
THE UNIVERSITY OF EDINBURGH

An Interactive Web Explorer for Compilers

Michael Aylesbury

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2022

Abstract

Intermediate representation is a format utilised by compilers as a step between source code and machine code, which is traditionally difficult for humans to understand. The development of compiler infrastructure introduces the LLVM IR, followed by Multi-Level Intermediate Representation (MLIR) and xDSL, revolutionising IR syntax and structure. As a result, these IRs are human-readable, and human-writeable. This project introduces the Multi-Level Visual Intermediate Representation (MLVIR) and the MLVIR tool, that converts xDSL IR into MLVIR, visualising and further improving the readability of IR. It additionally allows for linear pattern matching and rewriting of MLVIR.

Acknowledgements

I dedicate this project my mother Kris, brother Charlie, and partner Alice. All of you encouraged me to follow what I love and makes me happy, and I couldn't have done this without your love and support. I further dedicate this project to my late Nana, Lynne, who inspired and filled me with curiosity to learn from a young age.

I would like to thank my supervisor, Dr. Tobias Grosser, as well as Mathieu Fehr, both of which provided invaluable guidance, knowledge and help throughout this project.

Table of Contents

Chapter 1

Introduction

1.1 Aim

The aim of this project is to create the first iteration of the MLVIR tool. Successful completion will result in the tool allowing xDSL IR to be input, resulting in a MLVIR representation being generated. This tool will also allow for exploration of pattern matching and application of rewrites.

1.2 Motivation

The MLVIR tool is motivated by the furthering development of compiler intermediate representation. With the emergence of the LLVM IR, inspiring the development of MLIR and subsequently xDSL IR, human-readable and human-writeable IR is now possible. With improved syntax and the release of VSCode IDE features for MLIR, modern IRs such as the .mlir and .xdsl format are bordering on being high-level languages. However, the syntax and large size of such documents leads to difficulties in readability. MLVIR provides a framework that allows effective visualisation of xDSL IR, breaking the code into nested regions and collapsing large numbers of consecutive operations through the use of a nested Tab and Accordion system, powered by ipywidgets. Additionally, pattern matching and rewriting can be specified, applied, and visualised, with the MLVIR representation of the IR updating with the rewrites.

1.2.1 Goals

The goal of this project is to implement MLVIR, an Intermediate Language (IR) visualiser for MLIR. Using Jupyter Notebook, Python and the xDSL toolkit, IRs should be able to be inspected, and pattern matching and rewriting. Furthermore, large IRs should allow pattern rewrites to be applied, with condensed, helpful visualisation of the rewrite locations within the IR.

This allows MLVIR to meet three goals, two of which dependant on the user's familiarity with compilers. Firstly and most importantly, is IR visualisation. Secondly,

education on compiler passes and pattern rewrites, and lastly, debugging of IRs through visualisation.

The MLVIR tool will be evaluated across the following criteria:

- Readability
- Context retention
- IR exploration
- Region visualisation
- Compression of large IR
- Pattern matching and rewriting

1.3 Report Outline

The background frameworks and concepts of the project are explained and explored in chapter 2. Related works are explored in chapter 3, and technologies used, as well as existing visualisations within these technologies are described in chapter 4. The MLVIR interface design is then explored in chapter 5, followed by the implementation and class systems used in chapter 6. The MLVIR tool is then evaluated in chapter 7, and the project is concluded in chapter 8.

1.4 Contribution

- Nested Tab System (NTS)
- Multi-Level Visual Intermediate Representation (MLVIR)
- MLVIR tool

Chapter 2

Background

2.1 Intermediate Representation (IR)

An intermediate representation is any abstract machine language produced by a compiler in between a source and target language, that remains independent of the source language (Walker, n.d.(a)). IRs are used to break translation into separate steps (Toal, 2022a) to allow analysis and optimisation at intermediate stages, and are created in the semantic analysis stage of compilation (Walker, n.d.(a)). Without the use of IR, a new compiler is required for each combination of source language and target language (ibid.). For n source languages and m target languages, $n * m$ compilers are thus required. The use of IR instead allows the separation of compiler front and back-end (ibid.), requiring just n front-ends and m back-ends. This separation of front and back-end allows for extensible, modular compiler design. These representations can be structured using a graph or tree-based structure, can be flat and tuple-based or stack-based (Toal, 2022a). Basic Blocks are the building blocks of compiler IR; they are linear executing sequences of instructions, with one entry point and one exit point. The entry point of blocks are branch targets provided with a label, and the exit point is a branch to another basic block (ibid.).

2.1.1 IR Types

There are five common IR types explained in this section: SSA, AST, CFG, tuple, and stack-based.

2.1.1.1 Single Static Assignment (SSA)-Based IRs

Single Static Assignment (SSA)-Based IRs ensure that every variable is defined, and assigned exactly once (Woodruff, 2020). This simplifies use-define chains (defined as the list of pointers of all definitions of r that reach use u (Walker, n.d.(b))), by only having one definition. Use-def chains are used for IR optimisation, and simplifying these chains allow for easier optimisation. The simplification of these chains is also noticeable to humans, improving IR readability as variables can be effectively tracked

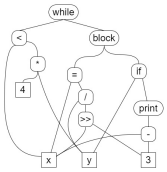

Figure 2.1: CFG-based IR

```
(JUMP, L2)
(LABEL, L1)
(SHR, 3, t0)
(DIV, y, t0, t1)
(COPY / t1, x)
(LE, y, L3)
(SUB, x, 3, t2)
(PRINT, t2)
(LABEL, L3)
(LABEL, L2)
(MUL, 4, y, t4)
(LT, x, t4, t5)
(JNE, t5, L1)
```

```
    goto L2
L1: t0 := 3 >> x
    t1 := y / t0
    x := t1
    if y == 0 goto L3
    t2 := x - 3
    print t2
L3:
L2: t4 := 4 * y
    x := t4 < t5
    if t5 != 0 goto L1
```

Figure 2.2: Tuple-based IR

```
    goto L2
L1:
    load y
    load_constant 3
    load x
    shr
    div
    store x
    load y
    jump_if_zero L3
    load x
    load_constant 3
    sub
    print
L3:
L2:
    load x
    load_constant 4
    load y
    mul
    less_than
    jump_if_not_zero L1
```

Figure 2.3: Stack-based IR

through the IR, improving visualisation. Additionally, having only one definition reduces redundancy in IR, and improves the effectiveness of transformations such as dead code elimination, constant propagation, and register allocation (Walker, n.d.(b)).

2.1.1.2 Abstract Syntax Tree (AST) and Control Flow Graph (CFG)-Based IRs

AST and CFG-based IRs are the most visual intermediate representations. ASTs and CFGs are usually used in conjunction; source code is translated to an AST-based IR, which is converted to a CFG-based IR, which is converted to machine code (Jana, n.d.). ASTs are simplified tree representations of source code, showing the structure of a program, and control flow can be deduced. CFGs are a lower-level representation than ASTs, forming graphs where nodes are basic blocks and edges indicate control flow (ibid.). Inspection of CFGs and ASTs allow determination of the purpose of the source program, providing readable representations, providing helpful visualisation. An example of a CFG can be seen in figure 2.1.

2.1.1.3 Tuple-Based IRs and Stack-Based IRs

Tuple-Based IRs store instructions as a tuple consisting of an instruction, and a number of arguments ranging from zero to three (Toal, 2022d). This can be seen in Figure 2.2. For example the JUMP instruction is listed with parameter L2; the location in the IR to jump to. This concise tuple representation is then usually rendered to a form seen on the right of the tuple form, again in figure 2.2 (Toal, 2022a).

Stack-Based IRs have defined subroutines within code (denoted as L1, L2, L3 in figure 2.3), and each region has a stack of instructions that are executed linearly, finishing when the stack is empty (ibid.).

2.2 The LLVM Project

The LLVM Project is collection of modular, extensible toolchain and compiler technologies (Toal, 2022c) that allow developers to effectively create compiler framework, removing some of the low-level complexity mystifying IR and compiler design. Providing high-level information in real-time to the compiler, the LLVM compiler frame-

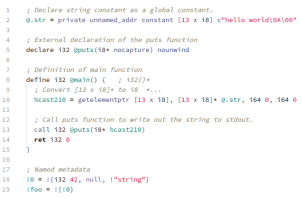

Figure 2.4: LLVM Hello World Code
Darth-Revan, 2022

work is designed for the compilation of multi-language modern applications making use of run-time add-ons, adapting to applications as they evolve throughout compilation. This is defined as lifelong code optimisation (Chris Lattner, 2004).

LLVM presents a new IR format, LLVM IR. This is an SSA-based IR (Toal, 2022b), with a new structure defining modules, each comprised of functions, global variables, and symbol table entries. This structure can be seen in figure 2.4. LLVM IR is lightweight and low-level code representation, aiming to be a universal IR that can clearly represent high-level ideas. There are three main uses for LLVM IR; in-memory compiler IR, on-disk bitcode representation, and most importantly for this project, as an assembly language representation that is readable by humans (ibid.). Despite IR structure being theorised and developed to be interpreted by compilers rather than developers, this IR form is human-readable.

The functions in LLVM are identified by the ”define” keyword, a return type, a name, zero or more arguments, and a list of basic blocks enclosed by curly braces (ibid.). There are many other optional attributes and information that can be defined with a function, making these structures highly extensible. These basic blocks form a CFG, defining the flow of the function. Each LLVM basic block is optionally defined by a label, and is followed by a list of instructions and a terminator instruction. An example of a terminator instruction is return or branch operation (ibid.).

The redeveloped syntax and SSA form allow for a huge leap in readability. An interesting result of the human-readable code is human-writeable code, allowing developers to write IRs. This writing of IRs provides a need for visualisation to better understand the structure and flow of LLVM IRs.

2.3 Multi-Level Intermediate Representation (MLIR)

MLIR is a subproject of LLVM (Toal, 2022c) revolving around extensible, reusable compiler design (MLIR, 2021). MLIR provides the next and biggest leap in compiler design and IR. With new syntax and structure, introducing a modernised operation representation and basic block redesign, MLIR takes a jump further toward high-level syntax, further demystifying the compilation process. This allows developers to have a hand in the compilation process, and even write portable MLIR. As well as the new IR structure, the MLIR framework aids in the design and creation of compiler infrastructure, encouraging optimal compiler design practices, allowing for creation and integration of domain-specific languages (DSLs). The infrastructure intends a set of

Figure 2.5: MLIR IR Structure
MLIR, 2021

modular libraries to be built and interact with one another (MLIR, 2021).

2.3.1 MLIR Structure

MLIR introduces a new infrastructure, using dialects, regions, blocks and attributes, which make use of recursive nesting, as well as an extensible syntax through operations. MLIR is an SSA form hybrid IR (MLIR, n.d.(b)) consisting regions and blocks, made up of operations, greatly simplifying IR structure (ibid.). This structure can be observed in Figure 2.5.

In MLIR, the idea of an operation is introduced, providing an extensible, reusable syntax. Operations cover everything from instructions, to functions, to modules. These highly extensible operations have a unique string identifier, take zero or more operands, return zero or more results, have zero or more successors, and may contain enclosed regions (ibid.). This consistent format becomes quickly recognisable to users and developers, improving the readability of MLIR.

Attributes are defined as compile-time static information, and cover numerical and string values (ibid.).

Regions are a nested structure made up of an ordered list of one or more basic blocks, which are in turn made up of operations. This structure gives the IR its nested form. Regions and blocks can be passed as arguments to operations (ibid.).

Dialects are the mechanism through which MLIR is extended and interacted with. Dialects must define a namespace, and can further define operations, types, and attributes. Multiple dialects can interact and coexist, and there is a focus on progressive lowering within MLIR, allowing for dialect conversion. Progressive lowering is the process of taking a high-level abstraction, and over small increments applying lowering until a low-level abstraction is obtained. This is done through a series of compiler passes and IRs (ibid.).

2.4 xDSL

The xDSL toolkit combines MLIR's revolutionised framework with Python's structure and simple syntax, allowing for further improved and accessible, demystified compiler design. xDSL is an under-development common ecosystem for the building of DSLs. Usually DSLs are implemented in an isolated manner; code and infrastructure is kept separate rather than shared (Jekyll, 2021). This development practice for DSLs

means that longevity and future support may be challenged. As a solution, xDSL proposes a common ecosystem powered and enabled by MLIR, encouraging good compiler design practice. Using LLVM and MLIR allows reuse of existing tooling within these infrastructures, and both are designed to be long-term supported, extensible and reusable, and are regularly updated and managed by a large community. This ensures that the structure of a DSL implemented with this infrastructure will also be long life (Jekyll, 2021). Designers only have to implement the front-end of their DSL, with the supporting back-end being generated underneath by MLIR, and the two managed by xDSL (ibid.). The xDSL toolkit is used in the scope of this project for its operations, pattern matching, and pattern rewriting capabilities, providing the infrastructure on which MLVIR is built. The toolkit allows conversion of MLIR to xDSL useable IR, using the .xdsl file extension.

2.5 Pattern Matching and Rewriting

Pattern matching and rewriting consists of two parts; pattern matching and pattern rewriting, which can be combined or carried out separately. This allows for patterns in IRs to be specified and matched, and for new patterns to replace the matched patterns.

2.5.1 MLIR's Declarative Rewrites

Declarative rewrite (DRR) is a DSL written in LLVM's Tablegen, allowing IR modifications and operation manipulations. Source and target patterns are specified, along with pattern prioritisation parameters. DRR outputs C++ code, which can then be modified for complex cases (MLIR, n.d.(c)).

2.5.2 MLIR's 'pdl' Dialect

A pattern definition language (PDL) dialect exists within MLIR. This allows the specification of patterns and rewrites in MLIR, allowing the rewrite pattern infrastructure to be used directly in MLIR, streamlining verification, optimisation, and general rewriting (MLIR, n.d.(a)).

2.6 MLIR Visual Studio Code Extension

Adding the final high-level development feature to MLIR, the MLIR Visual Studio Code extension allows for IDE features within VSCode (llvm, 2022). Editing IR files is a tedious process due to the low-level nature of the code, despite the advancements in IR syntax. The MLIR extension brings integrated development environment (IDE) features to MLIR files through the MLIR Language Server Protocol. Features such as go-to-definitions and references, and entity information upon cursor hovering are introduced, with syntax highlighting, as well as a live parser and verifier diagnostics (ibid.). The addition of these features further encourage the writing of MLIR by humans, moving the language closer to a high-level language.

Chapter 3

Related Works

3.1 Godbolt's Compiler Explorer

Matt Godbolt's Compiler Explorer (Godbolt, n.d.) is a web application that takes source code as input on the left-hand side (see figure 3.1), and shows its compiler representation on the right-hand side. Many input languages are available, including LLVM IR, Python, and C, with their accompanying compiler, llc, Python 3.10, and gcc. The text editors are split into regions, which are colour-coordinated to show the compilation of source code instructions into machine code instructions. Hovering over one of these regions will highlight the code within, as well as the corresponding input/output code. Additionally pre-typed examples are provided, as seen in figure 3.1.

Figure 3.1 also shows the visualisation of the compilation of LLVM IR before and after passing through the llc compiler, with hovering visualising different regions. The tool does not allow for exploration of MLIR.

Evaluating this against the defined criteria in section 1.2.1, the following can be deduced. GodBolt's Compiler Explorer improves readability by colour highlighting, a feature often harnessed by IDEs. Readability is not improved otherwise. Context retention is unchanged in this tool, as the entire code is shown. That being said, context is provided for compiled code as hovering shows the translation of different regions. However IR exploration is not utilised; LLVM IR is the only IR that is currently available, and this tool does not allow for any exploration greater than simply viewing the IR. The exploration in this tool is instead the transformation from LLVM IR to llc code. Region visualisation is present in this tool, but only because the code is unaltered, showing tab indentation and bracketing. Though the tool is able to compile large IR, it does not provide any compression of large IR input, greatly affecting readability. There is no pattern matching or rewriting mechanism.

3.2 CakeML Compiler Explorer

A similar tool that has a much smaller scope of languages and compilers, but further explore intermediate representation, is the CakeML Compiler Explorer. The Com-

Figure 3.1: Godbolt Compiler Explorer: Example Compilation of an LLVM IR MLIR, 2021

Figure 3.2: CakeML Compiler Explorer (Hjort, Holmgren, and Persson, 2017)

piler Explorer is a web application developed in Facebook's React JavaScript library (Hjort, Holmgren, and Persson, 2017), that takes CakeML source code as text input, and produces output for each compiler pass. The CakeML compiler produces machine code through progressive lowering of source code, with each intermediary step represented by an intermediate language (IL). These ILs are a series of defined IR formats. Expressions are coloured upon cursor hovering, showing the expression in the parent IL, allowing for visualisation and tracing of how an operation traverses the CakeML compiler.

Again, evaluating against the criteria specified in section 1.2.1, readability is the first concern for the CakeML Compiler Explorer. The output of the tool is a series of ILs printed to the output stream as text. This tool has its own representation, instead of just the ILs, but this representation is not readable due to the single-line bracketed printing. Text highlighting is harnessed upon hovering (blue for active expressions, red for ancestor expressions, and green for descendant expressions (ibid.)), improving readability, but the output remains a block of text. This output is not easily readable, especially for a user that is not familiar with the CakeML compiler ecosystem. As the entire ILs are printed, context retention is not a concern; all surrounding information for each expression is printed. However, region visualisation is not handled effectively. Different regions of code are visualised through the use of bracketing. These brackets are difficult to follow, even for small basic ILs such as the provided examples in figure 3.2. IL exploration is provided, but the main focus of this project is to follow and trace how expressions travel through the CakeML compiler, widening the scope to every IL used in the compilation progress. As a result, individual IL visualisation is ineffective, but the tracing mechanism is intuitive and helpful. MLVIR will instead focus on visualisation of a single IR, narrowing the scope and allowing for visualisation better aligned with the project criteria. Given the large output from the one line example code seen in Figure 3.2, it follows that large input would produce even larger output, quickly becoming overwhelming and difficult to follow as no compression is applied. There is no pattern matching or rewriting mechanism.

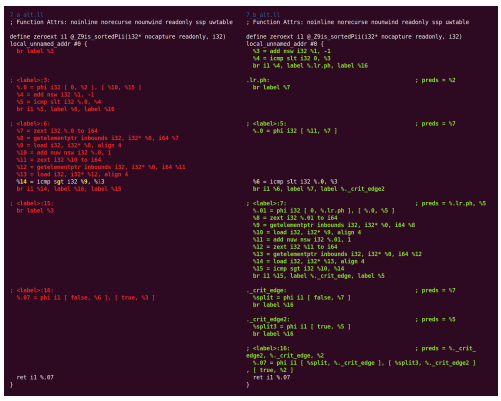

Figure 3.3: LLVM IR Before (left) and After (right) LLVM Optimisation
Regehr, 2018

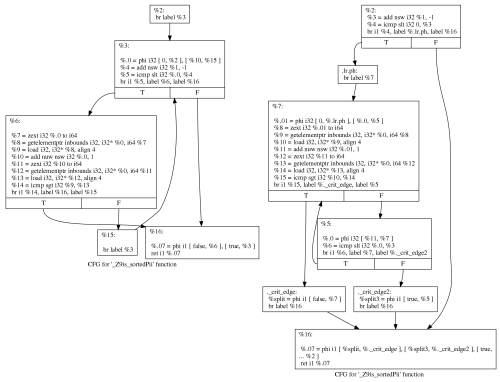

Figure 3.4: CFG Visualisation Before (left) and After (right) LLVM Optimisation
Regehr, 2018

3.3 LLVM opt Tool

Within the LLVM toolchain exists the ”opt” (optimise) tool, taking LLVM IR files as input and running analyses or optimising transformations, and returning the optimised file (LLVM, n.d.(b)). When the -view-cfg option is specified, LLVM IRs are visualised as a CFG (LLVM, n.d.(a)). Specific functions can also be specified, rather than whole IRs. These CFGs outputs are generated using GraphViz (GraphViz, 2021), a graph visualisation software. Other opt visualisations also exist, including call graphs, dominator and post-dominator trees (LLVM, n.d.(a)).

The CFG representation of an LLVM IR before and after optimisation (figure 3.3) can be seen in figure 3.4. This representation is highly readable as information is split into basic blocks, which is less overwhelming for users. These blocks allow context retention through inspection of block content and adjacent blocks, identifiable by arrows showing where control flow changes, and what conditions must be met for such jumps are shown at the bottom of the blocks. Exploration is intuitive thanks to the clear, linear flow of the representation. Although large IR visualisation is possible through this tool, there are still improvements that are possible. Large IRs may lead to many large blocks containing many operations, forcing the user to either zoom out, or scroll due to the lack of compression. As the representation is linear, large input will cause a longer output, increasing the distance from the entry and exit block of the IR. This will affect readability, and additionally affect context retention as the entire IR cannot be visualised at once, though it does still allow for effective exploration of large IR. There is no pattern matching or rewriting mechanism, however.

Chapter 4

Technologies Used

4.1 The Jupyter Notebook

Project Jupyter is both a community and project dedicated to the development of open-source software and services for interactive computing (Jupyter, 2022). The Jupyter Notebook is a web application run and deployed locally on a computer system, that acts essentially as a virtual laboratory notebook. Jupyter Notebooks are computational notebooks that allow code, markdown text, multimedia resources, and code output to be combined into a single notebook (Perkel, 2018). Multiple Jupyter Notebooks were kept and maintained for the development of this project, and the resultant MLVIR tool is contained within such a notebook.

4.2 Python

Due to its compatibility with Jupyter Notebooks, as well as the dependence of ipywidgets and xDSL, Python (Python, 2022) was used for the implementation of MLVIR. The specific Python version used was Python 3.9.7, and this project requires a version greater than or equal to 3.9 due to specific Python 3.9 functionalities being used.

4.3 ipywidgets: Visualisation in Jupyter Notebooks

Data visualisation follows from data collection, and Jupyter Notebooks allow for this. For the scope of this project, Python visualisations within Jupyter Notebooks are focused on. This section explores visualisation libraries in Python, and justifies the selection of ipywidgets for MLVIR.

4.3.1 Matplotlib and Seaborn

Matplotlib (figure 4.1) provides basic plots such as scatter and bar plots, but also offers extensive visualisations for different data types, including array and field plots and statistical plots (Matplotlib, 2022). Seaborn (figure 4.2) also offers some of the same

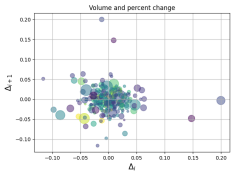

Figure 4.1: Matplotlib Scatter Plot (matplotlib, n.d.)

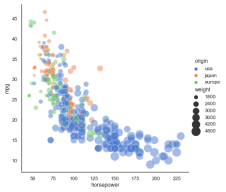

Figure 4.2: Seaborn Scatter Plot (seaborn, n.d.)

Figure 4.3: Plotly Scatter Plot (Plotly, n.d.)

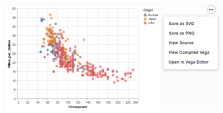

Figure 4.4: Altair Scatter Plot (Altair, n.d.)

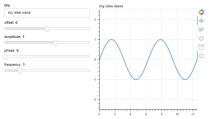

Figure 4.5: Bokeh Sine Wave Plot with Widgets (Bokeh, 2022b)

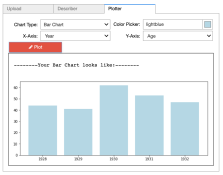

Figure 4.6: Example ipywidgets Data Project GUI (Gün, 2019)

plots, with a more aesthetic visual. These plots include relational, distributional, and categorical plots (Waskom, 2022). The problem with these plots is that they are static and for numerical data, rather than IR structures.

4.3.2 Plotly, Altair, and Bokeh

Three interactive data visualisation libraries are Plotly, Altair, and Bokeh. The Plotly (figure 4.3) library allows interactive visualisation of over 40 plot types (including 3-dimensional plots) (Plotly, 2022). Plotly is highly interactive, showing information such as coordinates and category on hover, also providing graph interaction tools such as zooming, downloading, and isolation of data types. Altair (figure 4.4) is another tool that allows generation of plots, static saving, as well as generated Vega (Vega, 2022) code (Altair, 2022). The most powerful interactive data visualisation library of the three is Bokeh (figure 4.5), allowing JavaScript powered visualisation, with a large range of plots. (Bokeh, 2022a) Panning, zooming and saving tools are provided, similarly to Plotly. Bokeh also provides widgets that can be used to interact with the data in real-time, modifying parameters affecting graphs (Bokeh, 2022b). Bokeh provides infrastructure similar to what is desirable for MLVIR, as parameter modification would be a useful tool for pattern rewriting. But, as with the other mentioned libraries, Bokeh is for visualising numerical data.

4.3.3 Ipyvolume

Another visualisation tool within the Jupyter ecosystem is Ipyvolume, a 3D volume and glyph visualiser (Breddels, 2017). This visualiser is in the early stages of development, but already provides easy implementation of very effective visualisations, allowing saving of images, zooming and exploring like Plotly, as well as widgets allowing different renders such as front or back to be shown, different colours to be

applied, and different parameters to be tweaked. The tool even allows clicking and dragging to smoothly move around visualisations in 3-dimensional space.

4.3.4 Graph Visualisation

Graph visualisation is a technique that was considered, due to the CFG nature of IR. Upon discovering a post on the LLVM Discussion Forums in the MLIR section (Mehta, 2021), another student, Freyam Mehta, was discovered. Freyam was also working on MLIR visualisation, suggesting a graph based method. A meeting was set up with Freyam to discuss such visualisation, and he later presented his ideas in an MLIR Open Meeting. He suggested using a graph visualisation tool, GraphViz (GraphViz, 2021), to represent MLIR. This is a useful visualisation technique, and such a visualisation exists in LLVM, explored in section 3.3. However, this visualisation method did not meet the criteria for this project, as also explained in section 3.3.

4.3.5 ipywidgets

Plotly, Altair, Bokeh, and Ipyvolume are powerful interactive visualisation tools, but all revolve around numerical data. However, taking advantage of an active kernel, JavaScript and HTML embedding (P. Jupyter, 2022), ipywidgets allow for interactive visualisation through standalone widgets. Rather than only visualising numerical data, which does not apply to IRs, ipywidgets allow graphical user interfaces (GUIs) to be designed and interacted with through a series of widgets that can be nested and combined. It was decided that ipywidgets would be used over a graph structure as nesting of widgets could occur, reflecting the structure of MLIR. Further fueling the choice of ipywidgets, there are existing projects that make use of ipywidgets. The 2-dimensional visualisation system bqplot (bqplot, n.d.) uses ipywidgets to interact with pyplot and Object Model. Additionally, ipywidgets has been used for graphical user interface design. The hyperspy project (hyperspy, 2021) creates a GUI entirely out of widgets, and applications such as image viewing galleries can also be created in Jupyter Notebooks (Telenczuk, 2019). Finally, full applications can be seen in Python data projects, covering file input, widget interaction for data processing, followed by data output (Gün, 2019). This can be seen in figure 4.6.

Chapter 5

Design

5.1 MLVIR Design

MLVIR was designed to compliment the design of MLIR. With MLIR's emphasis on nested blocks and regions, it followed for the MLVIR design to simulate this through widget nesting. It was desired that the MLVIR would reside entirely in one container. The only widgets allowing this was the Tab and Accordion widgets. The Tab widget was chosen to be the MLVIR base, with each block and region depicted by a new Tab nested within the existing parent Tab. The design then addressed the graph depiction issues discussed in section 3.3. First of all, the issue of too many nodes in the graph is mitigated through Tab nesting, causing the representation to grow inward, rather than linearly downward. Secondly, the issue of too many operations within a block causing nodes in the graph to become too large, again growing downwards, Accordion widgets were introduced. These Accordions are widgets that act as drop-down containers, that greatly reduce in size when closed. Accordions are automatically added when the number of operations in a tab exceed a value, N, specified by the user. The Accordions default as closed, and contain exactly N operations, clearing up the representation and preventing MLVIRs from becoming too large. Operations are added as Label widgets, displaying the operation type. Currently, only the operation type is displayed, with the exception of two operation types: the func.func type, which displays the type and the function name, and the arith.constant type, which displays the type and constant value. Both of these can be observed in the Gen Stage in figure 5.4. The IR structure deduction, identifying blocks and regions, and actual MLVIR creation is carried out by the MLVIRGeneration, MLVIRPattern and NestedTabSystem classes, described in chapter 6. Two example MLVIRs can be observed: The MLVIR for the IR depicted in figure 5.8 can be seen in the Gen Stage in figure 5.4, and the MLVIR for the IR depicted in figure 5.9 can be seen in figure 5.11.

The final design of the MLVIR tool GUI can be seen in figure 5.1. The simple front-end design imports the back-end, provides an example IR to copy and paste into the tool for the user, and a series of three instructions guiding the user through the tool. The simple Jupyter Notebook allows focus to be placed on the tool, presenting no unnecessary information. The MLVIRbackend Python file imported consists of all the

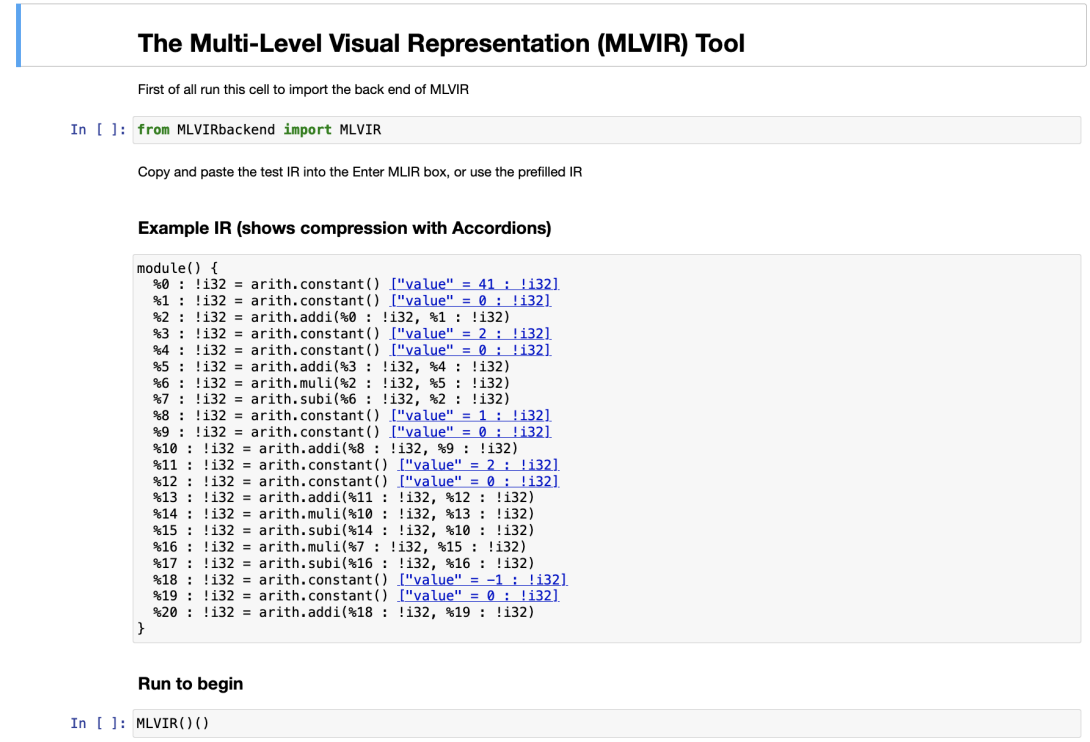

Figure 5.1: MLVIR Tool

classes implemented in chapter 6. This further shadows the MLIR and xDSL intended implementation methodology of splitting front-ends and back-ends, for a modular design. The example IR provided is the same IR as in figure 5.9, and was generated from a metamorphic test case generator provided by Anh Nguyen, a student working on a related thesis.

5.2 Tool Flow and States

Four MLVIR tool states are defined: the Init Stage (figure 5.3), Gen Stage (figure 5.4), Selection Stage (figure 5.5), and Rewrite Stage (figure 5.6). A Rewrite Mode can also be selected by the user. If this mode is not selected, the tool only consists of the Init Stage and Gen Stage. The Init Stage makes a call to the MLVIR class seen in section 6.3. If Rewrite Mode is not selected then the Gen Stage is reached, which is also contained in the MLVIR class. Otherwise, the selection and rewrite stage are reached, making a call to the RewriteMLVIR class in section 6.4. The tool is designed such that the user does not get stuck at any point. That is, any program state can be reached from any other program state through Button pressing and switching to and from Rewrite Mode. This can be seen in the tool's state flowchart in figure 5.2. The cyclic structure of the tool allows for continuous pattern rewriting on IRs across different operations and rewrite patterns.

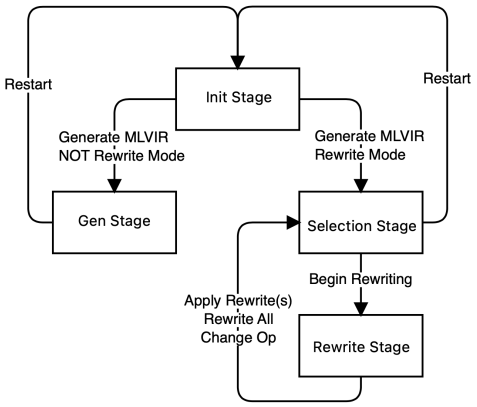

Figure 5.2: MLVIR Tool State Flowchart

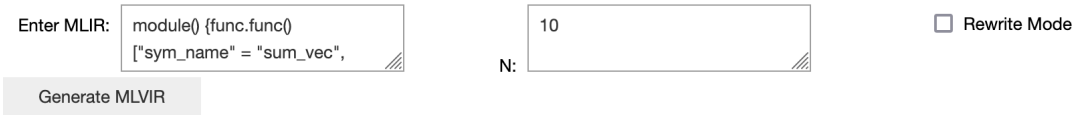

Figure 5.3: Init Stage

5.3 User Interface

Users are able to begin the MLVIR tool by activating the MLVIR virtual environment, launching the MLVIR Jupyter Notebook, running the import cells, and then running the call function of the entry point class: $MLVIR()()$. This simple syntax allows users to focus on the application, removing focus from the Jupyter Notebook cell calling it.

All areas that require user input come with prefilled values for example purposes. The default IR can be seen in figure 5.8 providing an example containing an affine.for loop, showing the MLVIR tools capability to handle nested regions. The default value for N is 10, and the default rewrite operation is an arith.constant of value 42, with SSA value %0. For the ”Select op” SelectMultiple widget, the default value is always the first unique operation in the IR. These values are set in the MLVIR and RewriteMLVIR classes when setting up the widgets. A breakdown of the GUI and widgets is explained in section 5.4.

Due to the cyclic nature of the tool states, the figures of the different states were taken in one run of the application, using the default values. The flow of the figures is as such: beginning at the Init Stage (5.3), the Gen Stage is reached by pressing the ”Generate MLVIR” Button (5.4). The ”Restart” Button is pressed, returning to the Init Stage. The ”Rewrite Mode” Checkbox is pressed, followed by the ”Generate MLVIR” Button, progressing to the Selection Stage (5.5). The ”Begin Rewriting” Button is pressed, leading to the Rewrite Stage (5.6). Finally the ”Rewrite All” Button is pressed, which applies the rewrite pattern to the constant of value 0, replacing this with a constant of value 42 and reverting to the Selection Stage. This new value can be observed in figure 5.7.

Figure 5.4: Gen Stage

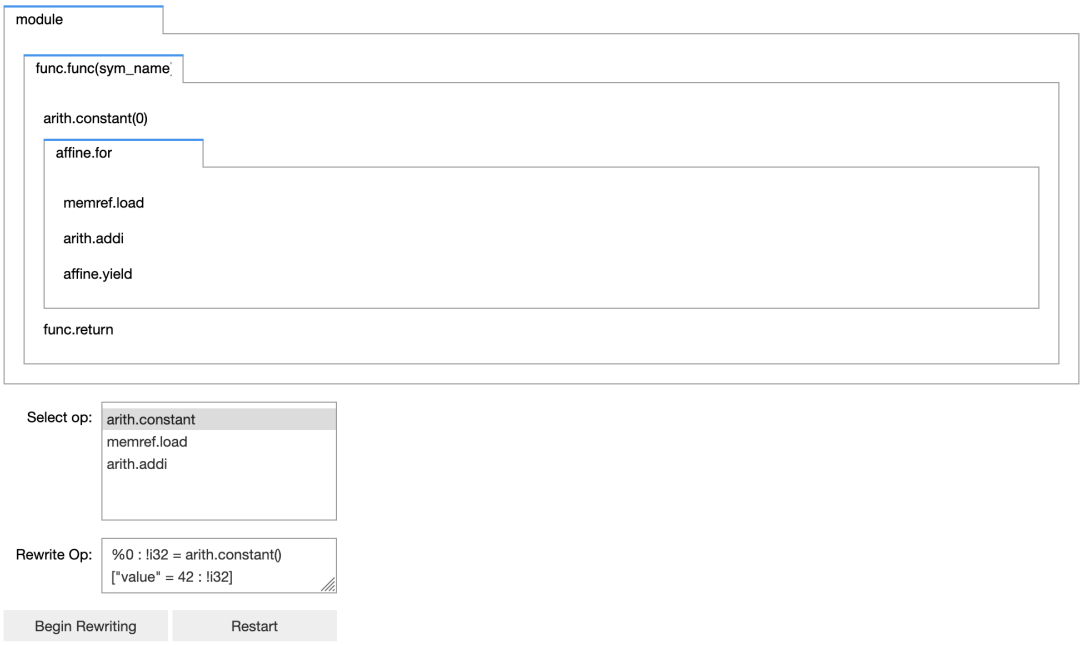

Figure 5.5: Selection Stage

Figure 5.6: Rewrite Stage

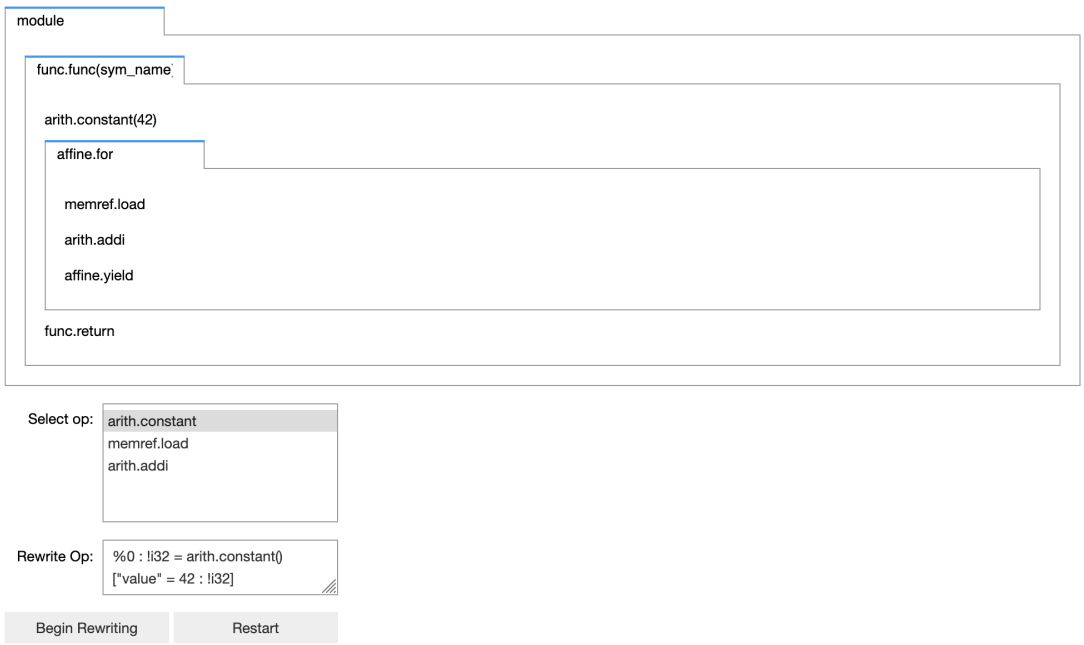

Figure 5.7: Selection Stage (2)

5.4 MLVIR Design and Widget Breakdown

The design of MLVIR is dictated by the NestedTabSystem class discussed in 6.5, and consists of a new Tab for each IR block and region, shadowing the MLIR structure. The tools makes use of a series of widgets at each stage. The full list of ipywidgets used is as follows: Tab, Accordion, Label, Checkbox, SelectMultiple, Textarea, Loadbar and Button widget. Note that all text displayed within an MLVIR uses the Label widget allowing the Labels to be added as children to other widgets. A description of the widgets and design at each stage can be seen below.

5.4.1 Init Stage

The purpose of the Init Stage is to allow users to enter an IR, choose an N value, and to decide whether or not to enter Rewrite Mode. It thus followed to use a Textarea widget for IR input, being the only widget allowing large text to be input. For the N value there were multiple options for entry, such as a SelectionRangeSlider, Dropdown or Text widgets, but ultimately it was decided to also use a Textarea widget. This was chosen as it allows the user to choose any N value rather than producing a finite selection list, and it matches the Textarea used for the IR input. For choosing Rewrite Mode, the Checkbox widget was the clear choice as it allows boolean selection, and Rewrite Mode selection is only ever true or false. To advance to the next stage, a Button was used as they have effective event handling mechanisms, allowing widget closing and opening to happen behind the scenes.

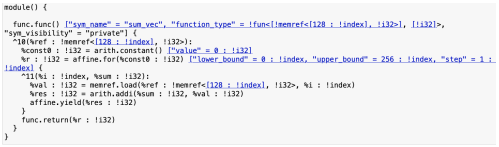

Figure 5.8: Default IR
xdslproject, 2021

```
module() {
  %0 : !i32 = arith.constant() ["value" = 41 : !i32]
  %1 : !i32 = arith.constant() ["value" = 0 : !i32]
  %2 : !i32 = arith.addi(%0 : !i32, %1 : !i32)
  %3 : !i32 = arith.constant() ["value" = 2 : !i32]
  %4 : !i32 = arith.constant() ["value" = 0 : !i32]
  %5 : !i32 = arith.addi(%3 : !i32, %4 : !i32)
  %6 : !i32 = arith.muli(%2 : !i32, %5 : !i32)
  %7 : !i32 = arith.subi(%6 : !i32, %2 : !i32)
  %8 : !i32 = arith.constant() ["value" = 1 : !i32]
  %9 : !i32 = arith.constant() ["value" = 0 : !i32]
  %10 : !i32 = arith.addi(%8 : !i32, %9 : !i32)
  %11 : !i32 = arith.constant() ["value" = 2 : !i32]
  %12 : !i32 = arith.constant() ["value" = 0 : !i32]
  %13 : !i32 = arith.addi(%11 : !i32, %12 : !i32)
  %14 : !i32 = arith.muli(%10 : !i32, %13 : !i32)
  %15 : !i32 = arith.subi(%14 : !i32, %10 : !i32)
  %16 : !i32 = arith.muli(%7 : !i32, %15 : !i32)
  %17 : !i32 = arith.subi(%16 : !i32, %16 : !i32)
  %18 : !i32 = arith.constant() ["value" = -1 : !i32]
  %19 : !i32 = arith.constant() ["value" = 0 : !i32]
  %20 : !i32 = arith.addi(%18 : !i32, %19 : !i32)
}
```

Figure 5.9: Test IR 1

5.4.2 Gen Stage

The Gen Stage focuses on MLVIR generation, and as such is the most simple stage with the fewest widgets. It displays the generated MLVIR, as well as a Button allowing the tool to revert to the Init Stage.

5.4.3 Selection Stage

The Selection Stage first of all generates an MLVIR to allow the user to visualise their IR. This stage needs to allow the user to select operations to match, as well as a rewrite operation. As a finite list of operations can be matched (the list of unique operations within the IR), SelectMultiple, RadioButtons or Dropdown widgets were the options to choose from. The SelectMultiple widget was chosen as it quickly shows the user all operations available for rewriting, without taking up a large amount of space. A Textarea widget was used for the rewrite operation input, with the same reasoning as in section 5.4.1. A forward and backward Button was then implemented, allowing users to move forward to the Rewrite Stage, or backward to the Init Stage.

5.4.4 Rewrite Stage

The Rewrite Stage shows the MLVIR, along with a visualisation of all possible rewrite locations for the chosen operation. Each match will either be rewritten or not rewritten; a boolean choice. As such, a Checkbox is added to a HBox with the Label of the rewriteable operations, appearing next to the Labels, with the caption ”Rewrite” (see figure 5.6). This allows users, if desired, to select which operations they wish to rewrite by clicking the Checkbox. From here, three Buttons are added, with three distinct purposes. These purposes are to allow rewrites to be applied in locations where the Checkbox is ticked, for rewrites to be applied in all possible locations, and for the matching operation to be changed. Clicking either of the first two Buttons changes the tool state to the Selection Stage with the applied rewrite(s), and clicking of the ”Change Op” Button changes the user to the Selection Stage also, but without the application of any rewrites.

Figure 5.10: MLVIR 1 Generated from Test IR 1

Figure 5.11: MLVIR 2 Generated from Test IR 2

5.5 Initial User Interface Design

The initial GUI design consisted of three stages, the Init Stage, Selection Stage, and Rewrite Stage. Rewrite Mode was always on, thus the Gen Stage was not needed at this point. At this point this tool was mostly cyclic, but required the user to restart the tool to reach the Init Stage for a second time. It also did not implement MLVIR. The tool flow is as follows: the Init Stage (figure 5.12), consisting of a Textarea and Button prompted the user to input an IR. Following IR entry and clicking ”Parse IR”, the Selection Stage (figure 5.13) was reached, printing the IR, and presenting the user with a SelectMultiple widget containing the unique operation types, as well as an IntText widget. The IntText widget allows the user to select an integer value that is used as the value of the constant rewrite operation. Clicking the ”Begin” Button proceeded to the Rewrite Stage (figure 5.14). The Rewrite Stage, with the output of the Selection Stage above, reprinted the IR with green arrows before pattern matches, as well as a series of Buttons. For each pattern match within the IR, a Button was displayed with the line number as a label. Additionally, a Change Op Button was displayed. Clicking any location Button would apply a constant rewrite operation of the value entered by the user to the IR, and move the Rewrite Stage again. When the rewrite stage is reached after the first time, the rewritten operations are printed in green, as observed in figure 5.15. After all of the location Buttons were pressed, or the ”Change Op” Button was pressed, the tool moves to the Selection Stage, allowing the user to choose a different operation to match.

For the customised printing of the IR, a custom printing class for xDSL was implemented, the ColourPrinter child class, inheriting from the xDSL Printer class (xdsl,

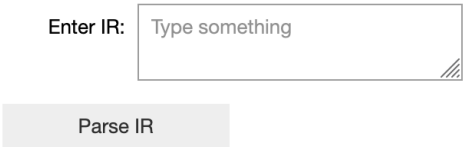

Figure 5.12: Initial Init Stage

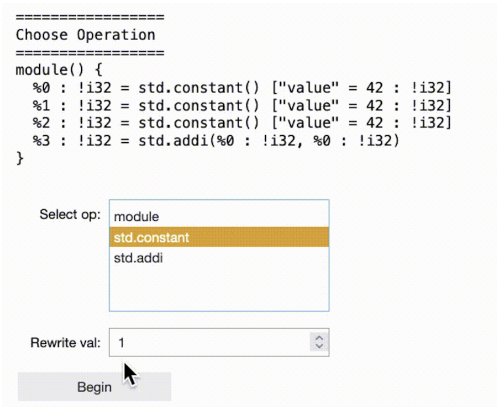

Figure 5.13: Initial Rewrite Stage

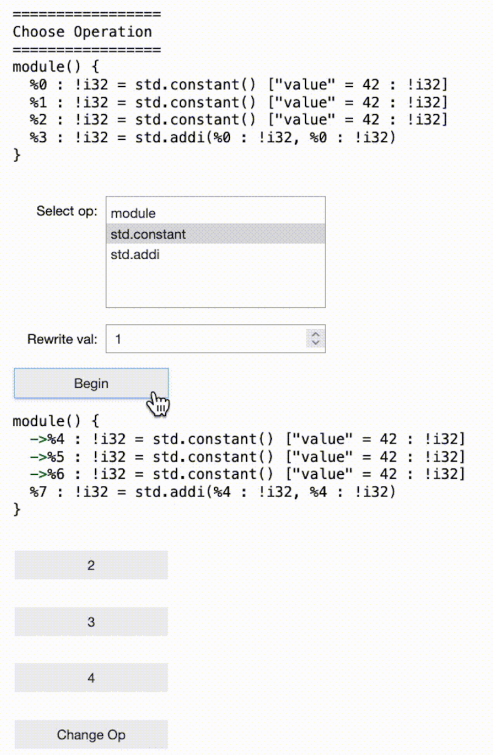

Figure 5.14: Initial Rewrite Stage

Figure 5.15: Initial Rewrite Stage After Performing a Rewrite

n.d.(b)). This would take rewrite locations, and rewritten locations through get functions. Rewrite locations would prefix print operations with a green-coloured right-facing arrow, and rewritten locations would print operations in green using Python print colour codes. After prefixing, a Python *super*() call would be made to the parent Printer class, printing the operation, and afterwards a print colour code would be printed to stop colour printing. As the MLVIR was adopted and IR printing was not used for the next design iteration, this class was ultimately scrapped.

Chapter 6

Implementation

6.1 Overview

The MLVIR tool back-end consists eight classes, four of which are pattern matching data classes; LocationFinderPattern, GetOps, OpsToRewrite, and MLVIRPattern. The other four classes define the functionality and GUI of the tool: MLVIRGeneration, NestedTabSystem, RewriteMLVIR, and MLVIR. All of these classes were implemented from scratch over the course of this project. Note that argument variables prefixed with "(optional)" are Python optional values that default to None if not passed, and class self variables are printed in bold. Functions that take "b" arguments are Button click event handlers, and optional b arguments allow functions to be called regularly, or also on Button click events.

6.2 Pattern Matching for IR Iteration

In the xDSL toolkit, rewrite patterns can be defined as data classes, with a match and rewrite override function called by the PatternRewriteWalker class (xdsl, n.d.(a)). Supplied IRs are passed to the PatternRewriteWalker, and iterated over. As such, four patterns were utilised in the implementation for multiple purposes.

The first pattern, LocationFinderPattern, finds all instances of a supplied operation, and stores it as a value in a dictionary, where its key is its line number. The next pattern, GetOps, adds all unique operation names to a dictionary as keys, with the values as the actual operations. The pattern OpsToRewrite keeps track of the line number in the IR, and takes a line number argument to match. The operation at this line number is returned. The final pattern is the MLVIRPattern. This pattern takes two empty dictionaries and an empty list, keeps track of line number, and fills these data structures with their respective data. The pattern iterates to deduce the structure of the IR, keeping track of blocks and regions in order to generate the MLVIR. All operations are added to a list named ops, and the branch level (level of nesting, synonymous to counting the number of opening brackets ({)) of each line is stored in a dictionary named line_level, where the key is the line number and the value is the branch level. The final dictionary

keeps track of branching operations; operations on which a new region is created or returned to. This dictionary is entitled branches, and every line number has an entry as the key, with a value of either 0, 1, or 2. A value of 1 indicates an increase in branch level, defined for ModuleOp, FuncOp, IfOp and ForOp operations. A value of 2 indicates a decrease in branch level, defined for Return and Yield operations, and any other operation does not change the branch level, and is assigned a value of 0. These pattern matching data classes carry out all iteration over IRs, and are mostly called through the helper functions defined in the class MLVIRGeneration, described in 6.6, with the exception of the GetOps, which is called from the RewriteMLVIR class in section 6.4.

6.3 MLVIR Class

The MLVIR class is the entry point for the tool, generating the initial GUI and taking care of MLVIR generation when rewrite mode is not initiated. This class displays two Textarea widgets, a Checkbox and Button widget. Upon clicking the ”Generate MLVIR” button, this class determines if the ”Rewrite Mode” Checkbox is ticked. If it is, then the RewriteMLVIR class is called. Otherwise, the respective MLVIR is generated, and a restart button is displayed, allowing users to restart the tool. This is done through two functions; a Python __call__ function (allowing the class to be called as a function) (algorithm 1), and function on_click, dealing with the button click event (algorithm 2).

Algorithm 1: MLVIR Class Function 1

Function `__call__`(*self, (optional) b, (optional) nts, (optional) restart*) **:**

- **if** *optional arguments* **then**
  - close nts and close restart
- **end**
- create textbox1 ”Enter MLIR:” and add to new vbox
- create button ”Generate MLVIR” and add to vbox
- add vbox to new hbox
- create textbox2 ”N:” and add to hbox
- create checkbox ”Rewrite Mode” and add to hbox
- display hbox
- on button click, call function on_click

6.4 RewriteMLVIR

Upon rewrite mode being activated, this class is called from the MLVIR class, and takes care of MLVIR generation and rewriting, as well as the accompanying GUI. The function presents the user with a generated MLVIR, a SelectMultiple widget allowing the user to select from the IRs unique operations, a Textarea widget allowing a rewrite pattern to be specified, and two Buttons. The two Buttons allow the user to begin rewriting, or to restart the tool.

Algorithm 2: MLVIR Class Function 2

Function `on_click`(*self, b, textbox1, textbox2, checkbox, hbox*)**:**

```
    if textbox1 empty then
        print "Text area is empty: Enter an MLIR"
        return
    end
    if textbox2 empty then
        print "Text area is empty: Enter an N value"
        return
    end
    if textbox2 value not integer then
        print "N value must be an integer"
        return
    end
    module = parse and verify textbox1 value
    n_value = cast textbox2 value to int
    close hbox
    if checkbox ticked then
        RewriteMLVIR(module, n_value)
    else
        nts, _, _, _ = MLVIRGeneration.GenerateMLVIR(module, n_value)
        restart = create and display button "Restart"
        on restart click, call function __call__ with parameters nts, restart
    end
```

Upon beginning rewriting, Checkbox widgets are added next to all matching operations, allowing them to be selected. From here the user has three choices, in the form of three Buttons. The first is to apply the rewrites to the selected operations, the second to apply the rewrites in all possible locations, and finally the third option is to change the selected operation, reverting to the previous stage. Upon applying rewrites, a new MLVIR replaces the previous one with the changes applied, and the user is placed back in the operation selection stage. To create and interact with the Checkboxes in the MLVIR, this class has interaction with the NTS class. This class is made up of seven functions, with the main functionality occurring in the __init__ and fun functions. Three of the functions are helper functions, reInit, restart and setOpType, and the final two functions, rewrite_actn and rewrite_all, are button event functions.

6.4.1 __init__ and fun Functions

The __init__ function is the first function called in the class, and sets up the GUI for rewriting. The user is presented with a MLVIR for their entered IR, and is able to chose an operation to match, and to input an operation to rewrite matched locations with. Algorithm 3 shows this functionality.

The fun function follows from the init function and is called upon clicking the ”Begin Rewriting” button, and displays the possible rewrite locations for the users chosen operation. The previous widgets are closed, and a new GUI is shown. This allows for the user to select where to apply pattern rewrites. This function is shown in Algorithm 4.

6.4.2 rewrite_all and rewrite_actn Functions

The rewrite_all function is called when the ”Rewrite All” Button is clicked. It applies the supplied operation to all locations that match the chosen operation. It then reverts to the rewrite stage, which generates the new MLVIR. The rewrite pattern themselves are applied to the internal IR, using the xDSL Rewriter and replace_op function to ensure rewrites are valid according to xDSL. This is shown in Algorithm 5.

The rewrite_actn function is called when the ”Apply Rewrite(s)” Button is clicked. The code for this function is identical to the rewrite_actns code, shown in algorithm 5, except for **locs**.keys only uses locations that have value true in the list returned from the get_checkbox_vals() function.

6.4.3 setOpType, reInit, and restart Functions

This function is called upon clicks on the SelectMultiple widget in the fun function, and sets the value of the class variable OP_TYPE to the value chosen by the user. SelectMultiple events behave slightly differently than Button click events; instead of explicitly calling a function and changing state as a Button does by using on_click (due to its boolean nature of clicked/not clicked), SelectMultiple uses observe instead. On each interaction with the widget, the supplied function, setOpType, is called and the value chosen can be used. But, the key difference is that after the function is executed,

Algorithm 3: RewriteMLVIR Class Function 1

Function `__init__`(*self, mod, n*) **:**
- **nts**, brnchs, ops, _ = MLVIRGeneration.GenerateMLVIR(module, n_value)
- **module** = module
- **n** = n
- **ops** = walk IR with GetOps pattern
- **types**, **selectable_ops** = empty list
- **for** *lineNum, op in enumerate(**ops**.keys)* **do**
  - **if** *brnchs[lineNum]=0 and op not in **selectable_ops*** **then**
    - add type of ops[lineNum] to **types**
    - add op to **selectable_ops**
  - **end**
- **end**
- **OP_TYPE** = **types**[0]
- **select** = create and display SelectMultiple from **selectable_ops**
- on **select** click, call function setOpType
- **tbox** = create and display Textarea
- begin = create Button and add to new **hbox**
- on begin click, call function fun
- restart = create Button and add to **hbox**
- on restart click, call function restart
- display hbox

Algorithm 4: RewriteMLVIR Class Function 2

Function `fun`(*self, (optional) b*) **:**
- close init **hbox**
- close init **select**
- close init **tbox**
- close init **nts**
- **rewrite_op** = parse and verify **tbox** value
- locs = FindRewriteLocs(**module**, **OP_TYPE**)
- **if** *no **locs*** **then**
  - print ”No matches to rewrite”
  - __init__(**module**)
- **end**
- **nts**, _, _, _ = MLVIRGeneration.GenerateMLVIR(**module**, **locs**.keys, **n**)
- btn1, btn2, btn3 = create Button and add to new **hbox**
- on btn1 click, call function rewrite_actn
- on btn2 click, call function rewrite_all
- on btn3 click, call function reInit
- display **hbox**

Algorithm 5: RewriteMLVIR Class Function 2

```
Function rewrite_all(self, b):
    close fun hbox
    rw = Rewriter()
    for loc in locs.keys do
        OpToRW = MLVIRGeneration.GetRewriteOp(module, loc)
        rw.replace_op with rewrite_op
    end
    close fun nts
    __init__(module, n)
```

the function containing the widget is returned to. This function is called only by the function fun.

Clicking the ”Change Op” Button in the fun function calls the reInit function. This Button simply closes the HBox and NTS from fun, and calls the __init__ function with the arguments **module** and **n**.

The restart function is called from the __init__ function upon clicking the ”Restart” Button. This function first closes all widgets created in __init__ (hbox, nts, tbox and select), and then calls the __call__ function from the MLVIR class. This is the entry point of the MLVIR tool, and acts as a complete restart.

6.5 Nested Tab System (NTS) Class

The purpose of the NTS is to generate the widget system for an MLVIR using widget nesting. This class creates the actual MLVIR representation. Nesting occurs when new regions of IR are detected, with each new region corresponding to a new nested tab level, referred to as a branch level.

This class takes one optional parameter; n, the number of consecutive operations after which collapsing should occur. If n is not supplied, it defaults to 50. Collapsing takes *N* consecutive label widgets representing operations, and inserts them into a VBox widget, and inserts the VBox into an Accordion widget which defaults as closed. The Nested Tab System consists of 7 functions; init, add_tab, batch_add_text, get_child_at, get_tab, get_tab_from_vbox, and get_checkbox_vals. These functions are used by the GenerateMLVIR class to generate MLVIRs for supplied IRs, first creating an NTS object instance, and then iteratively building up the final MLVIR from the gleaned IR information. Note the first NTS addition must be a Tab, not a batch of text.

6.5.1 __init__ Function

This is the first function called upon creating an NTS object instance, and takes an optional n parameter. This function simply sets the class **N** value to the n value passed in. If one is not provided, the value defaults to **N**= 10.

6.5.2 add_tab Function

This class is called by the GenerateMLVIR function in the MLVIRGeneration class when a new branch level is observed, and nests a Tab within the existing NTS system. The lowest level child, a Tab which is at the current branch level, is obtained. If the Tab has no children, the new Tab is set as the child, otherwise the new Tab is appended to the list of children. This can be observed in algorithm 6.

Algorithm 6: NTS Class Function 1

Function `add_tab(`*self, tab_name*`)`**:**
- **branch_level** += 1
- **if** *tab_name.type="func.func"* **then**
  - append function name to tab_name
- **end**
- **if** ***first_call*** **then**
  - **tab** = create and display Tab tab_name
  - **first_call** = false
- **end**
- child = get_child_at(**branch_level**)
- **if** *child has no children* **then**
  - child.children = create Tab tab_name
- **else**
  - child.children = child.children + create Tab tab_name
- **end**

6.5.3 batch_add_text Function

This function, seen in algorithm 7 adds a batch of text to an existing code region, where the region is specified by a branch level in GenerateMLVIR. This takes care of Accordion collapsing of widgets if the number of Label widgets exceeds the self N value. This is done twice. First of all, if the length of the labs list exceeds N, then an Accordion is created and added to the list add containing N widgets, and the remaining labels are added to add. If the length is equal to N, then an Accordion is added to add containing N widgets. Otherwise, the labels are added to add. Secondly, if the inner_vbox children contains Labels (or HBoxes in the case of pattern matched operations), then collapsing to Accordions is carried out as before. This function also makes a check after the labs list generation to determine if the to_add dictionary value is 1 for the operation key, indicating a pattern match, and if so adds the Label to a HBox along with a Checkbox.

6.5.4 get_child_at, get_tab, and get_tab_from_vbox Functions

The get_child_at function is called in order to get the Tab at either the current branch level or a supplied level. If there is no children in the NTS system, the main parent Tab is returned. Otherwise, the get_tab function is called with no arguments, or the specified level argument, and the returned child is returned.

Algorithm 7: NTS Class Function 2

```
Function batch_add_text(self, to_add, alt_level, (optional) matching):
    if first_call then
        print "You must add a tab first!"
        print "This can be done with nts.add_tab("tab name")"
        return
    end
    tab = get_child_at(alt_level)
    labs = [create Label text for text in texts, if text.type is "arith.constant" append
      constant value to text]
    if matching is passed to function then
        for loc, key in enumerate(to_add.keys) do
            if to_add[key]=1 then
                labs[loc] = create HBox with labs[loc] and Checkbox "Rewrite"
            end
        end
    end
    if labs.len > N then
        add, last = empty list, None
        for i in range(0, labs.len, step_size=N) do
            if not last and last!=0 then
                last=i
            else
                append (create VBox, Accordion, add labs[last:i] to VBox, and
                  VBox to Accordion) to add
            end
        end
        [append i to add for i in labs[last:labs.len]]
    else if labs.len=N then
        add= create VBox, Accordion, add labs to VBox, and VBox to Accordion
    else
        add=labs
    if tab has children then
        inner_vbox = tab.children[0] + add
        acc_loc = None
        for count, i in enumerate(inner_vbox.children) do
            if i.type=Tab then
                acc_loc = count
            end
        end
        if acc_loc then
            labels = [i for i in inner_vbox.children[acc_loc:] if i.type=HBox or
              Label]
        else
            labels = [i for i in inner_vbox.children if i.type=Label]
        end
        if labels.len=N then
            old_children = [i for i in inner_vbox.children if i.type!=Label or HBox]
            old_children += [create VBox, Accordion, add labels to VBox, and
              VBox to Accordion]
            inner_vbox=old_children
        end
    else
        tab.children=[add add to VBox]
    end
```

The get_tab function returns the Tab at either the specified level, or the branch level. Often a VBox will be present, requiring a call to the get_tab_from_vbox function. The child at the correct level is returned. This function works by assuming the last item in the child list is the required Tab due to the linear addition of widgets, as seen in the use of the -1 index to return the last element of a list. This is seen in algorithm 9.

The get_tab_from_vbox function fetches Tab widgets from inside VBoxes due to their constant use. It recurses the VBox children until the current widget is a Tab. Note the children list is reversed, again due to the linear implementation. This function can be observed in algorithm 10.

Algorithm 8: NTS Class Function 3

```
Function get_child_at(self, (optional) level):
    if tab has no children then
        return tab
    end
    if level then
        child = get_tab(level)
    else
        child = get_tab()
    end
    return child
```

Algorithm 9: NTS Class Function 4 get_tab

```
Function get_tab(self, (optional) level):
    child = tab
    if level then
        iters = level
    else
        iters = branch_level
    end
    for i in range(iters) do
        child = child.children[-1]
        if child.type=VBox then
            if get_tab_from_vbox(child) is None then
                return child
            else
                child = get_tab_from_vbox(child)
            end
        end
    end
    return child
```

6.5.5 get_checkbox_vals Function

The get_checkbox_vals function, seen in algorithm 12, fetches the values of the pattern match Checkboxes. It iterates over the children of a Tab to find nested Checkboxes,

Algorithm 10: NTS Class Function 5 get_tab_from_vbox

```
Function get_tab_from_vbox(self, vbox):
    for wid in reverse(vbox.children) do
        if wid.type=Tab then
            return wid
        end
        if widget.type=VBox then
            return get_tab_from_vbox(wid)
        end
    end
```

and appends their values to a list. The if statement splits the function into two cases: if the branch level is zero or non-zero. If the branch level is zero then the child to be fetched is the class **tab** variable, containing a Checkbox nested within a HBox, within a VBox, within an Accordion, within a VBox. Otherwise, branch level is iterated over and the child at each level is fetched. This will result in Checkboxes within a HBox, within a Vbox.

6.6 MLVIRGeneration Class

This contains the tools used to generate MLVIRs, and functions that call the pattern matching data classes. Three functions, GenerateMLVIR, GetRewriteOp, and FindRewriteLocations are contained. GetRewriteOp is a helper function for OpsToRewrite that takes a module and location as input, and returns the operation to be rewritten. The FindRewriteLocations function is a helper function for the LocationFinderPattern, taking a module and an operation as an argument, and returning a dictionaries of the operations found matching the operation, with their line numbers as keys.

6.6.1 GenerateMLVIR Function

This uses a pattern which first traverses the input pattern, finding branching and merging locations, seen in algorithm 12. Three important variables are declared; two dictionaries, line_level and branches, and one list, ops. The dictionaries use line numbers as keys. The values for line_level are integer values indicating nesting level. The values for branches are integers $\in \{0, 1, 2\}$, mentioned in section 6.2. Finally, ops is a list of all the ops in the IR. The aforementioned variables are then iterated over, and the MLVIR itself is generated, using the NTS. When branch_level does not change, operations are added to the to_add dictionary. If a match is identified, the operation is given a value of 1, otherwise a value of 0 is added. For each branching operation, batch adding is triggered if any operations require adding by nts.batch_add_text, and the operation is added as a new tab by calling nts.add_tab. Otherwise, operations are kept added to to_add, unless a non-zero branch_list value is detected, in which case batch adding is triggered. A loading bar widget is displayed during the generation of MVLIR, increasing value from zero to the length of the ops list by one for each iteration. This function finally returns the NTS generated, as well as the three data structures maintained.

Algorithm 11: NTS Class Function 6 get_checkbox_vals

```
Function get_checkbox_vals(self):
    checkbox_values = empty list
    if branch_level=0 then
        for c in tab.children do
            if c.type=VBox then
                for c0 in c.children do
                    if c0.type=Accordion then
                        for c1 in c0.children do
                            if c1.type=VBox then
                                for c2 in c1.children do
                                    if c2.type=HBox then
                                        for c3 in c2.children do
                                            if c3.type=Checkbox then
                                                append c3.value to checkbox_values
                                            end
                                        end
                                    end
                                end
                            end
                        end
                    end
                end
            end
        end
    else
        for i in range(branch_level) do
            child = get_child_at(i) if child has children then
                for c in child.children do
                    if c.type=VBox then
                        for c0 in c.children do
                            if c0.type=HBox then
                                for c1 in c0.children do
                                    if c1.type=Checkbox then
                                        append c1.value to checkbox_values
                                    end
                                end
                            end
                        end
                    end
                end
            end
        end
    end
    return checkbox_values
```

Algorithm 12: MLVIRGeneration Class Function 1 Generate_MLVIR

```
Function GenerateMLVIR(self, module, (optional) locs, (optional) n):
    line_level, branches = empty dictionary
    ops = empty list
    matching = True if locs else False
    walk module with pattern MLVIRPattern , occupying ops, branches, and
     line_level
    create and display loadbar widget
    N = 10 if n is None else n
    nts = NestedTabSystem(N)
    last_level, last_branch = None
    init_batch = True
    to_add = empty dictionary
    for op, lineNum in zip(ops, line_level.keys) do
        loadbar.value = lineNum
        name = op.name
        level = line_level[lineNum]
        brnch = branches[lineNum]
        if brnch == 1 then
            if to_add then
                nts.batch_add_text(to_add, level, matching)
                to_add = empty dictionary
                last_level, last_branch, init_batch = None, None, True
            end
            nts.add_tab(op)
        else
            if init_branch is True then
                last_level = level
                last_brnch = brnch
                init_batch = False
            end
            if brnch==last_brnch then
                if matching and lineNum+1 in locs then
                    to_add[op] = 1
                else
                    to_add[op]=0
                end
            else
            end
            if matching and lineNum+1 in locs then
                to_add[op] = 1
            else
                to_add[op]=0
            end
            nts.batch_add_text(to_add, level, matching)
            to_add = empty dictionary
            last_level, last_brnch = level, brnch
        end
    end
    if to_add then
        nts.batch_add_text(to_add, level, matching)
    end
    close loadbar
    return nts, branches, ops, None
```

Chapter 7

Evaluation

7.1 Evaluation of Initial Design

The initial design shown in section 5.5 provided the backbone of the MLVIR tool, and was the first implementation. Evaluating against the criteria in section 1.2.1 shows the issues that emerged, and how they were mitigated in the next iteration. The ColourPrinter class improved readability of IRs slightly, but essentially the same IR as input was printed. Context retention, IR exploration and region visualisation were not improved, and compression was not applied. However, pattern matching and rewriting was possible. The arrows showing matching patterns were helpful, and rewrites worked. The main issue was that the output grew continually downward and no widgets were removed. This meant for large IRs that the output was very large, and in IRs with many pattern matches, many Buttons were displayed, leading to even larger output. The Buttons also lacked context. They only displayed line numbers, so users would have to scroll and manually count the lines to find the matched operation as no line numbers were printed. Finally, patterns could not be defined for rewriting; all rewrites used constant operations, allowing the constant value to be specified. The next iteration learned from this implementation, and required the following:

- Find a new IR representation with region visualisation
- Apply compression
- Visualise pattern matches more effectively, retaining context
- Allow patterns to be input for rewriting

7.2 Evaluation of MLVIR Tool

Following the initial iteration, the next iteration of the MLVIR tool was implemented with a defined direction. First, this iteration is evaluated against the criteria defined following the first implementation, in section 7.1. A new IR representation was defined, MLVIR. It provides region visualisation through Tab nesting. Compression is applied

through the specification of an N value, collapsing operations within Accordion widgets. Patterns matches are visual, with Checkboxes appearing next to matches, greatly retaining context. Patterns can also be input and used for rewrites in the Selection Stage seen in figure 5.5. Next, evaluating the tool against the initial project criteria specified in section 1.2.1. The MLVIR tool is highly readable, with the nested Tabs isolating blocks and regions, increasing readability, and additionally with Accordions collapsing consecutive operations that exceed the provided N value. Context is retained through surrounding MLVIR information, allowing regions and blocks to be quickly visualised, and control flow to be deduced. Currently, the operation printing loses context. For all operation types other than func.func and arith.constant, just the operation type is printed. This loses the context of the actual content. Region visualisation is strongly utilised as the MLVIR structure simulates the MLIR structure through the NTS. As mentioned evaluating against the previous criteria, compression is utilised through Accordion collapsing, allowing for a compressed, simplified form. This compression also helps with context retention as focus is removed from long blocks of operations, and allows the rest of the IR structure to be observed. The pattern matching and rewriting system allows for any matching pattern to be input, and the user to select an operation within the IR to match. This could be improved, however, by allowing users to provide attributes to match. Currently only operation types can be selected to match, but future iterations could allow for SSA values to be entered, or other attributes to be entered and matched along with the operation types. Additionally, it would be interesting to explore matching of patterns that are longer than one operation, matching regions or blocks, as in this implementation only operations with a branch level value of 0 can be matched and rewritten.

In the Init Stage (figure 5.3) GUI the ”N:” label appears at the bottom of the Textarea widget. A Text widget likely should have been used instead for N value entry, as the input for N is much smaller. This may also prevent the labelling error. If not, layout styling can be harnessed, allowing for manual reconfiguration.

Accordions when closed show a blank rectangle with a drop down arrow, as seen in figure 5.11. Despite context retention by allowing more of the MLVIR structure to be observed, this lost context of the operations inside. A simple fix would be to add a label to the Accordions, such as the range of SSA values contained. For example, the label $\%10 - \%20$ indicates that the Accordion contains 10 operations, with the SSA values in that range.

As seen in the Selection Stage in figure 5.5, a SelectMultiple widget is used to choose the operations to match. With large IRs with many operations, the list of unique operations may become very large, causing this SelectMultiple widget to grow downward. A simple fix would be to use a Dropdown widget instead, which collapses and would allow context to be better retained for large IR.

As well as Textarea input, the tool could make use of the FileUpload widget, allowing users to directly upload .xdsl files, removing a copy and pasting step, and allowing for easier, more natural interaction.

7.3 Development and Implementation Evaluation

When using Tab and Accordion widgets, displaying multiple children within these widgets gave unexpected and unwanted behaviour. In displaying more than one child in Tabs, the first child would be displayed, and any subsequent children would be assigned a blank neighboring Tab at the same branch level. Similarly, with Accordion widgets if more than one child was displayed then a separate Accordion widget would be displayed for each subsequent child. The solution was to introduce VBoxes and HBoxes. For multiple vertical child widgets, VBoxes were used, and for multiple horizontal widgets, HBoxes were used. This mitigated the issue and the desired behaviour was obtained, but it complicated implementation due to further nesting.

An issue that arose with the development of the NTS was with the indexing and altering of children. By the nature of nested systems, the system consisted of a series of widgets within one another, and indexing quickly became difficult with scale. This made it difficult to edit the existing NTS representation, which is why live updating is not utilised, and the system instead deletes and regenerates the MLVIR by modifying the IR, which is not stored in the NTS. Following from the multiple deletions and recreations of MLVIRs, for large input IRs users would have long waiting times upon applying rewrites and switching stages. Obtaining the values from Checkboxes was also difficult due to nesting, and lead to the unattractive implementation of the get_checkbox_vals function observed in algorithm 12. As this was caused by the implementation of ipywidgets, in order to mitigate this the NTS class should be modified to contain the IR, and make use of an internal representation of the MLVIR in order to edit existing MLVIRs.

An unexpected issue that arose was that in generation of large MLVIRs, widget generation took a long time. When testing a $4,000$ line IR, MLVIR generation took multiple minutes, and it was determined this was due to the generation of the Label widgets. Label widgets were the most frequently generated widgets as most operations have a branch level of 0, and on each iteration of GenerateMLVIR a call to an add_text function was made. This method of generation was ineffective once widget generation was determined to be expensive, and instead, batches of operations were stored, and upon a branch level change, were added together through batch_add_text. The use of list comprehension to generate large numbers of label widgets in batch_add_text was much faster, and generation time of the same IR was reduced to under a minute. This generation time was explored in section 7.4. The remaining factors affecting the time of generation are due to the implementation of ipywidgets, and would be difficult to mitigate in the MLVIR system, if at all possible. An issue with the Loadbar arose, as the Loadbar updates with each GenerateMLVIR iteration. But, as the ipywidget generation in the batch_add_text function was consuming the most time, the Loadbar would stop updating while this loaded. This may be frustrating to users, and does not effectively represent the loading progress.

Currently the operations dictating changes in branch level, ModuleOp, IfOp, ForOp, YieldOp and ReturnOp, are hard coded into the system in the MLVIRPattern class, which may lead to extensibility issues. They are dialect specific, and these dialects are currently evolving and changing with xDSL. For instance, the builtin.func function

operation becoming the func.func operation in the func dialect only just occurred in commit $d31a413cb321a447a0743ed6871d4de8ee449bf2$, on $24/04/2022$.

Instead, a better mechanism more aligned with the extensibility of MLIR would be to make a check for enclosed blocks and regions within operations, branching from there.

The branch_level variable in the NTS was not always accurate upon testing, implying that it was not incremented correctly, leading to issues with adding Tabs and Labels. As a result, the line_level dictionary from the GenerateMLVIR class was used and passed in as an optional variable, that when specified was used instead of branch_level. This should be removed in future implementations, and the branch_level variable incrementation should be fixed.

In the RewriteMLVIR class there is some function redundancy. The reInit and restart functions could have been merged with the __init__ class, taking optional arguments to determine the relevant widgets to close, and functionality to execute. Another issue resides in the RewriteMLVIR class in the rewrite functions. The function GetRewriteOp is called from the MLVIRGeneration class with a location of an operations to get, which iterates the IR until the location matches the line number, and returns the operation. This function is called iteratively for each location in a list of locations, causing many IR iterations. This presents, for large IRs, the issue that the IR must be iteratively partially traversed many times, when one full iteration would be able to fetch all of the operations instead.

The tool could also benefit from some further error handling. Currently error handling checks for empty inputs and non-integer N value input in the Init Stage. A check is also made to ensure that a Tab is added before Labels, avoiding errors. The implementation of the NTS likely provides the greatest risk of error throwing due to the excessive nesting. The class relies on the linear addition of widgets through the NTS creation, and changing of the implementation would throw errors, especially in the get_checkbox_vals class. Error handling is of importance to the tool as, if caught, the error can be relayed to the user allowing them to modify their choice. Otherwise, the error will stop the tool, requiring the user to restart the tool to mitigate the error, and be provided with a more complex error message.

For the code implementation itself, a combination of PascalCase, camelCase, and snake_case are used. Class names consistently use PascalCase, but all function and variable names are a combination of the three. For consistency, one of the three cases should be selected, and the function and variable names should be changed to match the chosen case. This should have been done at the start of the project implementation to ensure consistency throughout.

7.4 MLVIR Generation Time

One of the biggest issues with the visualisation is the generation time. Generation time is of extreme importance to user experience, with long waiting times leading to frustration. MLVIR has a reasonable generation time of 10 seconds for up to approximately 2000 operations, but going beyond this can lead to leaps in generation time.

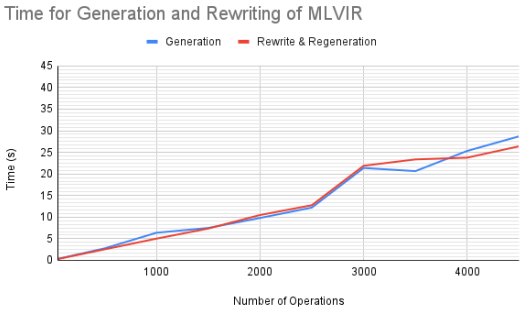

Figure 7.1: Time for Generation and Rewriting of MLVIR

Experimentation was carried out to probe the generation and regeneration time of MLVIR. Using the tool provided by student Anh Nguyen, modules of varying size were generated. These modules consist of a ModuleOp, followed by a series of randomly generated arith operations, including arith.constant, arith.sub, and arith.add. IRs of sizes 50, 500, 1000, 1500, 2000, 2500, 3000, 3500 and 4000 were input into the MLVIR tool, and the time from beginning to completion of the MLVIR generation was measured by using the time library in Python. Then, rewriting and regeneration in the experimentation is defined as selecting the arith.constant op to be matched, and then applying a rewrite of another arith.constant with value one to every identified rewrite location. The time for this was also measured using the time library. These results were noted, compiled and used to generate the graph seen in figure 7.1.

The resultant graph allows for the following observations. If the number of operations in the IR range from 0-2500, the generation and regeneration time is reasonable, maxing out at around 15 seconds. However, over 2500 operations, the generation times increase quickly, and reach generation times of almost 30 seconds, becoming increasingly frustrating for the user. A clear proportional relationship between number of operations and generation time is shown, implying that beyond 4000 operations the generation time will continue increasing.

It can be seen that below 1500 operations, regeneration actually occurs quicker than generation, despite rewriting, and generating the same number of operations. During regeneration, the original MLVIR is closed. However, this may remain in memory within the kernel, explaining the shorter regeneration time. Between 1500 and 3000 operations generation and regeneration times are similar as expected. At 3500 operations, regeneration takes longer than generation. This makes sense as a large number of operations are required to be rewritten, on top of the regeneration. However, at 4000 operations, initial generation takes longer. This is unexpected, and could again be explained by existing widgets remaining in the kernel after closure. During experimentation, in some cases the closing of the initially generated MLVIR did not work, resulting in multiple MLVIRs. This bug could also have contributed to the unexpected data points. Note the bug was later resolved and was caused by NTS objects not being closed at certain stages.

Chapter 8

Conclusions

The goal for this thesis was to create a first iteration of the Multi-Level Intermediate Visual Representation (MLVIR) tool, allowing users to explore IRs and pattern rewrites on IRs. With the movement and development of IR away from representations only understood by compilers to human-readable, higher-level IR, such a tool would further improve ease of understanding, provide effective visualisation, and allow analysis and development of such IRs. The MLVIR tool is implemented in its second iteration, and currently allows IR input to be parsed and represented as seen in figures 5.10 and 5.11 as MLVIRs. Operations can be selected to be matched, with matches represented by Checkboxes next to the operations, and rewritten, allowing for the MLVIR to be updated with the new operations. The MLVIR tool complies with the criteria set out in section 1.2.1, improving readability, context retention, allowing IR exploration, region visualisation, compression of large IRs, and pattern matching and rewriting. The second iteration also improves on the first iteration, implementing identified weaknesses; a new IR representation (MLVIR) was developed with region visualisation, compression was applied, pattern match visualisation was handled effectively with context retention of match locations, and patterns could be input for rewrite application.

8.1 Suggestions for Future Work

The most pressing issue to be fixed in future work is the operation printing. Currently operation printing creates labels of just operation types, with the exception of func.func and arith.constant operations (see such operations in figure 5.4). This loses a huge amount of information, and is important to fix in the next implementation. This could be fixed one of two ways: a printing system could be made from scratch, or the xDSL Printer class could be taken and modified to, rather than print to output, create a string, which would then be passed to as the value to Label widgets, producing full output.

In additional future work, the MLVIR tool should be distributed in a format other than the Jupyter Notebook currently containing it. This implementation was effective and useful for the implementation of the tool, but the resultant tool does not feel like a complete, distributed tool. This could be mitigated by releasing the MLVIR tool through distribution tools such as Jupyter Voilà (QuantStack, 2019), allowing the notebook to

be ran from the terminal, or Jupyter Lite (Tuloup, 2021), allowing a fully static distribution of the tool.

Pattern matching could also be improved, allowing more effective choosing of patterns to be matched, as well as multiple operations and patterns to be specified. Regions and blocks should also be allowed to be matched in future distributions, further improving the pattern match and rewrite system. These rewrites should be able to be carried out live in future releases; rather than deleting and creating a new MLVIR with each modification, one single MLVIR should be kept, with the IR tracked in the NTS class, and updated as changes are made. This would cause less waiting time for users as the tool would not be regenerated upon stage switching and pattern rewrites.

Small GUI changes should also be implemented as discussed in section 7.3: the entry for *N* should use a Text widget instead of a Textarea widget, the ”N:” label location should be fixed, Accordions should contain some sort of label to improve context retention, SelectMultiple for operation selection should be replaced with a Dropdown widget, and a FileUpload widget should be added to allow users to upload .xdsl files for use in the tool.

Finally, the branch_level variable in the NTS class should be fixed, removing the need for the alt_level variable as discussed in section 7.3, redundancy in the RewriteMLVIR class should be removed by the combination of the reInit and restart function with the __init__ function, the rewrite functions in RewriteMLVIR should be modified to perform one IR iteration instead of many, a class and function naming format should be decided on, and a large amount of testing should be carried out to ensure the tool works as desired.

Bibliography

Altair (2022). *Overview*. URL: https://altair-viz.github.io/getting_started/overview.html.

– (n.d.). *Simple Scatter Plot with Tooltips*. URL: https://altair-viz.github.io/gallery/scatter_tooltips.html.

Bokeh (2022a). *First steps*. URL: https://docs.bokeh.org/en/latest/docs/first_steps.html.

– (2022b). *First steps 9: Using widgets*. URL: https://docs.bokeh.org/en/latest/docs/first_steps/first_steps_9.html.

bqplot (n.d.). *bqplot*. URL: https://github.com/bqplot/bqplot.

Breddels, Maarten (2017). *Ipyvolume*. URL: https://ipyvolume.readthedocs.io/en/latest/.

Chris Lattner, Vikram Adve (2004). "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation". In:

Darth-Revan (2022). *language-llvm-ir package*. URL: https://github.com/Darth-Revan/language-llvm-ir.

Godbolt, Matt (n.d.). *Compiler Explorer*. URL: https://godbolt.org/.

GraphViz (2021). *GraphViz*. URL: https://graphviz.org/.

Gün, Erol Mesut (2019). *Python-Data Projects — Data Analysis UI Reinforced by Ipywidgets*. URL: https://medium.com/analytics-vidhya/python-data-projects-data-analysis-ui-reinforced-by-ipywidgets-d680493464b8.

Hjort, Rikard, Jakob Holmgren, and Christian Persson (2017). "The CakeML Compiler Explorer Visualizing how a verified compiler transforms expressions". In:

hyperspy (2021). *hyperspy_gui_ipywidgets*. URL: https://github.com/hyperspy/hyperspy_gui_ipywidgets.

Jana, Suman (n.d.). *Basic Program Analysis*. URL: https://www.cs.columbia.edu/~suman/secure_sw_devel/Basic_Program_Analysis_CF.pdf.

Jekyll, Beautiful (2021). *xDSL*. URL: https://xdsl.dev/ (visited on 11/03/2022).

Jupyter (2022). *Jupyter*. URL: https://jupyter.org/.

Jupyter, Project (2022). *ipywidgets*. URL: https://ipywidgets.readthedocs.io/en/latest/.

LLVM (n.d.[a]). *LLVM's Analysis and Transform Passes*. URL: https://llvm.org/docs/Passes.html.

– (n.d.[b]). *opt - LLVM optimizer*. URL: https://llvm.org/docs/CommandGuide/opt.html.

llvm (2022). *vs-code mlir*. URL: https://github.com/llvm/vscode-mlir.

Matplotlib (2022). *Plot types*. URL: https://matplotlib.org/3.5.1/plot_types/index.html.

matplotlib (n.d.). *Scatter Demo2*. URL: https://matplotlib.org/stable/gallery/lines_bars_and_markers/scatter_demo2.html#sphx-glr-gallery-lines-bars-and-markers-scatter-demo2-py.

Mehta, Freyam (2021). *[MLIR] MLIR Visualization Project*. URL: https://discourse.llvm.org/t/mlir-mlir-visualization-project/4280.

MLIR (2021). *Operation Canonicalization*. URL: https://mlir.llvm.org/docs/Canonicalization/ (visited on 11/03/2022).

– (n.d.[a]). *'pdl' Dialect*. URL: https://mlir.llvm.org/docs/Dialects/PDLOps/.

– (n.d.[b]). *MLIR Language Reference*. URL: https://mlir.llvm.org/docs/LangRef/.

– (n.d.[c]). *Table-driven Declarative Rewrite Rule (DRR)*. URL: https://mlir.llvm.org/docs/DeclarativeRewrites/.

Perkel, Jeffrey M. (2018). *Why Jupyter is data scientists' computational notebook of choice*. URL: https://www.nature.com/articles/d41586-018-07196-1.

Plotly (2022). *Getting Started with Plotly in Python*. URL: https://plotly.com/python/getting-started/#JupyterLab-Support-(Python-3.5+).

– (n.d.). *Plotly Express in Python*. URL: https://plotly.com/python/plotly-express/.

Python (2022). *python*. URL: https://www.python.org/.

QuantStack (2019). *And Voilà!* URL: https://blog.jupyter.org/and-voil%5C%C3%5C%A0-f6a2c08a4a93.

Regehr, John (2018). *How LLVM Optimizes a Function*. URL: https://blog.regehr.org/archives/1603 (visited on 12/03/2022).

seaborn (n.d.). *Scatterplot with varying point sizes and hues*. URL: https://seaborn.pydata.org/examples/scatter_bubbles.html.

Telenczuk, Bartosz (2019). *Introducing templates for Jupyter widgets layouts*. URL: https://blog.jupyter.org/introducing-templates-for-jupyter-widget-layouts-f72bcb35a662.

Toal, Ray (2022a). *Intermediate Representations*. URL: https://cs.lmu.edu/~ray/notes/ir/.

– (2022b). *LLVM Language Reference Manual*. URL: https://llvm.org/docs/LangRef.html.

– (2022c). *The LLVM Compiler Infrastructure*. URL: https://llvm.org/.

– (2022d). *Tuple-Based Intermediate Representations*. URL: https://cs.lmu.edu/~ray/notes/squid/.

Tuloup, Jeremy (2021). *JupyterLite: Jupyter WebAssembly Python*. URL: https://blog.jupyter.org/jupyterlite-jupyter-%5C%EF%5C%B8%5C%8F-webassembly-%5C%EF%5C%B8%5C%8F-python-f6e2e41ab3fa.

Vega (2022). *Vega – A Visualization Grammar*. URL: https://vega.github.io/vega/.

Walker, David (n.d.[a]). *Intermediate Representation*. URL: https://www.cs.princeton.edu/courses/archive/spring03/cs320/notes/IR-trans1.pdf.

Walker, David (n.d.[b]). *Summary of Optimization Material*. URL: https://www.cs.princeton.edu/courses/archive/spr03/cs320/notes/ssa.pdf.

Waskom, Michael (2022). *Overview of seaborn plotting functions*. URL: https://seaborn.pydata.org/tutorial/function_overview.html.

Woodruff, William (2020). *Understanding static single assignment forms*. URL: https://blog.yossarian.net/2020/10/23/Understanding-static-single-assignment-forms (visited on 11/03/2022).

xdsl (n.d.[a]). *pattern_rewriter.py*. URL: https://github.com/xdslproject/xdsl/blob/main/src/xdsl/pattern_rewriter.py.

– (n.d.[b]). *printer.py*. URL: https://github.com/xdslproject/xdsl/blob/main/src/xdsl/printer.py.

xdslproject (2021). *affine_ops.xdsl*. URL: https://github.com/xdslproject/xdsl/blob/main/tests/filecheck/affine_ops.xdsl (visited on 12/03/2022).