consumption estimation results. 40
Table 6. Comparison of reference atomic operation counts vs. those counted by the system. 41
Table 7. Dhrystone and Whetstone atomic operations counts. 43

1 Introduction

Over the past few years, addressing the energy consumption of various computational systems has become an increasingly important priority. In the case of smaller embedded devices, it would be useful if a developer could estimate the energy consumption of a program on potential target hardware. If this ability could be integrated into the developer's workflow, then it would be easier for them to make decisions that are conscious of energy consumption. This is what motivated Priit Ruberg to research a novel method of energy consumption and performance estimation for such platforms in his doctoral thesis and related works. [1]

The goal of this present work is to build upon that foundation. The methodology presented by Ruberg involves parsing C-code programs by hand and extracting specific atomic operations from them. And then combining that information with the output of a simulator or a code coverage tool. The fact that the application of this methodology has thus far been done by hand limits its application.

1.1 Task

The goal of this work is to have an outline of a software system which is able to automatically perform the analysis process presented in [1]. In short, this thesis describes the implementation of a system which can:

- Automate the C-code parsing and analysis methodologies that were described in [1].
- Extend the application of the aforementioned methodology by being able to parse relevant and known benchmarking programs, such as Dhrystone and Whetstone.

1.2 Structure

The remainder of the thesis is divided into five chapters. Chapter 2 will provide a review of the theoretical background of the work. A methodology for accomplishing the task, as outlined previously, will also be presented there.

Following this, the next three chapters will outline the system as a whole and describe its implementation in depth. This includes outlining the system requirements, establishing the system's structure, conducting review of available tools, and describing nuances of the implementation that were discovered. The system that is described in these chapters was also implemented and can be tested further in this work.

The remaining chapter, Chapter 6, will focus on testing and validating the system. First, a comparison against known test cases from [1] is performed. Then the system's ability to parse more complex programs is evaluated, and the results from it presented. The chapter will close with an overview for potential future work with respect to further developing the tool.

Finally, the work will be summarized in the last chapter.

2 Methodology & Previous Work

This chapter will conduct a review of the work that this one is based on. It will summarize the performance and energy consumption methodology outlined in [1]. The need for the system that is described in this work will then be outlined. This will be followed by an overview of a proposed methodology for automating the performance and energy consumption estimation process.

2.1 Estimation of performance & energy consumption on embedded systems

In [1], a novel solution for estimating the performance and energy consumption of a piece of C-code being executed on a microcontroller is proposed. This method relies on static analysis of the source code, simulation of the source code, and on a database of profiling results. A flowchart of key actions for this methodology is provided in the following figure:

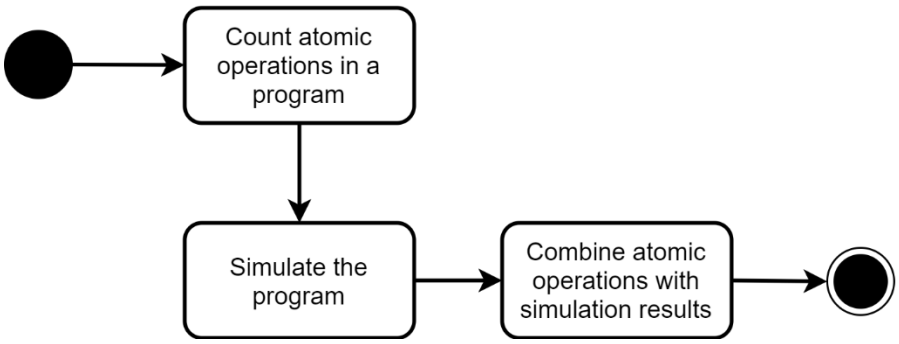

Figure 1. Overview key actions of the estimation methodology.

First, the atomic operations in a given C-code program are counted. Following this, the program is simulated, and the simulation results combined with the earlier results, to obtain the total number of atomic operations executed. This information can then be used to estimate performance figures.

The estimation methodology relies on summing the costs of various atomic operations that are executed in a piece of code. An atomic operation being a specific C-language

syntax operation, such as the add operator "+". The methodology requires the measurement of the cost of each individual atomic operation type, in order to compose a model of a given microcontroller. [1]

For a given program, the energy consumption of a program can be estimated as follows, per equation 2.3 from [1], presented here as equation (1):

$$E_{estimated} = \sum_{i=1}^{n} m_i \times E_{atomic-operation_i} \quad (1)$$

Where $E_{estimated}$ is the estimated energy consumption of the program. n is the number of different (unique) atomic operations present in the program. m_i is the number of repetitions of a given atomic operation. And $E_{atomic-operation}$ is the energy consumption of a given atomic operation.

In order to find the number of repetitions of each atomic operation, two pieces of information are necessary. First, the atomic operations within each C-code source file of a program must be mapped. In [1], this was done manually. And then the program must be simulated using a code coverage tool. Those two pieces of data are then merged, in order to find the total repetitions for each atomic operation.

The manual mapping of atomic operations in a source file is laborious process. And it limited the benchmarks which were used to test out the methodology established in [1]. The merging of the simulation report along with the mapped atomic operations was also done manually in [1]. And this is also a process which can be automated.

Automation of these two processes can enable the wider application of this performance estimation methodology. This will help further evaluating this methodology, by making it applicable for more complex and more common benchmarking programs. And will also allow for its further development and deployment.

2.2 Methodology for automating atomic operation analysis

As established in the previous subsection, there is a need for a system which automates the extraction of atomic operations from C-code source files. This extraction needs to be done by automatically parsing the C programming language.

All of the atomic operations described in the practical experiments of [1] are considered expressions by the C language standard: “An expression is a sequence of operators and operands that specifies computation of a value, [...]” [2, p. 76]. This means that a tool capable of extracting expressions from arbitrary C-code should be capable of extracting all of the necessary atomic operations from it.

This would best be done with a tool which is capable of parsing the C-code files into an abstract syntax tree (AST). The AST can then be traversed, and the required expressions extracted from it. Along with the necessary supplementary information, such as the datatypes involved. Due to the specifics of working with embedded technologies, such a tool should at least be able to parse all code that complies with the C99 ISO standard [3].

These atomic operations must be tied to lines within their respective source code files. The simulation tool then outputs the number of times each source code line is executed. These two pieces of information can then be combined to calculate the total number of repetitions for each atomic operation.

For each atomic operation, the total repetitions $m_{atomic\ operation}$ is calculated as the following sum in equation (2):

$$m_{atomic\ operation} = \sum_{i=1}^{n} k_i \tag{2}$$

Where n represents every source code line that specific atomic operation was encountered on. And k_i is the number of times that specific source code line was executed in the simulated run.

This repetition count can then used together with formula (1) to calculate the performance information of a given C-code program. Provided that the system has access to a previously measured database of atomic operation profiles.

2.3 Example of the Application of the Methodology

To illustrate the application of the methods presented in this chapter, let us consider the following C-code program presented in Figure 2:

```
1. int main() {
2.     int a = 5;
3.     int b = 0;
4.     while (b < 15) {
5.         b = a + b;
6.     }
7. }
```

Figure 2. An example C-code program for power estimation.

The program in question has a total of 5 atomic operations:

1. The assignment of constant value to variable a on line 2.
2. The assignment of the constant value to variable b on line 3.
3. The addition of variable a and b on line 5.
4. The assignment of that sum to variable b on line 5.
5. The loop condition and header on line 4.

Using a code coverage tool, we can determine that the atomic operations 1 and 2 will be executed once, and operations 3, 4, and 5 will be executed three times each. It should also be taken into consideration that operations 1, 2, and 4 are of the same type.

With the atomic operations parsed and mapped out, it is now necessary to calculate the total repetition count of each atomic operation. The comparison and addition atomic operations are both done three times each. And as all of the assignment operations are of the same type, their repetitions are summarized according to equation (2):

$$m_{compare} = 3$$
$$m_{addition} = 3$$
$$m_{assignment} = 3 + 1 + 1 = 5$$

If we then assign some corresponding energy values to each unique type of atomic operation, we can estimate the total energy consumption $E_{estimated}$ of the program by applying equation (1) as follows:

$$E_{estimated} = m_{compare} \times E_{compare} + m_{addition} \times E_{addition} + m_{assignment} \times E_{assignment}$$

A similar calculation will apply for execution times as well, provided that the execution time of each atomic operation is known. What can also be observed is that the only remaining variable for that program is tied to the microcontroller itself: the energy consumption of each atomic operation. This allows for this method to be used for comparing the same program on various hardware pieces.

3 Proposed System Architecture

This chapter will outline the specific architecture of the software system that has been developed in the process of creating this thesis. The software system will implement the automated estimation methodology as explained in 2.2. First, the system requirements for such a system will be outlined. Then the high-level architecture for the system will be presented. This will provide a functional description of each specific software component. These software components will then be explained in depth in later chapters of this work.

3.1 System Requirements

The general requirements for the system are defined as follows:

- the system must be able to parse a project spanning multiple C source files,
- the system must be able to parse all compliant C99 code,
- the system must be able to parse C source files that include common system header files and gracefully handle other external header files,
- the system must produce a list of atomic operations found in the program, associated with their line number in the respective source file,
- the system must be able to extract execution information from a simulated instance of the program under evaluation,
- the system must be able to combine these two outputs with a database of atomic operation information to output an estimation of run time and power consumption.

3.2 System Architecture Overview

The system will be composed of four major components that will be realized individually or already exist. Figure 3 illustrates both the key components of the system, along with the general information flow that was described previously. The components that will be

realized for the purpose of this thesis are marked as bold on the figure. The other components, marked with italics, will be off-the-shelf tools.

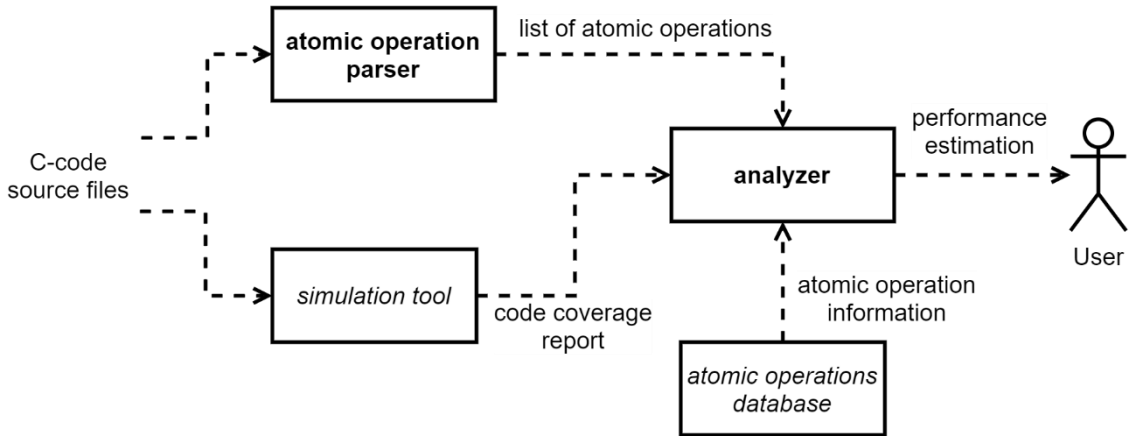

Figure 3. Proposed system information flow diagram.

The first component that will be described in-depth in this work is the atomic operation parser. This component will take the C-code source files and will extract atomic operations from them, as is required by the methodology and system requirements described previously. It will output a list of atomic operations present in the source code of the program being evaluated.

The analyzer will be the component that realizes the formulae described in Chapter 2. It will count the total number of repetitions each atomic operation has during the simulated run. And it will combine this with the atomic operations database in order to perform energy consumption and execution time estimations. For this, it will need to accept input from both the atomic operation parser and the simulation tool.

The simulation tool will be used to execute a simulated run of the program being tested. In [1], gcov was used for this. It is important that the simulation tool used generates a machine-readable report. The analyzer will be responsible for parsing said report.

The atomic operations database is a collection of energy consumption measurements for each atomic operation. As detailed in [1], it details a model of a given microcontroller or platform, for which the estimation is being composed.

From Figure 3, it can be seen that multiple components need to exchange information between each other. JavaScript Object Notation (JSON) will be used as a general information exchange format. This is due to its ubiquity and the availability of tools in various languages for parsing it: libraries for parsing it exist in both C++ and Python.

4 Atomic Operations Parser

As outlined in the previous chapters, the objective of the atomic operations parser is to extract a list of atomic operations present in given C-code source files. This chapter will detail the specific implementation of the tool, including the extra requirements that are specific to it, and how they are addressed.

4.1 Requirements

The key requirements, relevant from 3.1 to the parser, are that the parser must be capable of processing multiple files and these files may include system or external library headers, which should also be successfully processed. The parser must then output this information in a manner such that the analyser is able to consume it. As noted earlier, JSON will be used for this.

As outlined in 2.2, the primary task for the parser is to compile the C-code from source files into an AST, and to then extract specific operations from it. Due to the way that atomic operation performance is measured, all type names should also be resolved to their canonincal form, if possible. To minimize the need for manual removal of duplicates.

For the atomic operation extraction to work, the AST must also be bound to specific source line locations. This will later allow us to properly combine them with the simulator's output.

A further consideration is how easily the source files can be preprocessed using a complaint C language preprocessor. This means both being able to scan include files and being able to parse their contents within the currently active file, as to pick up on things like typedefs and global variable definitions.

4.2 Selection of Tooling

The primary component of the parser will be the library or tool which compiles C-code into an AST. For this purpose, two possible candidate libraries were found: pycparser and

LLVM with the clang frontend. Both tools can generate an AST from C-code source files, thought with different restrictions. The following sections will provide a short overview of both tools and will conclude with a selection of one of them, which will be used for the rest of this work.

4.2.1 pycparser

First, pycparser will be reviewed. As described in the documentation in [4], is a parser for the C language written in the Python programming language. The outlined goal of the tool is to be a fully compliant C99 parser; and it has some support for C11 features.

There exist multiple examples for parsing C-code files and obtaining their AST as a Python code structure. This AST can then be traversed. It further meets requirements in that the parser is able to tie specific AST entries to their location in the source file.

One issue with the pycparser is that by itself, it does not include a preprocessor [4]. As such, another compiler or preprocessor should be used to preprocess the source files prior to giving them over to the parser. In testing with GCC's preprocessor, using the -E switch, issues were observed with the parser being unable to handle all of the preprocessed output. As such, further work would have been needed to make the preprocessed files usable.

The second large issue with the parser is how easily it could be made to use system library headers. To quote the documentation: "While (with some effort) pycparser can be made to parse the standard headers from any C compiler, it's much simpler to use the provided "fake" standard includes in utils/fake_libc_include" [4]. The recommended method of stubbing out system headers would likely be very work intensive.

The remaining benefit of parser is that it is written in Python. Ergo the project setup time would likely be shorter. Initial testing confirmed as much.

4.2.2 GCC

The GNU Compiler Collection (GCC) is a collection of compilers and libraries for various programming languages, including for the C language. The project is entirely open source and free software. GCC is a very long-lived project, which is used in many large software projects, including the GNU operating system. [5]

The GCC component responsible for preprocessing and compiling C-code is called cc1. The cc1 tool has three distinct stages of compilation: the front end, middle end, and back end, as illustrated in Figure 4. The front end is the area where an AST is used as the primary representation. In the middle end, the AST is translated into a register transfer language (RTL) representation. And the final output from the back end is an object representation in assembly language. [6]

Figure 4. GCC's compilation path.

The toolchain is compliant with the latest published C language standards, including the system requirement of C99. As noted earlier, cc1 also contains a preprocessor to handle that stage of compilation. The compiler also constructs a usable AST, as is required by the methodology.

The key issue which arose with attempting to use GCC for this component of the system was that its source code is not easily extensible. Specially when compared to LLVM and clang, which were created with the express purpose of being modular. As such, while the tool itself is likely capable of producing the results required, applying it for the task at hand looked to be more difficult than LLVM.

4.2.3 LLVM & clang

LLVM is an extensible compiler project. It was first presented in [7] as a tool for research and future compilers. The working principle of LLVM is that front ends for specific languages provide a common output, which can then be analyzed using LLVM’s own infrastructure. This has led to the LLVM infrastructure being used for various static analysis tasks, such as those described in [8] and [9]. The LLVM infrastructure has also been applied to static analysis tasks that concern embedded systems, such statically checking for memory safety issues in C-code programs as in [10]. All of this preceding work presents the LLVM infrastructure as a potentially suitable option for this work.

clang is the most commonly used C language frontend for the LLVM tool library. Written in C++, the project is meant to be an extensible and modifiable toolkit for building C and C++ tooling. This includes the ability to preprocess source files, and support for most

GCC extensions. clang has multiple subcomponents which claim to make the process of creating a complete command-line tool along with AST matching relatively straight forward with libTooling, libASTMatcher. [11], [12]

Due to the presence of a complete preprocessor as a plugin, the problem of handling standard library headers and other external headers with clang and LLVM appears to be a non-issue. The preprocessing can simply be done on the input sources prior to the AST extraction phase, in the same executable.

libTooling and libASTMatchers also offer two interfaces for accessing the then-generated AST. The higher-level interface is using libASTMatchers, which has its own DSL for extracting only the relevant components from the AST. The lower-level interface allows one to recursively crawl the AST, much like would be the case with the pycparser library. [13]

While it was initially figured that the high-level interface might permit easier prototyping, it was later learned that it is not all that useful for the purposes of this parser, especially with respect how for-loops need to be parsed. [13]

A major concern with the library was also the fact that both LLVM and the clang libraries needed to be compiled for them to be used. However, the process is well documented, and this would be a one-time setup cost.

4.2.4 Conclusion

Table 1 provides an overview of the analysis carried out in the preceding subsections.

Table 1. Comparison of available libraries for the atomic operations parser.

Tool	**Can parse system headers itself?**	**Supported C language standards**	**Ease of use**
pycparser	No	C99	Easy
GCC	Yes	C99 and later	Hard
LLVM & clang	Yes	C99 and later	Moderate

For the purposes of this work, LLVM with the clang front end was chosen as best choice. The extra work required to enable parsing of system headers with pycparser eliminated it as a valid option. With GCC and LLVM, the modular and tooling oriented approach of LLVM made it the better choice of the two. Otherwise, both are equally capable of accomplishing the required task. This chapter will then continue with detailing the implementation of the atomic operations parser using the LLVM and clang libraries.

4.3 AST Parsing with LLVM & Clang

In this section, the implementation of the atomic operations parser component will be detailed. As a result of the analysis carried out in the previous section, it uses clang and LLVM for both the pre-processing and AST creation. With the AST parsing being written as a custom front-end action for clang.

Figure 5 illustrates the main activity and data flow of the atomic operations parser per C-code source file. The same activity is carried out for every source file that is given to the parser. The segmentation illustrates clearly which parts of the parsing activity are handled by clang's own front-end, and what components are custom code. As can be seen, clang itself manages the proper preprocessing and AST generation for the parser, so the custom extensions only need to consume the ready-made AST.

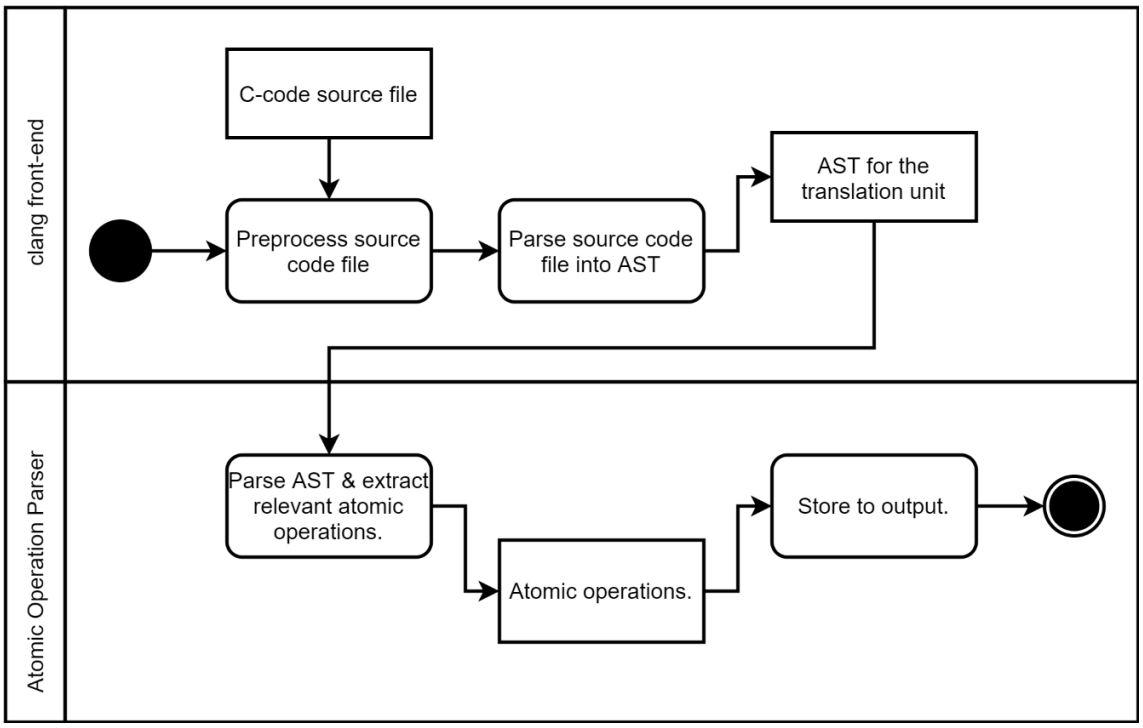

Figure 5. The main activity diagram of the atomic operations parser.

For consuming the AST, as mentioned in the previous section, two possibilities exist with clang and LLVM: using libASTMatchers and writing a custom AST consumer module. Originally, an attempt was made to use the DSL that comes with libASTMatchers, and it did provide a quick start for the project. However, complications arose when attempting to integrate execution branch labelling and more stateful parsing. Due to this, a custom AST consumer was written instead.

In order to interface between clang's AST, a small object hierarchy needed to be created, as shown in Figure 6. The OperationFinderAstAction implements a clang front-end action, which clang's own front-end will instantiate at the appropriate time. This object is then responsible for creating a OperationFinderAstConsumer, onto which clang's frontend passes complete ASTs for every translation unit. The OperationFinderAstVisitor that is attached to it is then handed those ASTs, one at a time, for recursive visitation.

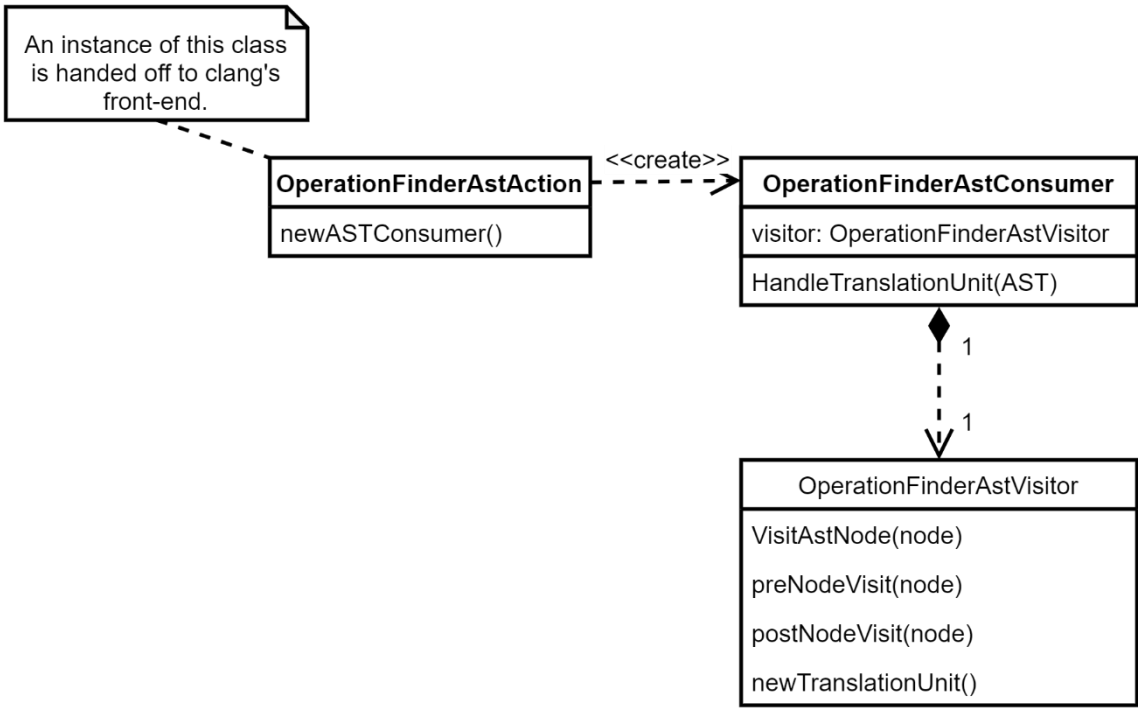

Figure 6. Main classes required to interface with clang's AST and front-end.

The OperationFinderAstVisitor object will recursively descend the AST in a depth-first manner. Appropriate hooks are called out when an AST node of interest is encountered, along with callbacks for node entry and exit. This visitor then extracts required information from the AST nodes, as is described in the following subsections.

4.3.1 Simple operations

With the goal of extracting atomic operations from the now compiled AST, the AST visitor is responsible for picking out the following operations from the AST provided by clang:

Table 2. Overview of atomic operations extracted by the atomic operations parser.

Operation type	**Included operands**	**Saved information**
Binary operators	+, -, /, *, <, >, <=, >=, ==, <<, >>, %	Left hand operand type, right hand operand type, result type, source code line.
Unary operators	!, ++, --, ~	Left hand operand type, result type, source code line.

Operation type	Included operands	Saved information
Array subscript operators	X[Y]	Array (X's) type being accessed, index's (Y's) type, result type, source code line.
Function call expressions	N/A	Result type, source code line.

For the basic operators, the AST visitor saves the following information to the file: types of the left-hand side, the right-hand side (unless unary), and the type resulting from the expression; the operation name; the line within the source file. For function calls, the function's name is saved along with the return type of the call expression, along with the line within the source file. All type names involved must be parsed to their canonincal names, this includes going through type names which are created using typedef's and resolving them. The clang libraries provide the necessary tooling for this.

Following this idea, the extraction of simple operations (primarily arithmetic expressions) can be accomplished without any complex state. The primary consideration there is the extraction of the evaluation type, as will be discussed in 4.3.2. A more complex parsing case exists for for-loops, as will be explained in 4.3.3.

4.3.2 Expression types

One point of consideration for the parser is expression types and rules regarding operand promotion that exist within the C language standard. These rules are covered in section 6.3 of ISO/IEC 9899:2011 [2]. These conversions are done implicitly and will determine what datatype is used to perform the actual atomic operation that is being considered.

For arithmetic operands, the first consideration is how all integer types smaller than int are promoted: "If an int can represent all values of the original type (as restricted by the width, for a bit-field), the value is converted to an int; otherwise, it is converted to an unsigned int. These are called the integer promotions." [2] This effectively means that any arithmetic operation that is done on types smaller than int, such as char, get promoted to int and compiled to the appropriate machine code.

As an example, Figure 7 illustrates a situation where an arithmetic operation (expr 1) involves implicitly converting both operands a and b to type int for the execution of the operation. This is the direct result of the previously specified rule from the standard.

```
int main() {
  char a = 4;
  char b = 6;
  char c = a + b; // expr 1
}
```

Figure 7. Example of an arithmetic expression resulting in implicit conversion.

The second consideration is governed by subsection 6.3.1.8 of [2], regarding the "usual arithmetic conversions" pattern. This pattern is applied whenever deciding what the resulting type of an arithmetic operation is. An example of this behaviour, Figure 8 has the expression marked as "expr 1" result in implicit promotion of the right hand operand to the type long long for the duration of the addition. And thus the result type of the addition expression is long long.

```
int main() {
  long long a = 10;
  int b = 40;
  long long c = a + b; // expr 1
}
```

Figure 8. An example of implicit operand promotion in arithmetic expression due to other operand type.

With consideration given to all of the above, it can be seen that the type as which an arithmetic expression is carried out is not necessarily obvious. And since the prediction methodology of the system relies on knowing exactly what datatype an atomic operation is done with, being able to retrieve the actual resulting expression from mixed typed operations is important.

The clang AST model is capable of retrieving this information, and it is saved as the result type for binary and unary operations, as described in the previous section. This information can then be used in the analyzer for more accurate estimations.

4.3.3 For-loop headers

Another point worth consideration is the handling of for-loops. In equation (2.8) from [1], the execution time of a for-loop is most accurately calculated as follows:

$$t_{loop} = t_{initialization} + (N + 1) \times t_{condition} + N \times t_{increment} + N \times t_{code}$$

where N is the number of iterations accomplished, the total loop execution time is t_{loop}, the loop initialization expression time is $t_{initialization}$, the loop's condition expression time is $t_{condition}$, the loop's increment expression time is $t_{increment}$, and the loop body's execution time is t_{code}. This requires the AST visitor to extract all 4 distinct components from a C language's for-loop and to separate them appropriately.

A section of the AST for a complete for-loop looks as follows:

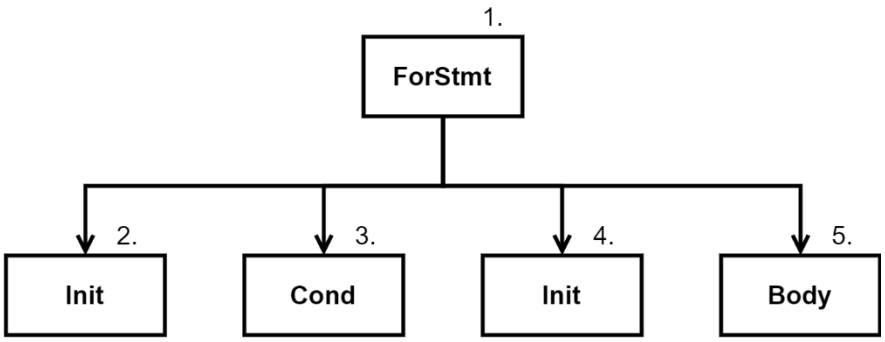

Figure 9. A for-loop AST example.

In Figure 9, the 4 distinct sub-nodes which make up a complete for-loop are shown: the initialization expression (Init), the condition expression (Cond), the increment expression (Inc), and the body compound statement (Body). Along with the visitation order of the nodes: 1 – 5.

The generalized activity for handling this can be seen in Figure 10. The branch number is incremented only if the for-statement's header has both an initialization expression, and either a condition or an increment expression or both. If this pattern is met, then the branch number that gets attached to nodes being extracted is incremented until the initialization expression is has been traversed. Otherwise, there is only one branch: either the initialization expression on its own, or the condition and increment expressions on their own.

This method of parsing guarantees that the branch numbers attached to the operations extracted from the for-loop headers match the output of the simulator. The practical implementation of this parsing ended up also requiring handling a few exceptional cases as well, due to the way the AST is traversed.

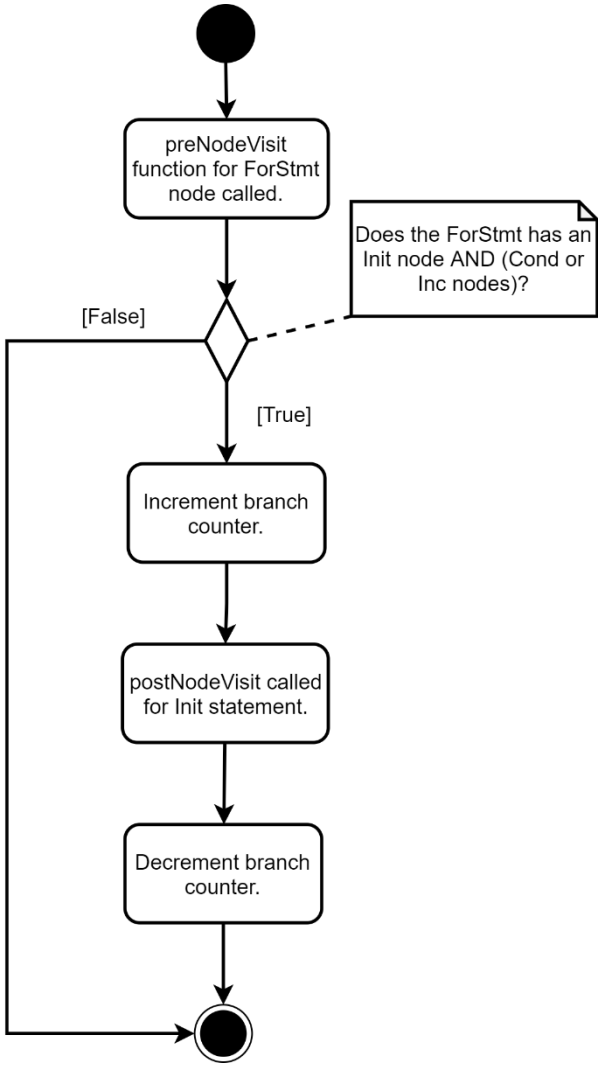

Figure 10. For statement parsing activity diagram.

4.4 Parser output format

The parser's output format, generated by the atomic operation storage object, consists of a JSON dictionary of C-code file names as keys, tied to a list of atomic operations as a corresponding value. An example of this is illustrated in Figure 11:

```
{
  "file_a.c": [
    {
      "branch_number": 0,
      "entry": [atomic operation]
    },
    {
      "branch_number": 1,
      "entry": [atomic operation]
    }
  ]
}
```

Figure 11. Example output of the atomic operations parser.

Each atomic operation in the entry field contains serialized information outlined in Table 2. In addition to this, an additional helper value is added to the JSON object, which contains the type of operation entry. For the purposes of this work, it indicates whether the entry is a basic operation or a function call.

It should be noted that since the list of serialized atomic operation objects is flat, it is possible that multiple atomic operation objects exist with the same line number. This has to be taken into account when detailing the analyzer in the following chapter.

4.5 Conclusion

In this chapter, the operation and behaviour of the atomic operation parser has been detailed. Along with the output format that is going to be ingested by the analyzer. The analyzer shall be the next component that will be detailed.

5 Profiling & Analyzer

This chapter will describe the analyzer, which will combine the results of a simulated run of the program that is being evaluated, and the output of the parser. The output of the analyzer is a total count of every unique atomic operation that was executed during the simulated run, along with an execution time estimate based on these totals. As has been noted in Chapter 2 of this work, gcov is being used as a simulator and code coverage tool, along with gcovr. The following section will detail this process, and the limitations it imposes.

5.1 gcov & gcovr

gcov is a test coverage tool, which is meant to aid in the profiling and analyzing of programs compiled with GCC. A common use-case for the tool would be measuring unit test coverage, for example. And this code coverage feature is the reason for using gcov within the current work as well: it is able to output how many times a given line of C-code is executed, if at all. [14]

In order to be used, the program under evaluation will first have to be adjusted into a state where it can be executed on a regular personal computer. For embedded systems programming, this would mean removing or stubbing out code which deals with peripheral control, for example. This does put limits on the type of code which can be evaluated by the system being described in this work. The code would then be compiled with the "*-ftest-coverage -fprofile-arcs"* flags using GCC and executed.

There are some shortcomings with the methodology of using an external simulator. One issue is that simulating external stimuli is difficult if not impossible. As such, when writing the simulation program, care must be taken to specifically plan out the scenarios that need to be measured.

To better preprocess the output of the gcov tool, a supplementary tool was used: gcovr. gcovr is a utility program in Python, which takes the output of gcov and generates a more

usable report from it. This work uses gcovr to generate a JSON report, which is similar to the output of the atomic operations parser: a list of executed lines that can tied to file names. [15]

The following figure illustrates the JSON output of a gcovr report:

```
{
  "gcovr/format_version": gcovr_json_version
  "files": [
    {
      "file": "file_name.c",
      "lines": [line]
    }
  ]
}
```

Figure 12. gcovr JSON output structure.

Where each line object is structured as follows:

```
{
  "branches": [branch],
  "count": count,
  "line_number": line_number,
  "noncode": noncode
}
```

Figure 13. gcovr JSON output line object structure.

Where “count” is the number of times this line was executed during the simulation run, line_number is the line number from the source file, and noncode is a boolean value indicating whether or not the line in question contained executable code.

5.1.1 Branches in gcov

An interesting issue to address has been gcov’s and gcovr’s reporting of multiple branches of execution on the same line. The way this is handled, in the output, is by adding a special “branches” list to a line entry, as was illustrated in the previous section. The list will contain all the different branches of execution that are present on that one line, along with specific hit “counts”. The sum of the branch hit counts is equal to the hit count of the line containing the branches.

For the purposes of this work, branches have been labelled with their index, as they appear in the JSON report generated by gcovr. The first branch, if present, will be marked as branch 0, the next branch on the same line as branch 1, and so on.

Understanding how these branches correlate to actual execution logic within the C-code is important to increase the prediction accuracy of the analyzer. As already explained in Subsection 4.3.3, the parser itself is able to separate items like the for-loop initialization expression out from the condition and increment expressions. Due to this, the analyzer is required to do the same. A second use-case for understanding branching is the dissection of eagerly evaluated logical expressions, where the time spent on different branches is non-trivial.

For-loop headers will generate two branches: one for the initialization clause, and the other branch for both the condition and the incrementation clause. Testing revealed that, consistently, gcovr on version 4.2 produces a JSON report wherein the initialization branch is always as branch index 1, and the condition and increment branch are presented as branch 0. The parser mimics this behaviour: marking all atomic operations in the initialization statement as branch 1, and the other for-loop header components as branch 0, if they are present.

The other common case for encountering branches is the expressions for complex conditional clauses. A simple logical OR or AND expression will generate 4 different branches from gcov, if used in an if-clause. Two OR or AND expressions joined together will produce 6 different branches. However, mapping these branches back to their semantic meaning has not been successful over the course of this work. As such, the analyzer cannot differentiate which exact branches of logical expressions are being executed. This may lead to error in the execution time prediction if the subexpressions in some logical expressions are non-trivial.

5.2 Analyzer

The analyzer program itself is implemented using the Python programming language. In Figure 3, it accepts the output of both the atomic operations parser and the simulator (gcovr) as inputs. As specified earlier, these are handled as JSON data files. And the

analyzer will then output JSON and a user readable result containing the following information:

- the total number of times each atomic operation is used for the given C-code source files,
- an estimation of the total execution time for the program, using the methodology outlined in the beginning of this work.

The analyzer's primary activity is described in the following UML activity diagram. The activity is executed per source file that exists in both the simulator output file and the atomic operations parser's output file.

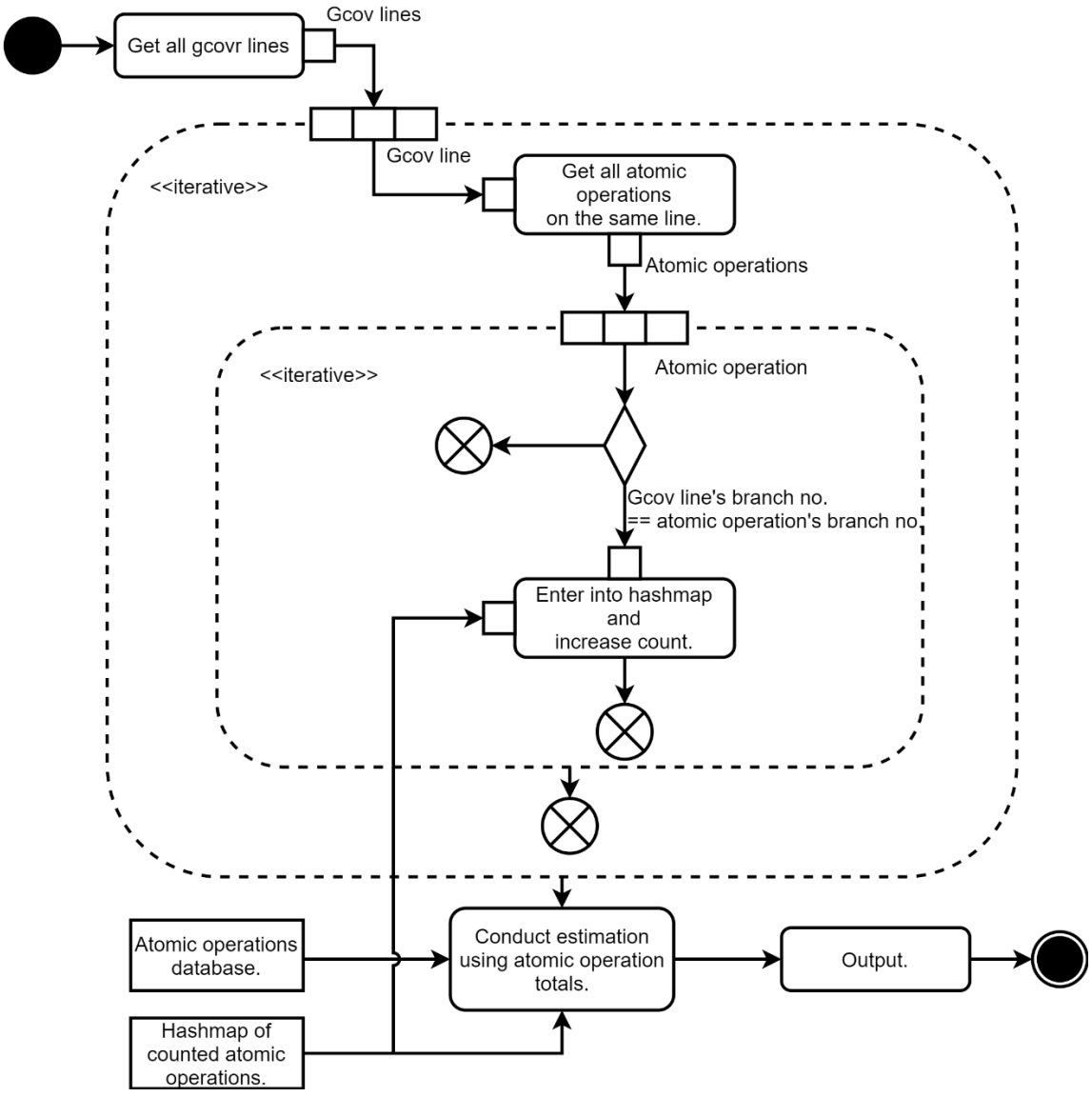

Figure 14. The analyzer's main activity diagram.

Lines from the gcovr simulation report are read and processed one at a time. Due to the way gcovr puts branches as members of the line object, the analyzer will actually create a gcov line object per branch per line. The branch numbers are set appropriately, so that atomic operations can be matched with appropriate branches from gcovr.

A hashmap-like structure is used to hold all of the atomic operations that have been encountered thus far, along with a count. This structure will, by the end of the analyzer's work, contain the total counts for each unique atomic operation. These totals can then be output to the user.

The atomic operations database can be realized as either another JSON file or some other format. It would contain a map of atomic operations with their execution times as measured by experimentation. The total time of execution would then be done by summarizing, as explained in 2.1. This will output the final prediction for the total execution time of the program.

5.3 Conclusion

This chapter detailed the implementation of the analyzer, which is the final component for the performance estimation system. It allows us to provide a list of atomic operations from parsed source code files, and to combine them with a simulator's output in order to conduct an estimation of both execution time and energy usage.

6 Evaluation and Results

This chapter will focus on evaluating the results of the system that was implemented over the course of this thesis work. The goal here is to first validate the system and to then test it against with more complicated and relevant problems. The source code for the system that was developed as a product of this work can be found in Appendix 2.

6.1 Full Example

First, a full example to demonstrate the complete functionality of the system. Let us take the matrix multiplication benchmark program as an example. The code for this program can be found in Appendix 3 and was originally published in [16]. The goal of this example is to illustrate how the system functions, what the intermediate outputs are, and how this can be integrated with the estimation methodology to provide an energy consumption estimation.

The source file of the program is first run through the atomic operations parser. This will extract all of the atomic operations, as was described in Chapter 4. In the case of the matrix multiplication program, the parser will extract 20 unique atomic operations from the source file. These are saved as a JSON file.

Following this, the source program is compiled with GCC and special switches to allow for code coverage reports to be generated. These switches are: "*-fprofile-arcs -ftest-coverage -fPIC -O0*". As explained in Section 5.1, *test-coverage* is necessary for gcov to work, along with the *profile-arcs* flag. The *PIC* flag enables generation of position independent code. And *O0* disables compiler optimizations, which is necessary to generate more accurate coverage output. The position independent code flag may not be strictly necessary for this methodology to work. Running the compiled output will then generate the files necessary for gcov to accurately report code coverage.

The gcovr tool is then ran, with the JSON output format specified and saved to a file. In the case of the current example, the tool will report that 7 lines of the source code were hit. With 6 distinct branches: 2 for each for-loop present in the program.

Finally, both JSON files output from the gcovr tool and the atomic operations parser are given to the analyzer. The analyzer will then output the following information as both JSON and user readable text as shown in Table 3. The total number of unique operations executed during the simulation run was 700.

Table 3. Example output of the system.

Operation name	**Count**	**Additional information**
<	78	Result type: int
++	78	Result type: int
=	19	Result type: int
=	15	Result type: unsigned short
Array subscript	75	Result type: unsigned short
Array subscript	75	Result type: volatile UInt16 [5]
+=	60	Result type: unsigned short
*	60	Result type: int
Array subscript	120	Result type: unsigned short
Array subscript	60	Result type: const UInt16 [4]
Array subscript	60	Result type: const UInt16 [5]

This information can now be used for estimating the power consumption of the program on different platforms for which models exist. For the purposes of this example, limited models for two microcontrollers were created: the STM32F401RE and the ATmega32U4.

A comparison of the two microcontrollers can be found in Table 4. Both microcontrollers were measured on development boards. This specifically caused the STM32's measurements to be higher than expected, as the board also supplied power to an additional programmer component. However, for gauging the accuracy of the estimation, this will not cause issues. The limited models were created using the measurement methodology outlined in Section 3.1. of [1].

Table 4. Overview of modelled microcontrollers.

Microcontroller	**Development board**	**Architecture**	**Clock speed**
STM32F401RE	Nucleo-64 F401RE	32-bit ARM	84 MHz
ATmega32U4	Arduino Micro	8-bit AVR	16 MHz

With the models present, the estimated energy usage of the program could be calculated by the system for both platforms. This can then be compared with the actual energy consumption of the benchmark. The results can be found in Table 5. The error values were calculated with the following equation:

$$e = \frac{|E_{actual} - E_{estimated}|}{E_{actual}} \times 100\%$$

Table 5. Energy consumption estimation results.

Microcontroller	$\boldsymbol{E_{actual}}$ **(mJ)**	$\boldsymbol{E_{estimated}}$ **(mJ)**	$\boldsymbol{e}$
STM32F401RE	11.4	13.2	15.8%
ATmega32U4	101.8	111.5	9.5%

In both cases, the estimated energy consumption was higher than the actual energy consumption of the program. The primary cause of this error is likely down to the manual and less accurate methodology. Specifically, the power usage measurement devices used

were not as accurate as they could have been or were in [1]. An illustration of this is that when just the execution times for the atomic operations are looked at, the errors between the estimation and the actual measured values for both microcontrollers become smaller: 10.6% for the STM32F401RE and 7.3% for the ATmega32U4. This would clearly indicate that a considerable difference comes from how the energy consumption figures were measured. It is likely that the more refined automated modelling methodology described in Section 3.2. of [1] would offer better results.

Overall, this section illustrated how a complete estimation could be acquired for a given program. The atomic operations parser in combination with the analyzer were used to gather the total number of unique atomic operations that were executed by a given program. And then this information was successfully integrated with a model, created according to earlier methodology, to provide a complete energy consumption estimation.

6.2 Validation of the Atomic Operation Parser

This chapter will continue by ensuring that the parser and analyzer are able to successfully parse other benchmarks used in [1]. No energy consumption estimations are done for these more complex programs due to missing models. The objective here is simply to validate the automatic atomic operation extraction and summarization process against known data. The source code for the specific versions of the programs used is found in Appendix 4. The matrix multiplication and FIR filter programs were originally published in [16]. The validation programs were run through the parser and the analyzer, to automatically count the number of total atomic operations found. The results and a comparison with the counts from [1] are shown in the following table:

Table 6. Comparison of reference atomic operation counts vs. those counted by the system.

Benchmark program	**Atomic operations per** [1]	**Atomic operations counted**	**Remarks**
Matrix multiplication	292	700	
FIR filter	2 557	4 213	

Benchmark program	Atomic operations per [1]	Atomic operations counted	Remarks
Image processing	541 208	602 549	Filter size is 50x50.

As can be noted, there is a discrepancy between the atomic operations counted in [1] and by the tools created for this work. There are two key points of difference: first, the previous methodology counted for-loops as a singular atomic operation. Whereas the analysis methods used by the system presented in this work will more accurately split for-loops into multiple atomic operations corresponding to the individual expressions. The second key difference is that the current methodology counts array subscript operations as atomic operations, while the other does not. The output of the system for all three test cases was manually verified as correct.

6.3 More Complex Benchmarks

As more complex benchmarks, the Dhrystone [17] and Whetstone [18] benchmarks were used. The objective of this experiment is to see if the system is capable of parsing these benchmark programs, so that future work may use it as a stepping off point.

Both benchmark programs parse successfully, with no error being created. One key difference between these programs and the benchmarks covered in the preceding work is the usage of goto statements and various standard library function calls. Goto-s are unparsed as of this time, and function calls will require supplementary measurement and profiling to get proper performance estimation.

The atomic operation counts for both of the mentioned benchmarks are presented in the following table:

Table 7. Dhrystone and Whetstone atomic operations counts.

Benchmark program	Atomic operations counted
Dhrystone	1 651 801
Whetstone	92 801

Closer inspection of both of those programs also reveals that they contain calls to external functions. As such, not all atomic operations are completely visible to the parser at this point. In order to gain accurate estimations of these external functions, they would have to profiled individually, or somehow otherwise introspected. However, the current parsing results are good enough for immediate use.

6.4 Summary and Future Work

This chapter has provided a full example demonstrating how the automated system detailed throughout this work can be used to apply the energy consumption methodology of [1] to produce a complete power consumption estimation. Following this, the system was proved able to successfully parse all of the test programs that were originally used to verify this methodology. And finally, it was demonstrated that the system is able to handle more complex programs, including relevant and popular benchmarking programs, such as Dhrystone and Whetstone. This allows for easier application and validation of the performance estimation methodology that was originally proposed in [1].

A possible topic for future work would be automating the analysis of C-standard library functions. Currently, the parser can only extract unique operations if it has access to the source code of a given function. This leaves external functions, such as the standard library functions, as black boxes which have to be individually profiled for increased accuracy during the estimation. Though the parser in its current state does extract and point out calls to them.

Tied to the previous point is also the matter of functions that belong to other external libraries. A code project may include proprietary dependencies, the source code of which is not available for parsing. This would require additional functionality, such as parsing the library's object code, in order to obtain atomic operations. Both estimation and parsing methodology would have to be reviewed to accommodate this.

One final matter that could be addressed would be to look further into the branches that gcov generates as a part of its analysis. This issue was raised in Subsection 5.1.1. While with trivial logical expressions the impact on estimation accuracy may be negligent, logical expressions with longer execution times will increase the estimation error.

7 Summary

Over the course of this work, a methodology for extracting and summarizing atomic operations from C-code files was proposed and detailed. The chosen methodology relies on parsing the AST of a C-code program, and extracting atomic operations from there. The atomic operations would then be automatically combined with the output of the code coverage tool gcov.

The atomic operations parser implementation was done using the LLVM and clang libraries. This was the component responsible for doing the atomic operation extraction. In its final implementation, the atomic operations parser is successfully able to parse both the example programs from [1] and more complex programs. These complex programs include the Dhrystone and Whetstone benchmarks.

Following this, the analyzer's implementation and functionality was described. The analyzer is the component responsible for combining the output of a code coverage tool with that of the atomic operations parser. Combining these two pieces of information allows the analyzer to count all of the individual atomic operations that would be executed during the runtime of the simulated program.

This total number of unique operations, along with their types, can then later be combined with models of various microcontrollers to trivially generate energy consumption and performance estimations. A demonstration of this was offered in Section 6.1, where the energy consumption of a benchmark was estimated using the system, and then compared with the real results.

As a result of this work, there now exist tools for applying the methodology devised in [1] to more complex programs. This is an important step in potentially integrating this methodology in regular development workflows. And it also provides the necessary tools for further testing and refining the methodology in the future.

References

[1] P. Ruberg, “Energy Consumption and Performance Estimation of Embedded Software,” Ph.D. dissertation, Dept. of Computer Engineering, Tallinn University of Technology, Tallinn, 2018.

[2] *Information technology -- Programming languages -- C,* ISO/IEC 9899:2011, 2011.

[3] *Information technology -- Programming languages -- C,* ISO/IEC 9899:1999, 1999.

[4] e. a. E. Bendersky, “pycparser 2.20 readme,” 22 September 2020. [Online]. Available: https://github.com/eliben/pycparser/blob/master/README.rst. [Accessed 22 April 2021].

[5] Free SoftwareFoundation, Inc., “GCC, the GNU Compiler Collection - GNU Project - Free Software Foundation (FSF),” 07 May 2021. [Online]. Available: https://gcc.gnu.org/. [Accessed 7 May 2021].

[6] Free Software Foundation, Inc., GNU C Compiler Internals, Boston: Free Software Foundation, Inc., 2013.

[7] C. Lattner, “LLVM: An Infratstructure for Multi-Stage Optimization,” Computer Science Dept., University of Illinois, Urbana-Champaign, 2002.

[8] D. Dhurjati, S. Kowshik, V. Adve and C. Lattner, “Memory Safety Without Runtime Checks or Garbage Collection,” in *LCTES-2003*, San Diego, CA, 2003.

[9] S. Kowshik, D. Dhurjati and V. Adve, “Ensuring Code Safety Without Runtime Checks for Real-Time Control Systems,” in *CASES 2002*, Grenoble, 2002.

[10] D. Dhurjati, S. Kowshik, V. Adve and C. Lattner, “Memory safety without garbage collection for embedded applications,” *ACM Transactions on Embedded Computing Systems,* vol. 4, no. 1, pp. 73 - 111, 2005.

[11] The Clang Team, “Clang C Language Family Frontend for LLVM,” 2021. [Online]. Available: https://clang.llvm.org/. [Accessed 22 April 2021].

[12] The Clang Team, “LibTooling - Clang 12 Documentation,” 2021. [Online]. Available: https://clang.llvm.org/docs/LibTooling.html. [Accessed 22 April 2021].

[13] The Clang Team, “Introduction to the Clang AST - Clang 12 Documentation,” 2021. [Online]. Available: https://clang.llvm.org/docs/IntroductionToTheClangAST.html. [Accessed 22 April 2021].

[14] Free Software Foundation, Inc., “Gcov Intro (Using the GNU C Compiler Collection (GCC)),” 27 April 2021. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc/Gcov-Intro.html#Gcov-Intro. [Accessed 27 April 2021].

[15] The gcovr authors, “gcovr - gcovr 4.2 documentation,” 19 June 2018. [Online]. Available: https://gcovr.com/en/stable/. [Accessed 27 April 2021].

[16] G. Morton and K. Venkat, “MSP430 Competitive Benchmarking,” Texas Instruments, 2006, revised 2009.

[17] R. P. Weicker, “Dhrystone: a synthetic systems programming benchmark,” *Commun. ACM,* vol. 27, no. 10, pp. 1013-1030, October 1984.

[18] H. J. Curnow and B. A. Wichmann, “A synthetic benchmark,” *Computer Journal,* vol. 19, no. 1, pp. 43-49, 1976.

Appendix 1 – Non-exclusive licence for reproduction and publication of a graduation thesis[1]

I Erki Meinberg

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis “C-language Parser & Analyzer for Hardware Performance Estimations”, supervised by Priit Ruberg and Peeter Ellervee
   1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
   1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.
2. I am aware that the author also retains the rights specified in clause 1 of the non-exclusive licence.
3. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

10.05.2021

1 The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.

Appendix 2 – Source code of the tool

The source code of the tool that was developed over the course of this work can be found in the git repository here, licensed under the MIT license: https://git.skullnet.me/erki/masters-thesis

Appendix 3 – Matrix multiplication benchmark program

```
typedef unsigned short UInt16;

const UInt16 m1[3][4] = {
{0x01, 0x02, 0x03, 0x04},
{0x05, 0x06, 0x07, 0x08},
{0x09, 0x0A, 0x0B, 0x0C}
};

const UInt16 m2[4][5] = {
{0x01, 0x02, 0x03, 0x04, 0x05},
{0x06, 0x07, 0x08, 0x09, 0x0A},
{0x0B, 0x0C, 0x0D, 0x0E, 0x0F},
{0x10, 0x11, 0x12, 0x13, 0x14}
};

int main(void)
{
int m, n, p;
volatile UInt16 m3[3][5];
for(m = 0; m < 3; m++)
{
for(p = 0; p < 5; p++)
{
m3[m][p] = 0;
for(n = 0; n < 4; n++)
{
m3[m][p] += m1[m][n] * m2[n][p];
}
}
}
return 0;
}
```

Appendix 4 – Other benchmark programs

All of the programs used as benchmarks for this thesis can be found in the following git repository, under the “testcases” folder: https://git.skullnet.me/erki/masters-thesis

Check specific file headers for attribution and licensing information.