Universiteit Leiden

Master Computer Science

Parallel Code Generation on the GPU

Name:	Marcel Huijben
Student ID:	s1780107
Date:	02/08/2021
Specialisation:	Advanced Computing & Systems
1st supervisor:	Dr. K. F. D. Rietveld
2nd supervisor:	Dr. A. Uta

Master's Thesis in Computer Science

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Traditional compilers are single-threaded programs running on CPUs that perform a sequential set of stages in order to transform the source code of a program into a machine representation. Recently, GPUs have become increasingly powerful and programmable and could be used to speed up modern compilers. By using only GPU primitives which can be efficiently executed on modern GPUs, we have designed and implemented a compiler that can be run on the GPU. This thesis describes the backend of the Pareas compiler, which transforms the abstract syntax tree generated by the frontend into machine code for the RISC-V architecture using a novel set of compilation stages. The performance of our implementation was characterized under various types of inputs. The experimental results show that the implementation scales well for wide abstract syntax trees reminiscent of real-world source files. However, for long functions the register allocation stage imposes a significant overhead.

Contents

1. Introduction

Compilers have been a vital part of software development for the past decades. Compilers are responsible for taking a program represented using source code written in a programming language and transforming it into executable code. For large source files, this process may take a significant amount of time using a traditional compiler. A more efficient compiler could significantly reduce the overhead involved when compiling large source files.

Traditional compilers are single-threaded processes going through a set of sequential stages to transform a source file into executable code. To speed up the compilation process, the traditional approach is to run multiple instances of the compiler in parallel to compile separate source files concurrently. However, modern computers are equipped with powerful hardware capable of running a large number of operations in parallel. By using the GPUs available in modern computers, a compiler can be designed which can achieve parallelism within a source file. This can allow us to speed up the compilation process even if only a single file is compiled.

The stages performed by a traditional compiler are however not well-suited for the GPU units capable of running the same operation for a large number of elements concurrently. If we want to make a compiler capable of using modern GPUs in order to speed up the compilation process, a redesign of the stages of a traditional compiler is required.

In this thesis, we propose an implementation of the back-end of parallel compiler designed to run on GPU architectures. The backend of a compiler is responsible for taking the abstract syntax tree returned by a parsing the source file and transforming it into executable code. The implementation described in this thesis is part of a joint project for creating a full parallel compiler. The frontend of the compiler, responsible for parsing and performing type analysis of the input source file, is described in the thesis ”Parallel Lexing, Parsing and Semantic Analysis on the GPU” [Voe21].

1.1. Goals

The primary focus of this thesis is on designing and implementing a compiler which fully runs on a GPU. To achieve this, we designed a new set of stages to transform the output of the compiler front-end into executable code. After this, an implementation of this design was written. The implementation had to be capable of taking an abstract syntax tree representing an imperative language similar to the C programming language and transform it into a set of instructions for the RISC-V platform. The RISC-V architecture was chosen due to its consistent fixed-width instruction encoding.

1.2. Contributions

This thesis contributes a novel approach of code generation to the field of compiler design within computer science. The major contributions of this thesis are the following:

- A design for the backend of a compiler running completely on a GPU, designed to make use of the parallel primitives available on such architectures to efficiently parallelize the compilation process.

- An implementation of the proposed design, capable of taking a program represented as an abstract syntax tree and transforming it into RISC-V instructions. This open-source implementation is available at `https://github.com/Snektron/pareas.git`. This repository contains all our code, data and experiments.

- A characterization of the performance of our implementation, detailing the performance for various types of input.

In this thesis, Chapter 2 provides an overview of the existing compiler principles and the GPU primitives our design uses to implement the compilation process. Chapter 3 provides an outline of our design, describing the various stages of our proposed compiler architecture. Chapter 4 details the implementation of our described design and contains information on the exact process used to implement the newly designed stages. In Chapter 5 we use a set of benchmarks to determine the performance of our implementation by measuring the runtime of our compiler for various inputs in order to characterize the performance of our implementation. Finally, in Chapter 6 we draw our conclusions based on the performed experiments and propose various additions to our proposed design that could be researched in the future.

2. Background

The proposed design in this thesis builds on previous work in the fields of compiler design and GPU programming. This chapter describes the relevant work and concepts which will be referred to throughout this thesis.

2.1. Compilers

Modern computer programs are written in programming languages. These languages give programs a textual representation, allowing programmers to write programs much more effectively. However computers cannot directly execute programs written in a programming language and a translation needs to be performed before the program can be executed. Compilers are programs responsible for taking a program source code, represented in a programming language and transforming it into machine code, allowing the program to be executed.

To achieve this, traditional compilers go through a set of sequential stages to transform the input source code into machine code. Most compilers go through a similar set of stages, represented in Figure 2.1. These stages are: lexical analysis, parsing, semantic analysis, intermediate code generation, machine-independent code optimization, code generation and machine-dependent code optimization [Aho+06]. Each of these stages transforms the output of the previous stage into a new representation, until finally machine code is produced. Traditional compilers execute these stages on a single CPU processor using only a single thread.

2.2. GPU

Graphics Processing Units, or GPUs, were originally designed as specialized hardware used to assist in the rendering of images. Traditionally this was done by using fixed functions which could be applied to large buffers to perform common subroutines efficiently. Due to the nature of graphics data, often consisting of a set of triangles or pixels which could be computed independently, processing this data in parallel was a useful method of speeding up these operations. Later, the operations required to render certain applications became more and more complex and the fixed function approach was no longer viable. As a consequence GPUs became steadily more programmable. Modern GPUs are capable of running a wide variety of programs and are able to run applications that do not operate on graphics data at all.

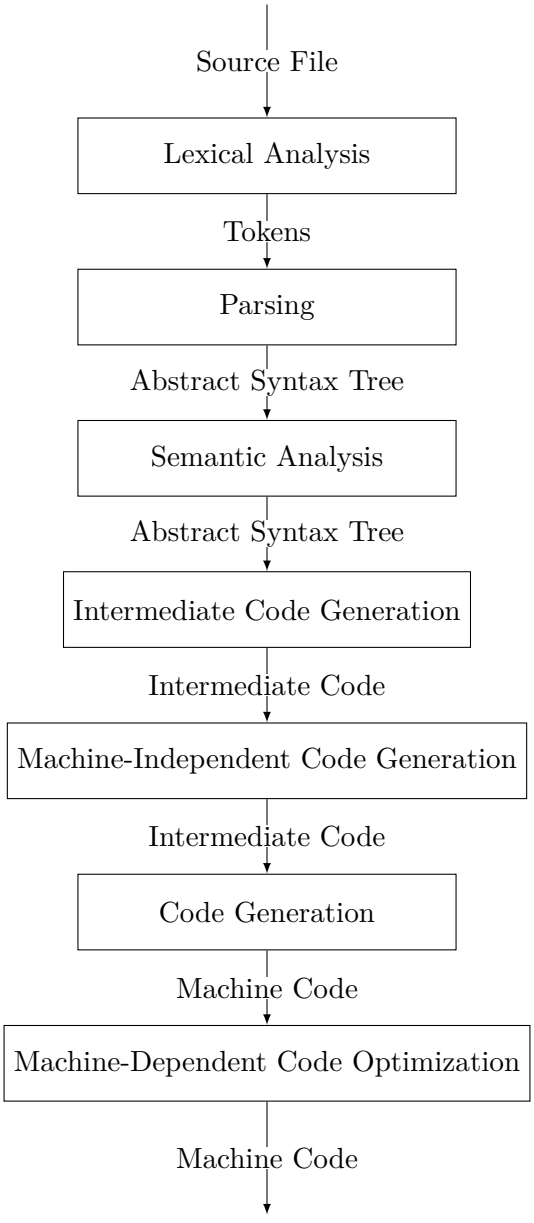


Figure 2.1. A schematic overview of the stages and the flow of data structures within a traditional compiler

GPUs are optimized to run similar operations on large amounts of data in parallel. To achieve this, GPUs are internally a collection of vector processors, which are processors capable of performing the same operation on a large set of data at the same time. Each vector processor has to execute the same operation for all data elements currently being processed by that processor. This allows for efficient parallel execution if all data elements require the same operation, however if data elements within the vector require separate operations, such as through a branch in the program, the GPU will have to execute both paths for all data elements, which can be inefficient if these paths become large.

2.2.1. GPU Primitives

To make effective use of GPU architectures, a number of useful primitives are commonly made available to programmers. These primitives generally take a large input buffer and perform some operation that can be efficiently run in parallel on a GPU architecture.

Scans

The scan operation is a GPU primitive used to efficiently perform a number of calculations efficiently. The scan operation is defined as an operation taking an associative operator and an array of data. Given the associative operator Δ and an array of data A of length n, the output of a scan operation is defined as an array B of length n, where the value of the element at index i is given using Equation 2.1. Thus a scan results in an array where each index in the resulting array represents the results of the operator applied to any element up to and including the current index.

$$B(i) = \Delta_1^i A(i) \tag{2.1}$$

A variation of the scan operation, called the exclusive scan, exists. This variation does not include the current index in the result. In this case, the operator is defined as is seen in Equation 2.2.

$$B(i) = \Delta_1^{i-1} A(i) \tag{2.2}$$

A specific application of the scan operation is the implementation of prefix sums, which can be implemented by using the addition operator as the associative operator.

A modified version of the scan operation, called the segmented scan, can be used to perform multiple scans within one array of data. The segmented scan takes an additional mask, which describes where a subsection of the total array starts. At the start of a subsection, the result is reset to the only include the current index, thus disregarding any input that came before it. A segmented scan can be defined using the recursive function given in Equation 2.3, where $M(i)$ describes the segment mask, which indicates whether a new

segment starts at index i. Segmented scans also have an exclusive version which operates similarly to an exclusive scan.

$$B(i) = \begin{cases} B(i-1) \Delta A(i) & !M(i) \\ A(i) & M(i) \end{cases} \tag{2.3}$$

2.3. Futhark

Our implementation was written in the programming language Futhark [Hen+17]. Futhark is a functional programming language designed to compile into parallel code. The compiler can target either multicore CPU targets, or GPU targets by compiling to either CUDA or OpenCL code.

2.3.1. Primitives

Futhark itself contains a number of useful parallel primitives that can be used to efficiently implement larger programs. The following sections list the primitives used by our implementation.

Map

The first useful primitive is the map primitive, which applies a function to every element of an array and returns an array containing the result of applying this function to each element. This operation allows us to run the applied function in parallel for each element of an array, making it a useful primitive for parallel execution. Variations of this primitive exist that take multiple arrays of equal size and a function taking multiple parameters, which allows for the parallel processing of multiple arrays.

Scan

Futhark has a scan primitive, this primitive corresponds to the inclusive scan primitive described in Section 2.2.1.

Scatter

The language contains a scatter primitive. A scatter operation takes three arrays as parameters: a destination array which will receive the elements to be scattered, a source array containing the elements to be written to the destination array and an index array containing the indices in the destination to write each element in the source array to.

Futhark itself adds one additional feature to the scatter primitive. Futhark performs a bounds check on each index in the index array and if this index is not a valid index in the destination array, the element in the source array corresponding to this index will be ignored.

2.4. Related Work

Despite the increase in popularity of GPU architectures in the recent years and a trend to make more applications capable of using these architectures to significantly increase their performance by running parts of their implementation in parallel, compilers tend to still be single-threaded CPU processes. Therefore, little work into the possibility of parallel code generation on GPUs has been done.

In the past, work on creating a parallel compiler that can run on the GPU has been performed by A. Hsu[Hsu19]. In this paper, a number of algorithms for parallel lexical analysis and parsing are proposed, however no code generation is performed. To create a full compiler capable of running on the GPU, the later stages have to be implemented.

Further work on running certain stages of a compiler in parallel has been performed by the parallel GCC project [GNU]. This project aims to run certain optimization and analysis stages of the GCC compiler in parallel on the CPU. A number of analysis stages are run in parallel by using multiple CPU threads in order to achieve parallelism. However, this still performs the traditional compiler pipeline and does not exploit the capabilities of modern GPU hardware in order to achieve a more significant speedup.

An implementation for running Lisp on GPUs using CUDA, named CuLi [Süß+18], is capable of executing a runtime environment for the Lisp programming language on GPUs by recursively evaluating the abstract syntax tree. Only the evaluation of the program is performed on the GPU and all other stages, such as parsing the source code, are performed on the CPU. Furthermore, this implementation only supports explicit parallelism, requiring a specific function to be used within the source code to mark branches that can be run in parallel. Our implementation can instead analyze nodes in the abstract syntax tree in parallel implicitly. Our implementation also supports code generation instead of a direct evaluation on the GPU.

Our implementation was written in Futhark, a pure functional programming language. In pure functional programming languages, no expression may have side-effects. This property makes them suitable for writing parallel programs, as expressions can theoretically be run in parallel. Previous work has been done on writing compilers in functional programming languages [KD13] [Ghu06]. However, these implementations make heavy use of language constructs and data structures that perform poorly on GPU architectures. Examples of common constructs include conditional branches, which leads to branch divergence on GPU architectures, or the use of list data structures into which elements are inserted repeatedly, which do not perform as well as data structures optimized for use on the GPU. Our implementation was written with GPU architectures in mind and makes use of primitives and data structures that can be more efficiently executed on these architectures.

3. Design

The Pareas back end is responsible for transforming the abstract syntax tree received from the front end into RISC-V instructions. The RISC-V architecture was chosen for two reasons. First of all, all instructions on this platform have the same size, easing the compilation process since variable-length instructions do not need to be accounted for, allowing us to treat entire instructions as one unit of data. Secondly, the encoding of the instructions itself is regular, with source and destination registers appearing at the same location within the instruction whenever they are used and the opcode encoding itself being composed of a category and an actual instruction opcode within that category, which makes identifying classes of instructions easier. The regularity of the instruction encoding allows us to more easily calculate the individual components of an instruction separately, then combine them in a later stage with the instruction opcode by using bitwise operations to shift the bits to the correct, fixed position in the instruction. These two factors make the RISC-V architecture a suitable target for our compiler.

To achieve the compilation into RISC-V instructions, the back end goes through multiple stages, each with a specific function. Each stage transforms the output of the previous stage into a new representation. The initial stage takes the abstract syntax tree as input and each subsequent stage performs transformations until the final stage outputs an array of instructions and a table describing where each function starts and ends. This chapter describes the design of each stage in the order they are executed by the compiler.

A schematic overview of the stages of the compiler and the data flow between stages can be found in Figure 3.1. Each node in this graph represents one of the major stages discussed in this chapter. The directed edges between these nodes are labeled with the name of one of the major data structures present during the compilation process. The direction of these edges represent the data flow within the program. If multiple stages receive the same data structure without modifying it, this is represented by having the edge go to the multiple target stages while only having one source stage.

The back-end expects the abstract syntax tree to adhere to a particular form. The layout of this abstract syntax tree was chosen specifically for use on a GPU. For instance, instead of using a traditional representation where each parent node would contain a variable-length list of pointers to child nodes, we use a representation where each child contains only a single pointer to a parent node and a field describing the depth of the node in the tree to allow for fast bottom-up processing of the syntax tree, a design previously introduced by A. Hsu[Hsu19]. Furthermore, some additional ancillary nodes have been added to reduce the number of operations performed by certain stages, such as introducing a while loop ancillary node which can be used to quickly locate the location of the first instruction of the while loop comparison without having to search the abstract syntax tree. An overview of the various node types in the abstract syntax tree and their function can be found in Appendix A.

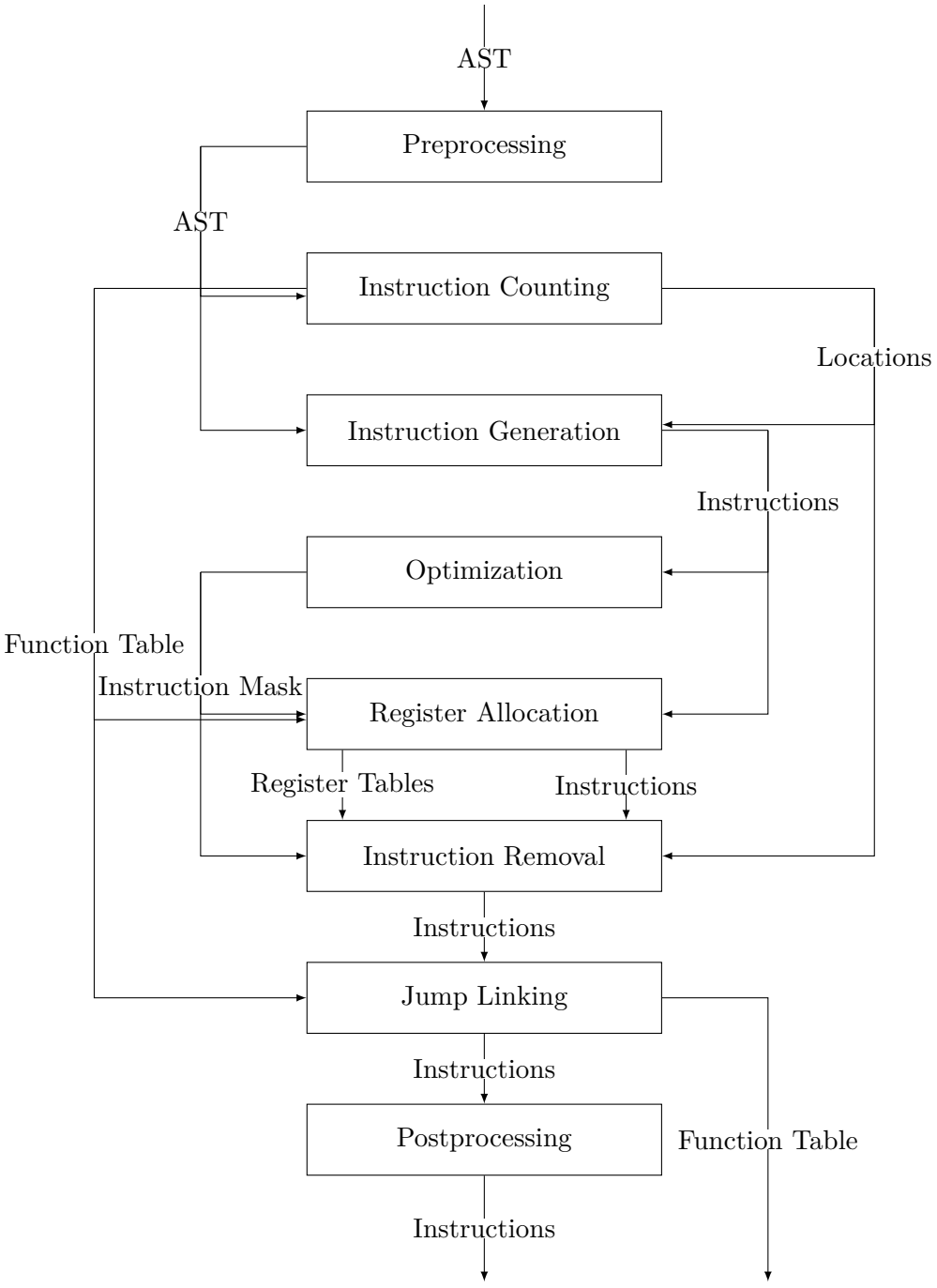


Figure 3.1. A schematic overview of the stages and the flow of data structures within the compiler

3.1. Preprocessing

The preprocessing stage of the back end is designed to perform minor transformations to the abstract syntax tree to ease the compilation process in the later stages. In particular, for two specific cases the node types of certain nodes in the abstract syntax tree are replaced such that the instructions generated for these nodes by later stages is unambiguous. This stage is sub-divided into two stages: function argument node preprocessing and comparison operator preprocessing. The preprocessing stage itself does not insert or remove any nodes from the syntax tree, it simply modifies information of existing nodes. A schematic overview of the sub-stages can be found in Figure 3.2.

3.1.1. Function Argument Node Preprocessing

The first sub-stage of preprocessing is function argument node preprocessing. On most platforms, including the RISC-V architecture, function arguments are passed according to some standardized calling convention. This calling convention describes how a function can find its arguments. RISC-V, like many architectures, passes a number of arguments via registers and only passes arguments on the stack if no registers are available to pass arguments in. It also differentiates between integer and floating-point arguments, as these are passed in different registers. Furthermore, RISC-V allows for passing floating-point arguments in integer registers if no floating-point registers are available, but integer registers are still free.

Later stages of the compiler expect a combination of node type and node data type to uniquely identify what sequence of instructions would be required to compile a node in the abstract syntax tree. However, to comply with a calling convention, the compiler will need to generate different sequences of instructions depending on the index of the argument the current node represents. The function argument node preprocessing stage ensures that the requirement of the later stages can still be met while also supporting realistic calling conventions. The stage achieves this by modifying the node type and node data of any function argument or function call argument nodes.

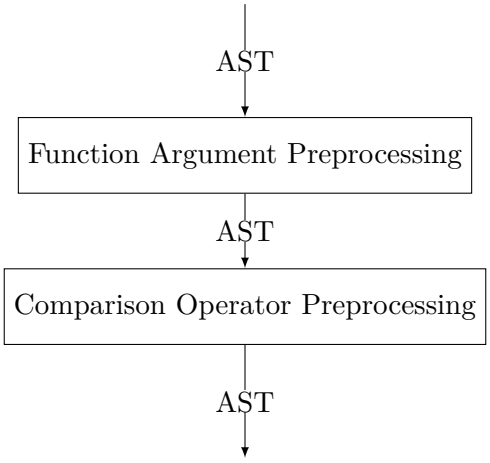


Figure 3.2. A schematic overview of the sub-stages of the preprocessing stage

The function argument node preprocessing stage will replace any function argument nodes or function call argument nodes in the abstract syntax tree by one of three possible node types:

- Function (call) argument node
- Function (call) argument node: floating-point value in integer register
- Function (call) argument node: value on stack

Each of these node types represents a different way an argument can be passed to a function according to the RISC-V calling convention. The default argument node type is used when the argument would be passed in its corresponding register type, integer arguments in integer registers or floating-point arguments in floating-point registers and will have its data type set to either integer or float to represent which of these two register types should be used. The floating-point value in integer register node type represents the scenario where a floating-point argument is passed via one of the integer registers, the data type of these types of nodes will always be set to float, although this data type is ignored by later stages of the compiler, as the node can only represent a single type of conversion. Finally, the value on stack nodes are generated when the function argument represented by the current node is passed on the stack, the data type will be set to either integer or float depending on the datatype of the passed argument.

The function argument node processing stage also sets a node's node data field to represent either the register number or stack offset that should be used for this argument. This is handled during preprocessing as this information is already computed when determining what type of node to return for each of the given arguments. For the default argument node type and for the floating-point value in integer register node type, this field is set to an integer n where a value of n represents that the n-th integer or floating-point argument register should be used to pass the value of this argument, whereas for value on stack nodes the value is set to the stack offset relative to the stack top after allocating space to store the stack arguments where the argument should be stored during a call.

3.1.2. Comparison Operator Preprocessing

The second preprocessing stage is comparison operator preprocessing. Comparison operators always return a boolean value, which is represented using the integer data type in the compiler, regardless of the data types of the arguments to this comparison operator. However, as the instruction count and instruction generation stages only differentiate nodes based on node type and return value data type, this would make it impossible to differentiate between comparison operators operating on integer arguments and comparison operators operating on floating-point arguments. But since these operations do require different instructions, a preprocessing stage designed to split the two options ahead of time was required.

The comparison operator preprocessing stage simply replaces the data type field of any comparison node depending on the data types of its children. If its children are floating-point values, this stage will set the resulting data type field of the comparison node to be float, whereas it will be set to integer otherwise. This allows the later stages to differentiate between comparisons on integer data types and comparisons on floating-point data types.

It is important to note however, that the actual operations still return boolean, thus integral, values. However, since the abstract syntax tree is already fully type-checked at this point, any parent node of a comparison operator should still expect the operation to return an integer value. Furthermore, since the resulting data type field is not used to allocate the registers themselves, setting it to a different type from what is actually returned by a node at this point does not result in any issues.

3.2. Instruction Counting

The next major stage in the compilation pipeline is the instruction counting stage. As the name implies, the main purpose of this stage is to calculate how many instructions each of the different nodes in the tree will require and use this information to compute a mapping of nodes to instruction locations. The instruction generation stage can then use these calculated locations to place the generated instructions at the correct locations in the final code, allowing us to process nodes independently. On a GPU, this means that nodes can be compiled in parallel. The instruction counting stage is also responsible for computing the function offset table, which is a table that specifies the starting instruction location and size of each function.

The instruction counting stage itself is sub-divided into three sub-stages. First, an initial location mapping is computed which provides a base overview where each node will end up. The next stage then takes this mapping and corrects the instruction offset of any node for which the initial stage does not yield a correct instruction location. Finally, the final stage takes the generated instruction locations and computes the function offset table based on these locations, completing the stage. An overview of the sub-components of this stage can be found in Figure 3.3.

3.2.1. Initial location mapping

To compute the instruction locations we make use of some notable trends in most programming languages. For the vast majority of node types, child nodes are computed, then the result is used by the parent node which performs some operation on the values returned by the child nodes. This holds for nearly all expression nodes, any list-type node and a majority of statement-type nodes. When translating this to instructions, this means that the instructions for the child nodes are located before the instructions of the parent node.

Furthermore, the abstract syntax tree is currently encoded in a post-order representation, meaning that any child nodes precede the parent node. This corresponds to the order in which instructions will be generated for most node types. This allows us to efficiently calculate instruction locations in parallel for any node type conforming to this ordering.

To perform the initial location mapping, we map the node to the number of instructions required to encode it. This mapping can be performed concurrently for any node in the tree, as this mapping has no dependency on the results of any other node type in the tree.

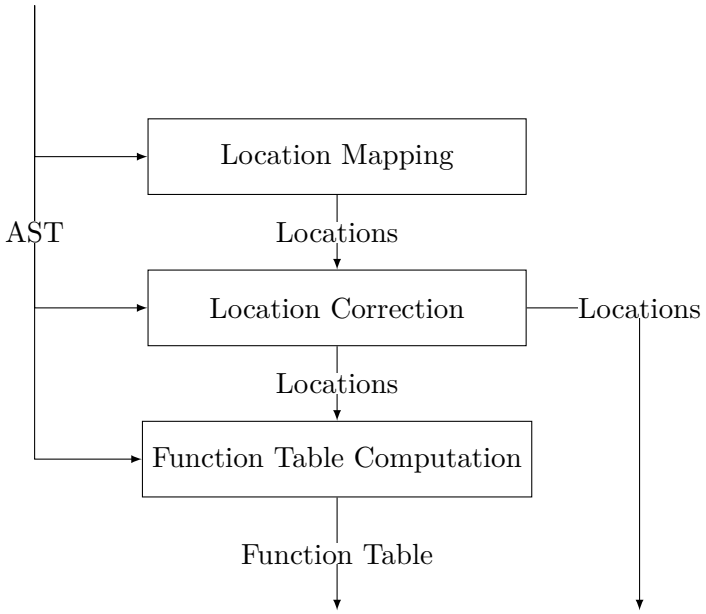


Figure 3.3. A schematic overview of the sub-stages of the instruction counting stage

After completing this mapping, we can get the initial instruction mapping by performing an exclusive prefix sum over this newly generated size mapping, which yields a mapping of each node to an instruction location. The complete computation is given by Equation 3.1. In this equation, $I(n)$ represents the instruction location for the n-th node and $S(n)$ represents the number of instructions required to hold the n-th node.

$$I(n) = \sum_{i=0}^{n-1} S(i) \tag{3.1}$$

The size function itself is further subdivided in three parts, as shown in Equation 3.2. The size is defined as the sum of the base size of the node, $S_b(n)$, which contains the number of instructions required to encode the node and two factors design to insert some padding into the instruction offsets. $A(n)$, used to insert padding for function arguments and $C(n)$, which is used to insert padding for jumps in conditionals. The padding functions are described in detail in the next subsection.

$$S(n) = S_b(n) + A(n) + C(n) \tag{3.2}$$

3.2.2. Location mapping correction

After the initial mapping has been computed, small corrections will need to be performed for nodes not conforming to the assumption that its instructions would be placed after the instructions of its children. For the syntax tree used by the Pareas compiler, there are two categories of nodes that require post-processing: branch nodes and function call argument nodes.

For nodes representing branches, such as if-conditions or while-loops, a conditional jump instruction has to be placed between the child node representing the condition and the child node representing the code to be executed depending on the condition. For nodes representing function call arguments, the instructions setting up all arguments must be placed immediately before the call instruction represented by the function call itself, whereas using the prefix-sum method employed by the earlier stage, the instructions for the children of these nodes will be placed in between the various argument nodes.

To insert these instructions, space to hold these instructions will first have to be made. This is done by introducing padding functions into the size function used in the initial location mapping stage, as seen in Equation 3.2. Two types of padding functions are used. First of all, $A(n)$ represents the padding required to hold function call arguments. Since these need to be placed directly before the call, we increase the padding of the function call argument list node, which is always the direct child of any function call expression node and would thus always be located immediately before the call itself. Increasing the size of this node would therefore increase the location of the function call expression node, thus allocating some space immediately before the call itself. $A(n)$ is thus set to the number of instructions required to set up the arguments immediately before the call if and only if n is an function call argument list node and to 0 otherwise.

To allocate space for the conditional jumps, the $C(n)$ part of the equation is used. To allocate some space between the conditional and the statement we set the value of $C(n)$ to the number of instructions required to perform the conditional jump if and only if n is the node representing the conditional. We can check that a node is the conditional by simply checking if its parent is one of the conditional nodes and its child index is set to the index representing the conditional for the given parent type. The conditionals for if statement nodes and if-else statement nodes are located in their first child, whereas for while statement nodes it is located in the second child. Using this information, we can update $C(n)$ accordingly. Finally, we also need to allocate some space for an unconditional jump located after the first statement of any if-else statement node. This unconditional jump is required to jump over the second statement after the first statement has executed. We can do this by simply setting $C(n)$ to the size of the unconditional jump if n represents the first statement of an if-else statement node, which can be verified using a similar condition used to verify if a node was the conditional of any of the branching nodes.

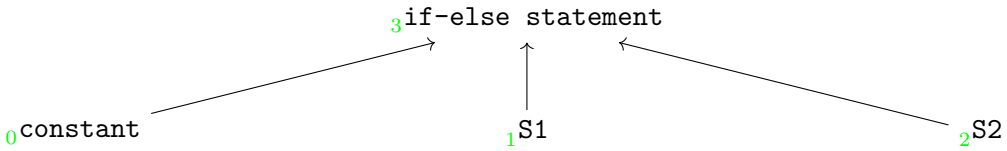
After the first stage has finished executing, we now have an instruction location for each node, with space allocated to hold instructions that do not fit the regular prefix-sum method of allocating instructions. The next step is to correct the instruction locations for nodes that did not have their locations set to the correct values by the prefix-sum and point them at the space allocated using the padding functions. We do this by going through all nodes once more and performing a check on the node type of the parent. If the parent node is one of the nodes that required correction, we create a tuple containing the parent's index, which is stored as a field in the child node and the location of the padding. The location of the padding is always located immediately after the instructions the node itself generated, which is given by the same $S_b(n)$ function used in Equation 3.2. We can then use a single scatter operation to write all these newly computed locations using the parent node indices as indices and the newly computed locations as values, thus completing the instruction location calculation.

Location	$S(n)$	$S_b(n)$	$C(n)$
0	3	2	1
3	2	1	1
5	1	1	0
2	0	0	0

(a) The instruction counts for the example nodes

LUI r69, UPPER(constant)
ADDI r70, r69, LOWER(constant)
BEQ r70, x0, 5
S1
JAL x0, 6
S2

(b) The generated instructions



(c) The abstract syntax tree

Figure 3.4. An example of instruction location correction for a conditional statement.

An example of the correction stage can be found in Figure 3.4. Here, an if-else statement using a constant as its conditional and two statements as its possible branches is being compiled. Each node in the tree is annotated with its order in the post-order representation of the abstract syntax tree, represented by a green index. Two tables are given with this tree, the first represents a breakdown of the output of the instruction location stage, where each row is ordered to contain data in the post-order representation of the abstract syntax tree. In this table the size function and its two relevant sub-components for conditional statements are given. The second table displays a set of instructions as generated by the next stage, instruction generation. In this table, the instructions for the conditional statement itself are marked red. For this example, we will assume that the two statements S1 and S2 will each generate one instruction.

In the instruction count table, we can see that the entries corresponding to the constant and S1 each have their value of $C(n)$ set to one. This creates spaces for two instructions, one immediately after the instructions for the constant and one immediately after the instructions for S1. These locations correspond to the conditional jump of the if-else statement, which should be performed immediately after the conditional and an unconditional jump over the instruction of S2, which should be performed immediately after S1. The instruction location for the if-else statement node itself is updated to point to the first instruction generated by the if-else statement node, which is located immediately after its conditional. This location is obtained during the correction performed at the end of this stage, where the value is set to the location of its conditional node, which in this case is 0 and adding the number of instructions this conditional node itself will generate, given by $S_b(0)$ which in this case is 2, thus creating the final instruction location for the if-else statement.

3.2.3. Function offset table

Given the newly computed instruction locations, we can now compute the function offset table. The function offset table maps function ids to the instruction location of the start of the function and the function size. This allows later stages to easily identify which blocks

of code belong to a single function and can also later be exported to generate information for the linker.

First of all, to compute the function offset table, a single filter operation is performed on the abstract syntax tree that selects any function declaration nodes. These nodes represent the declaration of a function and have the function identifier stored in their node data field. Due to the abstract syntax tree being stored in post-fix order, these nodes are always located at the very end of each function. Since instruction locations were computed using a prefix-sum, the instruction location assigned to these nodes will always be immediately after the last instruction of any of its descendants. Since the descendants of a function declaration node encompass the entire function, this means the instruction location will be at the very end of the function, with only the instructions generated by the declaration node itself being left over before the end of the function. Therefore, if we add the size of the function declaration node, given by the function S_b we used earlier in Equation 3.2, to its instruction location, we will always end up one instruction after the last instruction in the function, which will either be the first instruction of the next function, or the total number of instructions in the program if the current function was the last in the program.

By mapping each of the found function declaration node to this offset, we can now easily compute function start locations and sizes. The starting location of function i can be found as the value computed for function $(i-1)$, with the exception being the very first function, which simply has starting location 0. By rotating the entire array one place to the right and setting the value of the first function to 0, we can thus obtain an array mapping the i-th function to its starting location. The size of a function can then be found by taking the value stored for the function in the the original non-rotated array and subtracting the starting location from it, giving us an array mapping functions to their respective sizes, thus completing the function offset table.

Note that all of these operations can be performed efficiently on GPU architectures. The initial mapping of functions to the location one past the end of the function can be performed concurrently for every function. The rotation operation can be performed efficiently on GPUs. And the final size calculation can be performed entirely in parallel for each function again.

3.3. Instruction Generation

After the instruction counting stage calculated the locations for each node, the instructions themselves can be generated. Instruction generation is done using a tree traversal which calls the compilation routine for a set of nodes, all of which are then compiled in parallel. The actual compilation routine then handles the real per-node compilation, which consists of determining the instruction opcodes, registers and branch targets. The instruction generation routine compiles the abstract syntax tree into an intermediate instruction representation, consisting of an opcode, an infinite set of registers and a separate jump target given as an absolute instruction location.

3.3.1. Tree traversal

The core of the instruction generation algorithm is the tree traversal algorithm, which is responsible for scheduling which nodes can be compiled in parallel and in what order they need to be compiled to ensure that all data is passed correctly from the child nodes to the parent nodes. Our tree traversal makes a number of assumptions about the abstract syntax tree and the compiled code:

- A node's instructions and instruction locations can be determined uniquely from its node type, resulting data type, base instruction location and its child nodes.
- There is a fixed upper bound on the number of instructions any given node can generate.
- Nodes requiring data from their children have a fixed upper bound on the number of children that generate data required to compile the parent node and each child can only pass one entry to its parent.

From the first assumption, we can conclude that a node can always be compiled after all of its descendants have been compiled. Since descendants are always located deeper in abstract syntax tree than their ancestor node, we can schedule the tree to compile bottom-up, thus ensuring that child nodes are always compiled before their parents and that all information required to compile a node is always computed before the node is compiled. All nodes having the same depth in the syntax tree will not have any dependencies on each other during compilation and can thus always be compiled in parallel.

To propagate data up the syntax tree, we make use of the third assumption. Assuming the fixed upper bound is an integer n, if we allocate space to store n data elements per node, we can assure that we have enough space to store every piece of data that has to be propagated up the syntax tree. We then define the location of the i-th child of the node with node id x to be located at index $n * x + i$ in this buffer. Using this formula, the parent node can easily find its arguments with a single lookup and children write to the correct location in the parent array using a combination of the parent node id and the child index stored in the child node itself.

To complete the tree walk, we thus walk through the tree from the bottom layer of the tree to the top layer, calling the compilation routine for each node in parallel. The compilation routine, described in Section 3.3.2, then takes the given node and compiles it into a list of instruction locations and instructions. The instruction locations describe the final instruction address for the generated instruction. Note that due to assumption 2, we can simply allocate a fixed-size array to hold these results and use invalid instruction locations to mark instances where a node generated fewer instructions than the upper bound. This allows us to use a single scatter operation per layer of the tree to write all results back to the instruction buffer. The per-node compilation routine also returns the index and node data to write into the ancillary data array, which is used to pass results up to the parent. Since only one element can be generated per node, a fixed-size array can be used to hold these results and they can then be scattered to the ancillary data array in a similar fashion to the instructions themselves.

3.3.2. Per-node compilation

Using the tree walk, we can compile nodes in parallel. To compile the nodes themselves, the compiler needs to compute the opcode, registers and optional jump target to fill out the instruction information. One node can generate multiple instructions however and these instruction can also be compiled in parallel. To allow for per-instruction parallel compilation, we use the assumption made in Section 3.3.1 that each node can generate at most n instructions, with n being a fixed constant integer. For each node, we can now run n threads in parallel, where thread i compiles the current node has i instructions, or returns an invalid instruction offset if it does not. This ensures that the scatter operation used in the tree walk will only write back valid instructions, as invalid instruction locations are not written back to the final instruction buffer and thus ignored.

The compilation routine now goes through a sequence of steps to finalize the actual instruction, namely determining the registers, opcode and jump target and location for the generated instruction. Combining these four results yields our final instruction.

Instruction location

The first thing the per-instruction compilation routine determines is the actual instruction location of the current instruction. The instruction locations calculated during the instruction counting stage give us the address where the block of instructions generated by a given node would start. Thus for most of the nodes, we can simply add the offset of the current instruction within the node to the instruction location stored for the node to get the final instruction address.

There are however a few exceptions, which are the same exceptions as described in Section 3.2.2, namely the nodes representing branches and the nodes representing function call arguments. For nodes representing branches like the if-else node or the while loop node, the instructions themselves are split, with a conditional branch instruction between the conditional and the first statement and an unconditional jump after the first statement. Since the correction performed during the instruction counting stage already set the location of these nodes to point immediately after the conditional, we do not need to handle the conditional jump as an exception, but we do need to know where first statement node ends to calculate the final instruction location for the unconditional jump. Since these nodes are always some type of statement node, which does not have any return register, we know that this child's entry in the buffer used to propagate data up the syntax tree is still available. Furthermore, each child node knows the exact instruction location of the last instruction they generate, as they need to calculate this themselves during their instruction location calculation. Therefore, nodes simply check if they are the first statement node in a branching construction and if so simply propagate this location up the syntax tree using the buffer usually used for resulting registers. The parent node now knows exactly where its child node ends and can thus simply use that as the instruction location for its unconditional jump.

The second exception are function call argument nodes. These nodes generate instructions setting up the stack before a call and destroying it after the call. The correction performed

in Section 3.2.2 will set the instruction location of this argument list node to immediately before any of the instructions generated by the arguments, which is the location where we would normally allocate stack space to hold arguments passed via the stack. As a result the instructions setting up the stack can follow the default calculation for instruction locations. However, the destruction of this allocated stack needs to be done after the call instruction itself, thus we need to calculate the location. Since the abstract syntax tree is still in post-order, we can simply look at the node immediately before the function call argument list node to determine how many arguments were passed and how many instructions these would generate. We know that if there are arguments, the node immediately before the function call argument list node is one of the three node types emitted in the preprocessing stage described in Section 3.1.1. The child index of this node would give us the total number of arguments and the combination of node type and node data respectively will tell us how many integer or floating point arguments there are, allowing us to fully calculate the number of instructions all arguments would generate. By adding this to the instruction location for the function call argument list node calculated in the instruction counting stage, then adding the number of instruction a function call would generate, we arrive at the location of the instruction immediately after the call, giving us the instruction location for the instructions required to clean up the stack.

Opcode

To determine the opcode, the compiler uses a simple lookup table. The instruction to be generated for a node is uniquely identified by the node's node type, the node's return type and the instruction offset within a node. Due to the preprocessing described in Section 3.1, no exceptions occur here and a simple lookup is sufficient to determine the opcode for every valid node that may appear in a program.

Registers

To simplify the compilation process, we first compile the code into an infinite register architecture. This allows us to postpone the process of allocating registers and stack locations to a later stage. Determining free registers is a process that is highly dependent on the instructions around the current instruction, which complicates compiling our instructions in parallel. Using an infinite register set alleviates this problem. We do however impose a number of restrictions on our register set:

- Each register may be written to only once.
- Each register may be read from multiple times, but only if no writes are performed between the first time the register is read and the last time the register is read.
- Each register must be written to before it is read from.

These restrictions are in place to simplify the implementation of later stages such as optimization and register allocation. Note that these restrictions make our intermediate representation similar to a static single assignment form representation of the RISC-V

architecture. The benefits of this representation are similar to those that static single assignment form has in traditional compilers, such as allowing for easier detection of unused variables and easing register allocation. Our restrictions are slightly stricter than those of static single assignment, requiring no writes to occur between reads of of a register. This restriction was added to simplify the register allocation stage, which can mark a physical register as free the moment a register is read from, even if further reads can still occur, as no new allocations should occur before the register is truly free. Sections 3.4 and 3.5 describe the benefits of this representation for optimization and register allocation respectively in more detail.

The next task is to determine the source and destination registers for the current instruction. Note that due to the parallel compilation of instruction within a node, we need to ensure that each instruction can compile itself fully independently from any other instructions in the current node. This might pose a problem if we need to use resulting registers from earlier instructions as operands to the later instructions generated by a single node. We resolve this issue by forcing a convention on the resulting registers. At this point, we are still using an infinite register set, which allows us to simply compute the output register immediately from the instruction location, as we can ensure that these are unique and that our requirement of only writing to each register once is met.

This convention allows us to easily calculate the output register if the instruction requires one. The registers containing the instruction's operands are contained within the ancillary data array described in Section 3.3.1. Once a child finishes compilation, it will set its entry in its parent's portion of this array to its resulting register. The parent can then retrieve the register containing its child node's register directly from this array. The only exception are nodes using multiple instructions, in which case the register is not located in this array, as the instruction generating this register will be computed in parallel with the instruction using the register. However, in this case we already know that the instruction generating this register is being computed from the same node data as the instruction using the register. We can thus simply compute the register this instruction will generate from the convention relating output registers to instruction locations.

Jump target

The final element of the instruction that has to be determined is the jump target for jump instructions. On the RISC-V architectures, jump are usually stored as relative offsets from the current instructions, however, during this stage of the compilation we instead store the jump target as absolute addresses. This has a number of advantages. First of all, this allows later stages to easily find the instruction jumped to by directly indexing the instruction array with the jump target, which can be useful for certain optimizations. However, the main benefit is that it allows later stages to add or remove instructions more easily as jump targets can be more easily recalculated from absolute addresses. This recalculation is described in Section 3.7. There are three significant types of nodes that will generate jump instructions, the jump target of which will have to be calculated in three different ways.

The first type of nodes that generate jump targets are the branching nodes. The jump targets of these instructions are all local to the function, with the jump targets being

either immediately before or after the instruction of one of the branching node's children. To determine the jump target location, we re-use the same data was used to determine instruction locations, described in Section 3.3.2. To determine the location where an instruction ends, we take the instruction location computed for the child node and add its size as given by the function $S(n)$. This gives us the location of the instruction immediately following the last instruction of the child node. For if conditionals and if-else conditional, this contains all the data required to determine the jump target, as their jumps only target instructions immediately following their own child nodes. However, for while loops, the compiler also needs to generate a jump at the end of the loop which targets the first instruction of the conditional of the loop. Normally, this would be the first child node of the loop and since our method only allows us to determine where a node ends, we would not normally be able to use this method to determine the first instruction of the first child node of a node. However, we can resolve this by simply adding an extra child node that does not generate any instructions as the first child of a while loop. The first element of the child data array would then contain the location of the end of this zero-instruction node, which corresponds to the first instruction of the second node, which would in this case be the conditional. This allows us to determine the target of jumps to the start of the loop correctly.

The second type of node type are return statement nodes. Return instructions need to generate a jump to the function epilogue, thus we need to determine where the function ends. However, during the instruction counting stage, we generated a function table containing the start and size of each function, described in Section 3.2.3. Since return nodes contain the id of the function in their node data, we can simply use this as an index to the function table to determine the location of the end of the function. Simply subtracting the size of the epilogue from this address gives us the target of the jump to the function epilogue.

Finally, there are function call nodes, which need to generate a jump to the start of a remote function. We can use a similar strategy as we used for return nodes here, since the function table contains the location where each function in the program starts. Furthermore, the node data of a function call node also contains the id of the function to be called. Simply indexing the table with this id gives us the absolute jump target for the call, finishing the jump target calculation.

3.4. Optimization

After the initial instructions have been generated, initial optimizations on this instruction stream can be performed. Any optimization that does not require the information about the physical registers to be used can be performed at this stage. This includes optimizations such as dead code removal or constant folding. Many of these optimizations involve the removal of instructions. However, removing elements from the actual array of instructions is an expensive operation on most GPU architectures. To remove multiple elements from an array, the new locations for all instructions in the program will have to be computed using a prefix sum over an array indicating where the sizes of all instructions to be removed should be set to zero, after which all instructions need to be scattered into their new locations. Furthermore, any references to instruction locations, such as the entries of the function

offset table or all jump targets, need to be updated to reflect the new location. Performing this process repeatedly for large instruction buffers would be inefficient. Therefore, an instruction mask is introduced which later stages can use to determine whether certain instruction will be optimized away or not.

The instruction mask is a mask which indicates whether or not any given instruction in the program is enabled. Instead of recomputing the instruction array after every removal, only the mask is updated. Later stages can now check this mask to determine whether an instruction is still enabled and ignore the instruction if it is not. This allows us to remove instructions by setting the instruction's corresponding entry in the mask to false. The actual removal of instructions and correction of data structures is delayed until the very end of the compilation process, the instruction removal stage of the compiler, which is described in Section 3.6. The instruction removal stage is run right before the finalization of the instructions in the last two stages, since no instructions are removed beyond this stage and the jump linking stage requires the new instruction locations to be calculated. This means we can update the entire list of instructions in one single operation instead of performing these operations separately during each stage. Since a number of data structures generated and used by later stages of the compilation process will no longer be in use by the time this stage is executed, we can also forego updating these data structures entirely, further optimizing the compilation process.

At the moment of writing, the Pareas compiler is equipped with a single major optimization, dead expression removal, which will be described in the next subsection. In the future, this stage can be extended to include more optimizations.

3.4.1. Dead expression elimination

Our compiler implements dead expression removal, a sub-type of dead-code elimination which removes any expressions of which the result is not used or stored in a variable from the program. A further side effect of this optimization is that it ensures that every register that is written to, is read from at least once, which is a property the next stage, register allocation, described in Section 3.5, can use to optimize register usage within a function.

To efficiently perform dead expression removal in parallel, the compiler performs three major steps. First, it creates a table which maps each virtual register to a boolean constant, which is true if the register appears as a source operand of any instruction in the program and false if it does not. If a register is set to false, this effectively means that the register itself is not used in any instruction and the code to generate it can thus be removed from the program.

In the second step, the compiler determines which registers were involved in the computation of any unused registers. To do this, it simply looks at the destination register for all instructions and marks the operands to this instruction as unused by setting its entry in the table to false. One exception to this rule is if the instruction performed has a side-effect, such as a store operation, in which case the operand is not set to false to preserve instructions that create side-effects even if they do not yield any results used by other instructions. This step is then repeated until no modifications to the table are made, at which point all unused registers have successfully been marked.

Since register numbers and instruction numbers are closely correlated due to the register numbering convention described in Section 3.3.2, the generated table describing whether a register can be optimized away can be used to determine whether the corresponding instruction that generates that register can be optimized away as well. The only exception here is once again instructions that generate side-effects, such as memory store operations. Thus, as the final step, the table is updated by setting all entries for side-effect inducing instructions to true. Note that the operands to these instructions are already marked as instructions that should remain in the program due to the check on side-effects during the second stage of the algorithm.

After this algorithm has finished, a mask determining which instructions should remain in the program and which instructions should be removed has been generated. Note that for all steps in this algorithm, the actual computation can be run in parallel for each instruction in the program. Since each instruction can only belong to at most one independent expression to be optimized away and all independent expressions can be analyzed completely in parallel, the number of iterations will always be the depth of the expression with the maximum expression depth in the program. However, due to the vast majority of instructions appearing in expressions having two operands, on average, the maximum expression depth will be related to the binary logarithm of the number of instructions of the longest expression in the program. On average a program will contain a great number of small expressions however, with even the number of instructions of the longest expression being relatively small in comparison to the total size of the program, thus the algorithm should be efficient for usage on the average program.

3.5. Register Allocation

After initial optimizations have been performed, the compiler will start the register allocation stage of the compilation process. This stage is responsible for taking the infinite set of virtual registers the earlier stages have operated on and transforming this set into a finite set of physical registers corresponding to those available on the RISC-V architecture itself. The compiler might also have to allocate additional room on the stack if it cannot cleanly fit all used registers from the infinite register set into the finite set of registers. In such a case extra operations to load and store registers to this stack space are generated as well. We will refer to these additional memory operations as spill instructions and the registers stored on the stack as spilled registers. However, as reading and writing to memory are generally much more expensive compared to reading and writing to a register, the compiler should try to minimize the number of spill instructions if possible.

Traditional compilers tend to use a graph coloring algorithm to perform register allocation. Initially, an interference graph is constructed from the set of virtual registers. After this, an attempt to color the graph in such a way that the number of colors required is less than the number of registers is made. If no valid coloring can be found, one virtual registers is chosen to be spilled and spill instructions are inserted. After this a new interference graph is constructed from the new array of instructions and the process is repeated until a valid coloring can be found.

However, while the graph coloring algorithm itself can be run efficiently on a GPU [Gro+11],

the insertion of spill instructions between each iteration of the register allocation algorithm would impose a significant overhead. This would require not only reallocating the entire array of instructions for every iteration of the algorithm and inserting instructions into this array, which is an expensive operation. It would also require the insertion of nodes into the inference graph and a recalculation of the mapping of virtual registers to nodes in the inference graph after each iteration of the algorithm, both of which are expensive operations similar to an insertion into an array.

Instead, we decided to implement a greedy algorithm to allocate registers. While the mappings produced are not as optimal as those produced by a graph coloring approach, due to the large amount of registers present on the RISC-V architecture, this will still produce a reasonable mapping for most applications.

Register allocation can be sub-divided into multiple distinct stages. First, lifetime analysis is performed to determine which registers are in-use at which point of the program. Our lifetime analysis algorithm also makes an initial mapping of virtual registers to physical registers. After lifetime analysis, the compiler will determine the stack space each function requires to hold all spilled registers. Finally, it will calculate the number of instructions it needs to insert in order to finalize register allocation. The actual insertion of instructions and determination of the final set of physical registers to be used is delayed to the next major stage of the compilation process, instruction removal, where it is combined with the actual removal of instructions earlier stages of the compilation process may have optimized away, because the operations required to insert instructions into the instruction array are similar to those required to remove instructions from this array. Combining these two stages allows us to update this array in one large operation, thus avoiding having to perform similar operations twice.

An overview of the stages within the register allocation stage is given in Figure 3.5. Since the majority of the spill insertion substage is performed in the instruction removal stage, it is not listed as a component within the flow of the register allocation stage.

3.5.1. Lifetime analysis

The first stage of register analysis is lifetime analysis. This stage is responsible for determining the initial assignment of virtual registers to physical registers and for determining which registers need to be swapped out due to spilling. The results of this stage are stored in a table mapping virtual register numbers to a combination of one physical register number and one swap bit. The physical register number simply identifies which physical register to use for this virtual register when the register is used as a destination register and the swap bit is a bit used to identify if a register will need to be written to the stack at any point in its lifetime.

The compiler now needs to create the table, which can subsequently be used to finalize register allocation. To achieve this, the compiler initializes a register usage mask for each function. This mask contains one bit for every physical register in the RISC-V architecture, which is set to one if a register is in-use and to zero if it is free. The mask is then used to determine which physical registers are available for use during the compilation process. The lifetime mask is initialized to zero for any free registers that can be used within the

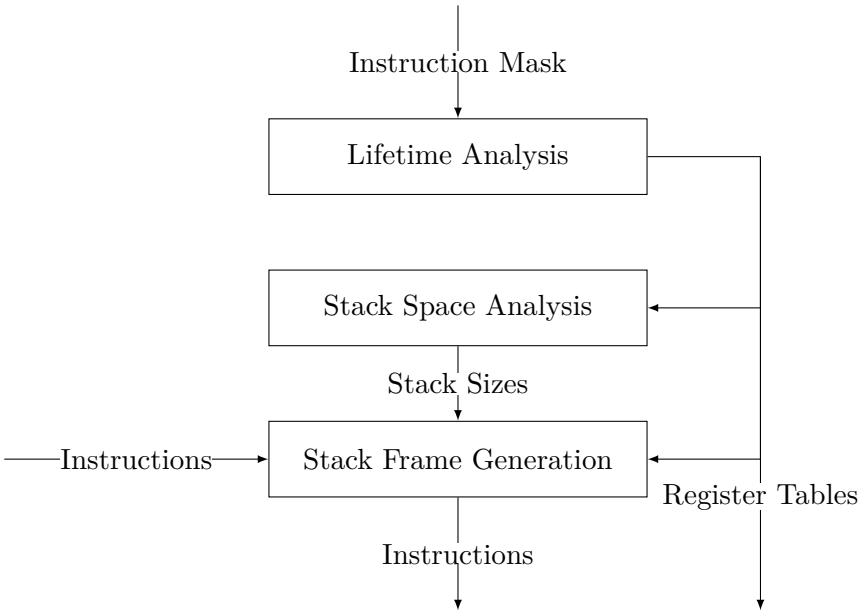


Figure 3.5. A schematic overview of the sub-stages of the register allocation stage

function and set to one for any register that cannot be used for general operations, such as the stack pointer register. This way, we can guarantee that our compiler only allocates registers that are available for use.

With the lifetime mask created, the compiler goes through each instruction of the function sequentially and greedily assigns registers as they become available. If an operation uses a register as an operand, the corresponding bit in the lifetime mask is cleared, as the register should now be free for use again. After freeing operand registers, the compiler tries to find a zero bit in the lifetime mask and sets it to one, thus assigning the corresponding physical register to the destination register of the instruction. If an instruction uses a hard-coded register, no allocation is performed, the lifetime mask is simply updated to reflect the usage of the hard-coded register.

The lifetime masks can also be used to keep track of which registers have been used within a function. By keeping track of the bitwise or of all values the lifetime mask has taken within a function, we will end up with a mask containing a bit of one for each register the function has touched. A bitwise and of this computed mask and a mask containing a one for each bit representing a non-scratch register in the architecture's ABI results in a mask that describes exactly which registers will have to be preserved during the function prologue and restored during the function epilogue.

If no valid zero bit can be found, the compiler will pick a bit set to one from a statically defined set of suitable scratch registers. However, since this register is already in use by another virtual register, we need to swap the other virtual register out before we can use the new register. This is done by keeping track of a set of swap bits for each virtual register, which is set to one if the virtual register needs to be swapped out at any point and set to zero if it can remain within its register for its entire lifetime. If a register which has its swap bit set is later used as an operand, the compiler will need to load the value into a register before it can be used as an operand to the instruction. However, instead

of allocating a new register here, the compiler will simply use a predetermined register. Since the register becomes available for usage immediately after the instruction is over, the lifetime mask does not have to be updated in this scenario. The swap bit itself can be used to tell what register should be used when a virtual register is used as an operand, if it is set it is simply the predetermined scratch register and if it is not, the value is still in the same physical register as it was when the value was initially created, which is already stored in the lookup table.

The lifetime analysis also takes into account function calls by detecting far jumps, which can be done using the jump target field of the instruction. On a function call, any register defined as a scratch register in the architecture's ABI will need to be marked as cleared. This is achieved by clearing the bits corresponding to any scratch register in the lifetime mask, thus freeing up the registers. We also set the swap bit for any virtual register that happened to be occupying any of these scratch registers, thus ensuring that the value is not lost after the function call.

Since the lifetime analysis procedure is a sequential process on the instructions within a function, it can only be performed in parallel for different functions, meaning that for the process to be efficient, a significant number of functions need to be compiled at once.

3.5.2. Stack space analysis

After the lifetime analysis stage determined register usage, the compiler can move on to computing the stack space required to store all spilled registers. The stack space analysis sub-stage of register allocation has two tasks. First, it needs to compute the total stack space to allocate the spilled registers for each function and secondly it needs to assign each register requiring stack space a unique location on the stack.

Since the lifetime analysis stage sets the swap bit if a value has to be moved to memory at some point during the function, we can determine for which virtual registers we will need to allocate stack space by checking the swap bit. Furthermore, since virtual register numbers are assigned using a convention that is directly dependent on the instruction number of the instruction that creates the value stored in the virtual register, as described in Section 3.3.2, we can partition the virtual register table into parts where each part represents one function. By using the function offset table, of which the creation was described in Section 3.2.3, we can obtain the instructions where a function starts and ends and calculate the virtual register numbers corresponding to these instructions. Using these virtual register numbers as the bounds of a partition, we can partition the virtual register table.

Using these properties, the compiler can now efficiently determine stack offsets using an exclusive segmented prefix sum operation, described in Section 2.2.1. By mapping each virtual register number to either the size in bytes of its contents if the swap bit is set, or zero if it is not and subsequently performing an exclusive segmented prefix sum over this buffer where each segment represents one function, we obtain a table mapping each virtual register number to an integer offset. If a virtual register's swap bit is set, this integer offset will correspond to a unique offset on the stack where the compiler can store the value contained within this register if it needs to be swapped out.

We can also use the results of the exclusive segmented prefix sum to determine the total stack size required to hold all values of a given function. If the last element of a segment does not have its swap bit set, its value will be the total number of bytes required to hold all swapped element of the function. If the swap bit is set, then it will contain the offset of the last element, which means we need to add the size of the last element to this offset to obtain the total stack size. Since each segment represents a function, this gives us the stack size required to hold all swapped values for a function.

3.5.3. Determining spill instructions

The final part of the register allocation stage is determining the number of load and store instructions that have to be inserted in order to ensure that spilled registers are written to and read from memory if required. Two rules govern how these instructions are inserted:

- If an instruction's destination register has its swap bit set, we generate a store immediately after the current instruction.
- If an instruction's operand register has its swap bit set, we generate a load instruction immediately before the current. instruction

Using these two rules and the swap bits, we can map each instruction to an integer describing the total number of instructions it will use, including the original instruction itself. Performing a prefix sum over the obtained array of integers would give us a new set of instruction offsets and allow us to insert the instructions. However, the register allocation stage does not immediately perform this action. Instead this process is postponed to the instruction removal stage described in Section 3.6. Since this stage will have to perform a similar exclusive prefix sum to determine the new instruction offsets after the optimized instructions have been removed and then move the old instructions to their new locations in memory to create a new instruction stream, we can combine this with the insertions required by register allocation by simply delaying the prefix sum and insertion until the instruction removal stage. This way we only have to perform this relatively expensive operation once instead of twice.

3.5.4. Inserting stack frames

Once register allocation has been completed, information on the size of the stack frames is available. Using this information the instructions to subtract or add to the stack pointer to create the stack frames can be created. While it would be possible to insert these instructions similarly to spill instructions by inserting them during the instruction removal stage, this would require additional logic in the next stage. Instead, we can optimize this by allocating room for these instructions in advance. By having nodes corresponding to function declarations return non-zero values in the instruction counting stage, see Section 3.2, the compiler can allocate a fixed-size buffer of unused instructions at the start and end of each function. Once register allocation is finished, it can insert the code to create the stack frame here.

Since the function prologue and epilogue always appear at the start and end of the function respectively, we can use the function offset table to determine their exact locations. The compiler now generates the instructions required to set up the stack frame for each function and performs a scatter operation to overwrite the previously unused instructions with the correct code to set up the stack frame.

The total size of the stack frame itself can be calculated by adding the stack space required to hold all spilled registers, calculated earlier during the stack space analysis part of register allocation, to the amount of space required to store all local variables appearing in the function. The amount of space required to store all local variables is passed to the later stages from the semantic analysis stage of the compiler, which determines which local variables appear within the scope of the function. By adding these two elements, each function can calculate its stack size and generate the appropriate instructions.

3.6. Instruction Removal

After register allocation has been performed, no further removal or insertion of instructions is required, so the compiler is ready to generate the modified array of instructions. This stage is responsible for both removing instructions marked as optimized away and for inserting the load and store instructions for registers if the register allocation stage determined these registers need to be stored on the stack due to a lack of registers to hold all virtual registers. To accomplish this, the instruction removal stage goes through two sub-stages.

The first sub-stage determines the new locations for each instruction. To do this, a prefix-sum similar to the one used during the instruction counting stage, found in Section 3.2, is used. First, the compiler maps each instruction to the number of instructions it will actually end up occupying. For instructions that have been optimized away this value will be zero, otherwise the value is given by Equation 3.3.

$$I(i) = 1 + O_1(i) + O_2(i) + D(i) \tag{3.3}$$

In these equations, $O_1(i)$, $O_2(i)$ and $D(1)$ return one if the first operand, the second operand, or the destination register have their swap bit set respectively, or zero otherwise. The swap bit for these registers indicate that the register should be moved to the stack and thus either a load should be generated if it is one of the two operands, or a store should be generated if it is the destination register.

After preforming a prefix-sum $I(i)$, each instruction will have a corresponding offset pointing to a block with room for 0 to 4 instructions where the compiler can now put the instruction and any load and store operations the instruction requires. For each instruction, the compiler maps each instruction to an array of final instructions, including load and store operations, in the order they should end up in the program. If less than 4 instructions are required, an invalid value is used to indicate that the instruction is not present. The compiler now scatters all generated non-invalid instructions into the new instruction

buffer using the new instruction locations as offsets, which results in a new continuous array of instructions containing load and store operations.

Now that new instructions have been inserted and old instructions have been removed, the compiler will need to recalculate the function offset table. However, since we still have an array of new instruction offsets that can be indexed using the old instruction offsets, the compiler can index this array using the old values of the function offset table to obtain the new start and end locations for each function. Once the new start and end locations have been obtained, the compiler can recalculate the new function sizes using a single subtraction to correct the function offset table.

3.7. Jump Linking

The previous stage of the compiler removed and inserted instructions into the instruction stream, however it did not correct the jump targets for any jump instructions appearing in the program. Therefore, these will still need to be corrected. The jump linking stage is responsible for taking the absolute jump targets, correcting these to their correct addresses and finally transforming these into jump instructions available on the target platform. A total of three steps need to be performed in order to fully generate the correct jump instructions.

On the RISC-V architecture, like many other architectures, multiple types of jump instructions are available. Some of these jump instructions allow for short single-instruction relative jumps to a limited number of addresses from the location of the jump instruction, whereas others require multiple instructions but allow jumping to any address in the address space. As a consequence one jump instruction may require multiple instructions depending on the jump target, thus instruction insertion will have to be performed. However, whether an instruction has to be inserted depends on the number of instructions between the jump instruction and its target. This calculation cannot be performed until after the new offsets have been calculated in the previous instruction removal stage. Therefore the compiler first recalculates jump targets based on the results of the instruction removal stage and determines the size of the jump instructions based on those results. Subsequently a second correction to the jump targets is performed using the sizes of the jump instructions determined using that calculation.

The initial correction of jump targets is performed similarly to the correction of the function offset table described in Section 3.6. The compiler uses the newly computed instruction offset array and uses the old absolute jump targets as indices into this array to obtain the new jump targets. Since the instruction offsets in this array point to the first instruction of a block of instructions containing any register loading required to perform the actual original instruction, this ensures that we do not jump over any load or store operations the instruction itself might require to set up its operand registers.

To calculate the new jump targets, the compiler first calculates new instruction offsets for each instruction in the program, in a similar manner to how the instruction removal stage, seen in Section 3.6, computes its instruction locations. Each instruction is mapped to the number of instructions required. The compiler now has to determine how many instructions

each jump will generate. On the RISC-V architecture, this is based on two factors: whether or not the jump instruction itself is a conditional branch instruction or an unconditional branch instruction and whether or not the relative distance between the jump instruction and the target instruction exceeds a given bound. The first condition can be checked by reading the instruction opcode. The second condition however is harder to verify, as this is also dependent on the results of other jump instructions. In this case, the compiler simply assumes the worst case where every jump instruction requires a multi-instruction sequence. After these sizes have been calculated, the compiler performs another exclusive prefix sum to determine the new instruction locations and scatters the old instructions into their new locations using the newly calculated offsets. For jump instructions, the compiler instead maps these instructions to their corresponding sequence of instructions and scatters these into the instruction buffer.

After this process has completed, the compiler will have successfully linked all jump instructions to their jump targets and will have updated their jump target fields accordingly. On the RISC-V architecture, all jumps are encoded as relative offsets, so the compiler translates the absolute addresses into relative offsets by subtracting the instruction location of the current instruction from the absolute address, thus yielding the relative offsets required for compilation.

3.8. Postprocessing

At this stage of the compilation process, the compiler has all information it requires to finalize the instructions. Instruction opcodes have been generated by the earlier stages, all registers have been translated into physical registers and jump targets are now encoded as relative offsets. In this final postprocessing stage, the compiler takes all these separate buffers containing per-instruction information and performs a bitwise or to merge all this information into a single instruction opcode, thus finalizing the instruction encoding.

The compiler itself returns the finished buffer of instructions, as well as the function offset table to the CPU. The function offset table, is returned to allow the CPU to provide information on where in the instruction buffer each function starts and ends, allowing the program to pass this information to a linker if the source file is to be embedded into a larger program.

4. Implementation

To demonstrate that the proposed design can be used to transform an AST into RISC-V code, an implementation of our proposed design was written, which we named Pareas. The Pareas compiler was written in Futhark, a functional programming languages which can be compiled into GPU kernels. Futhark allows GPU programs to be written as a single functional program and will automatically generate GPU kernels and the CPU code required to use these kernels. Futhark also provides some useful primitives for GPU programming, such as parallel scan operations which can be used to implement efficient parallel exclusive prefix sums, a common operation in our implementation. These features simplify the development of GPU programs significantly, as we do not need to spend any time writing binding code in order to set up our kernels.

This chapter will describe the details of our implementation by discussing each stage in turn. Details will be given on what data structures were used to efficiently implement all of the algorithms required by our design.

4.1. Common Primitives

Our implementation uses a number of common subroutines to efficiently implement certain operations. This section describes the most important subroutines used throughout the program.

4.1.1. Exclusive Prefix Sum

One of the most important subroutines in the program is the exclusive prefix sum operation. It is used to determine instruction locations and offsets throughout various stages of the compiler. The exclusive prefix sum, defined in Section 2.2.1, can be implemented efficiently using the parallel scan primitive provided by Futhark. Since the exclusive prefix sum is a specialization of the exclusive scan, where the addition operator is used as the associative operator given to the exclusive scan operation, implementing an efficient exclusive scan would give us an efficient exclusive prefix sum. Since a scan primitive is provided, it can be used obtain an obtain an array B from the original input array A, where the element at position i in B is defined according to Equation 4.1. This gives us an inclusive prefix sum.

$$B(i) = \sum_{1}^{i} A(i) \tag{4.1}$$

After this operation is finished, we can simply rotate the array by one position. This combined operation is defined according to Equation 4.2.

$$C(i) = \begin{cases} B(i-1) & i > 0 \\ 0 & i = 0 \end{cases} \tag{4.2}$$

Applying this rotation to the result obtained from the inclusive prefix sum yields an exclusive prefix sum, resulting in a parallel implementation of the exclusive prefix sum.

4.2. Preprocessing

The preprocessing stage is divided into two sub-passes: function argument node preprocessing and comparison operator preprocessing. The preprocessing stage directly transforms the syntax tree without inserting or deleting nodes. Since no nodes are inserted or removed, this means only substitutions can appear. To efficiently implement substitutions in the syntax tree, a parallel map operation is used on the individual nodes of the syntax tree to map each node in the original syntax tree to a new node. This new node can either be identical to the old node if no substitution is required, or be a new node if a substitution is required. Substitutions for each node can be performed entirely independently and can thus be executed in parallel. The substitution performed depends on the node type. The substitutions performed by each of the two substages are detailed in the following sections.

4.2.1. Function argument node preprocessing

During function argument node preprocessing, the compiler will substitute nodes of the function call argument and function argument nodes by node types more closely related to the target calling convention. In the original syntax tree, function call argument nodes represent arguments to a function call, whereas function argument nodes represent the arguments in a function declaration. In terms of code to be generated, function call argument nodes map to instruction to set up the arguments of a function call, also known as the actual parameters, whereas function argument nodes map to the code receiving function arguments and storing them in local variables for the called function, also called the function's formal parameters.

In RISC-V's calling convention, integer arguments can be passed either in integer registers or on the stack and floating-point registers can be passed either in floating-point registers, integer registers or on the stack. To simplify code generation, three possibilities are considered:

- An integer or floating-point argument is passed in a register of its respective type.
- A floating-point argument is passed in an integer register.
- An integer or floating-point argument was passed on the stack.

In case of the first two node types, the compiler would need to know which integer register to pass the argument in. Registers are assigned in left-to-right order by default. In the case of floating-point arguments, if no floating-point registers are available, but integer registers are available, the floating-point argument is passed via the next available integer register. If no register is available, the arguments are passed via the stack, in which case the offset on the stack on which to pass the register needs to be determined. Since this data is also required to calculate in what type of node to split the original argument node types, we can simply pass this information to the later stages using the node's node data field, which will contain the register number for the two register types and the stack offset for the stack types.

To determine which of the three node types to compile a given argument node into, the compiler performs a check based on the child index of the current node. This tells the compiler what the overall index of the argument is, as well as on the node data of the node, which should be set to the index of the argument of the given type. Assume the function prototype given in Listing 4.1 is compiled.

Listing 4.1 An example function declaration

```
function f(
    int i0,
    float f0, float f1, float f2, float f3, float f4, float f5,
        float f6, float f7,
    float if1,
    int i2, int i3, int i4, int i5, int i6, int i7,
    int s0, float s1)
```

In this listing, the variable names indicate the node type and node data of the nodes corresponding to each of the arguments. The number in the variable name describes the desired value to be stored in the node data field, which is either the index of the register of the desired type to pass the argument in or the offset of the stack, given in number of elements. The remainder of the name describes the node type and data type of the node, with a node type of `i` indicating a regular argument node with an integer type, a node type of `f` indicating a regular argument node with a floating-point type, a node type of `if` indicating a floating-point argument passed in an integer register and the letter `s` indicating an argument passed on the stack.

The compiler will now differentiate between integer and floating-point arguments. For floating-point arguments, the compiler will first check if the node data field of the node is less than 8, meaning less than 8 floating-point arguments have been passed. If so, the compiler will return a default argument node with the node data unchanged, as this already contains the correct data for the node.

In all other cases, the compiler will first calculate how many integer and floating-point arguments have been passed before the current argument. Given the child index of the node and the node data field of a node, this can easily be calculated. The node data field itself should contain the number of arguments of the node's own type that have been passed before the current argument and subtracting this value from the child index should give the number of arguments of the other data type that have been passed, as the sum of all integer and floating-point arguments should always match the total number of arguments.

Given these two variables, the compiler computes an integer register index, which corresponds to the register number for the integer register to pass the current argument in. This is computed using Equation 4.3.

$$IR(n) = I(n) + max(F(n) - 8, 0) \tag{4.3}$$

In this equation $I(n)$ represents the number of integer values passed before node n and $F(n)$ represents the number of floating-point values passed before node n. Note that this equation corresponds to the calling convention. The first 8 floating-point values are passed in floating-point registers, whereas integer registers are passed via integer registers if possible. Adding the two together with a special case to ensure that the number of floating-point registers does not go below zero gives us the integer register index.

In case this value is less than the number of integer registers used for argument passing, which is 8 on the RISC-V architecture, this will be the node data for the new node. In case the node itself was a floating-point argument, we will return an float-in-int-register node, whereas if it was an integer register we can simply return the default argument node type. If the value is larger than 8, we need to pass the value via the stack as no registers are available anymore. In this case, simply subtracting 8 from this calculated value will give us the correct stack offset to store the argument at.

If we take the argument `i2` from Listing 4.1 as an example. Its node data is set to 1, as it is the second integer node and its child index is set to 10. This means that the compiler will determine there to have been 1 integer argument before it and $10 - 1 = 9$ floating-point arguments before it, which matches the function declaration. Thus the result of Equation 4.3 will return $1 + max(9 - 8, 0) = 2$, meaning it will get passed in the third integer register, which matches the calling convention.

As a final step, after all nodes have been substituted using a parallel map operation, the compiler will also set the node data of the function call argument list nodes to the total number of elements passed on the stack. To do this we need to observe the node data of the last argument. Since the syntax tree itself is encoded in post-order, this can be done by simply observing the node directly before the argument list node itself. Since at this point the nodes have already been split into multiple types, three options may occur:

- If the previous node is of a stack type, add 1 to the node data to get the total number of stack arguments.
- If the previous node is of a default floating-point argument type, some integers may have been passed on the stack. Since the node data of this type of node should still correspond to the total number of floating-point arguments, we can simply subtract this number from the child index to determine the number of integer arguments. The number of stack arguments is then given by $max(I(n) - 8, 0)$.
- Any other type of node is present, in this case, there were no stack arguments.

This operation can be performed in parallel for any function call argument list node. Once this has been performed, argument node type preprocessing is finished and all nodes have

correctly been substituted.

4.2.2. Comparison operator preprocessing

The final stage of preprocessing sets the resulting data type of any comparison operator operating on floating-point values to a float datatype. Since instruction counting and instruction generation only differentiate nodes based on data type and node type, this has to be done in order to distinguish floating-point comparison operators from integer comparison operators, as comparison operators normally return integer values regardless of their child types.

To accomplish this the compiler maps every node in the tree to both an index and a data type in parallel. If the parent of a node is a comparison operator and the node itself is a floating-point node, this parallel mapping will return the index of its parent as given by the node's parent index field and the data type float. In any other case it will simply return an invalid index.

The compiler subsequently performs a scatter operation, scattering the indices into the array of resulting data types. Since Futhark ignores invalid indices when performing a scatter operation, this operation will only be performed for the results of nodes where the parent node is a comparison operator and the node itself has a floating-point data type, all of which have their result set to point to a comparison operator with a floating-point child. This effectively overwrites the resulting data types of all comparison operators with floating-point children to have a floating-point datatype. Thus completing the preprocessing stage.

4.3. Instruction Counting

The next major stage of the compilation process is instruction counting. This stage is responsible for determining the number of instructions and the instruction locations for each node. As described in Section 3.2, the instruction counting stage works by first determining the initial instruction locations using an exclusive prefix sum over a node size function, then correcting these locations in a separate sub-stage afterwards if the node requires it. The size function $S(n)$ itself was subdivided into three components, the details of which are given in the following sections.

4.3.1. Base size function

The base size function $S_b(n)$ returns the base number of instructions a node requires to be compiled. In our compiler, a node's instructions can be uniquely determined by its node type and its resulting data type. These two properties describe the operation to perform and the datatype the operation should be performed upon.

Since just these two properties are used to determine the base instruction size, the size

function $S_b(n)$ itself is implemented as a two-dimensional lookup table, taking the node type and the data type as indices. This effectively allows the calculation of the size function on multiple nodes at once using a single gather instruction, which is an efficient operation on most GPU architectures.

Special entries such as branching operations and function call arguments, whose instruction counts are handled by the $A(n)$ and $C(n)$ sub-functions of $S(n)$, have their entries in the lookup table set to zero. This allows the instruction count for these nodes to be entirely based on the other two functions.

4.3.2. Location mapping correction size functions

The two location mapping correction sub-function of $S(n)$, $A(n)$ and $C(n)$ responsible for generating padding for arguments and conditional branch instructions respectively, are implemented similarly.

For $A(n)$, the return value should be the number of instructions required to move the arguments into their correct positions for the calling convention of the target architecture. To do this, the implementation of $A(n)$ makes use of the fact that the abstract syntax tree is located in post-order and that the last child of n should be located immediately before n in the array holding the abstract syntax tree. The implementation of $A(n)$ is a single check to determine whether n is an argument list node and whether the node before it is a call argument, in which case it returns the child index of the node with a fixed constant added to it. This computation can be explained by to the fact that both moving a value from one register to another, as well as moving a value from a register to a predetermined location on the stack, can be done in a single instruction, implying that the total number of instructions required to implement argument passing is equal to the total number of arguments. The fixed constant is the number of instructions required to create space on the stack. The compiler unconditionally creates space on the stack before a call, even if no arguments are passed on the stack, in which case zero bytes will be allocated. This operation is however easy to detect during optimization, in which case it can be efficiently removed from the code. The implementation of $A(n)$ can on most GPU architectures be implemented using conditional instructions, avoiding branch divergence.

The implementation of $C(n)$ is very similar to that of $A(n)$. In this case, the compiler simply checks if the parent of the current node is a node representing a branching statement and the child index of the current node matches the child index after which the jump instructions should be inserted. This once again translates into a single select operation based on this conditional, which can be efficiently executed on most GPU architectures.

4.3.3. Location mapping correction

After the initial locations have been calculated, the compiler will have to perform a number of adjustments to the calculated locations for those nodes where the instructions are not placed immediately after its child nodes. To do so, the compiler will generate two values for each node: an index and a location. This index represents an index into the array of

instruction locations where an update to the location should be performed and the location describes the new location to be stored.

Similar checks will be performed for branching and argument type nodes as described in Section 4.3.2. For children of conditionals, the index will be set to the parent node's index, which represents the conditional itself. Whereas for argument nodes, the node's own index is returned. In case no updates need to be performed, the compiler will set the index to a value outside the range of the array of sizes.

After the indices and locations have been calculated, the compiler will apply a scatter operation to write all calculated sizes back into the instruction location array. As this uses Futhark's scatter function, which performs a bounds check before writing an element, invalid indices will simply be ignored and no updates will be performed for these elements. This allows us to effectively overwrite only these elements that require an update to their instruction location.

4.4. Instruction Generation

The next stage of the compilation process is generating actual instructions from the abstract syntax tree. The compiler uses a bottom-up tree walk to compile nodes into instructions, compiling elements on the same layer of the tree in parallel. Each instruction generated by each node is compiled in parallel and instruction opcodes, operand registers, resulting registers and jump targets are determined. The following subsections outline the implementation used to perform the tree walk and to determine each aspect of each instruction.

The instructions themselves are stored in five distinct parts. Each instruction has an opcode, a destination register, two source operand registers and a jump target. Immediate values are stored directly in the opcode upon generation and thus do not require an extra field of their own. If one component of the instruction is not required, its corresponding field is set to zero.

4.4.1. Tree walk

The main component of the instruction generation stage is a bottom-up tree walk designed to compile all instructions generated by all nodes present on the same layer of the abstract syntax tree in parallel. To achieve this, the compiler first allocates a buffer capable of holding all instructions that are going to be generated. The size of this buffer is obtained by taking the last element of the instruction count array, which describes the instruction count of the very last node in the program. This node should always be a list node under which all function declarations are located. Since this node does not generate any instructions by itself and is always located at the very back of the array due to the post-order storage of the abstract syntax tree, this element will have its instruction location set to the location immediately after the last real instruction in the program by the exclusive prefix sum used to determine instruction locations, which matches the total number of

instructions to generate.

An additional buffer is allocated to hold data that will be passed up the tree during the compilation process. This buffer will mostly hold register numbers in which child operations have stored their results, such that the parent is able to access these results. In some circumstances also contain instruction locations if required. These exceptions are described in more detail in Section 3.3.2. Each parent will have a fixed number of entries in this buffer allotted to it and child nodes can write to a parent's slots in this buffer to pass data to it. The total size of this buffer equals the total number of nodes in the syntax tree times the number of elements per parent, so to avoid an overly large memory overhead, we want to reduce the number of slots per parent as much as possible. In our case, the minimum number of slots required per node was three.

After the buffers are ready, the compiler can perform the bottom-up tree walk. To determine which elements of the tree are located on the same layer, the compiler generates an array of indices, one for each node in the tree. It then sorts these indices based on the depth field of the tree node they point to using a radix sort. Radix sorting can be efficiently implemented in parallel and is thus an efficient way of sorting the tree based on node depth. We sort the array of indices instead of the actual nodes themselves, as this allows us to still access the tree itself in post-order if required. The compiler now creates a maps each element of this sorted array to a value of zero or one, depending on if the depth field of the given element is not equal to the depth field of the previous element. This indicates that the depth has changed on this current node, meaning it is the start of the next layer. We can now filter to keep only the elements where this value was one, thus filter out only the start indices of new layers, to obtain an array describing where each layer starts. The end of a layer is then given by the element before the start of the next layer, which would be stored in the element that immediately follows the start of the current layer. This allows the compiler to determine where a layer in the tree starts and ends and obtain the indices of the nodes on any given layer.

The tree walk itself is a loop from the maximum depth to zero, using this array of layer start positions to obtain all nodes in a given layer. The compiler will now map each node to a list of two pairs of values. The first pair contains an instruction location and all instruction data, such as opcodes, registers and jump targets. The second pair contains an index into the array used to pass data to the parent and the actual data to be passed. At the end of each loop the compiler scatters each of these two pairs into their respective buffers, thus completing the tree walk.

An example of compilation performed by the tree walk can be found in Figure 4.1. This figure describes the syntax tree for a program of the form $x + y - z$, where x, y and z are constants. In the syntax tree, the position of the nodes in the post-ordering of the tree are annotated as green numbers and the instruction location as would be returned by the instruction counting phase are annotated in red. The nodes belonging to the current layer of the tree the tree walk is about to compile are marked in yellow. Furthermore, two tables are provided, corresponding to the buffer used to pass data to the parent nodes, where each node has exactly three elements to store information from its children and a table containing the instructions as they are generated by the tree walk.

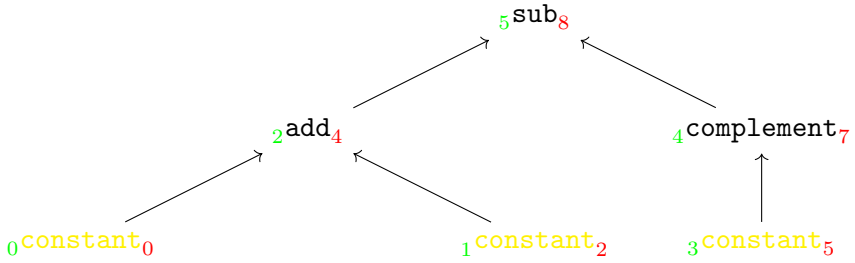
Figure 4.1 denotes the initial state before the first iteration has started, where all data from

children is still set to zero and no instructions have been generated yet. The bottom layer of the tree is marked as active. The next stage of the algorithm, performed after the first iteration, can be found in Figure 4.2. Here we see that the instructions for the constants at the bottom of the tree have been generated and that the entries corresponding to their parents in the child data buffer have been set to the resulting registers of the respective constant calculations. Figure 4.3 shows the continuation of the compilation process for the next layer of the tree, where instructions using the previously generated registers have been generated. Finally, Figure 4.4 shows the end of the tree walk, where all nodes have been compiled.

0	0	0
0	0	0
0	0	0
0	0	0
0	0	0
0	0	0

(a) The buffer used to propagate data to the parents

(b) The generated instructions



(c) The abstract syntax tree

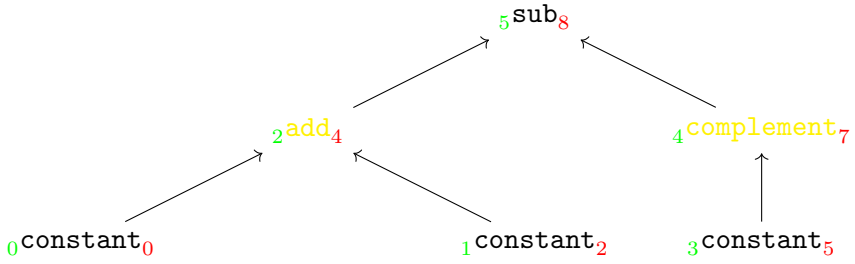
Figure 4.1. An example compilation using the tree walk.

0	0	0
0	0	0
65	67	0
0	0	0
70	0	0
0	0	0

(a) The buffer used to propagate data to the parents

LUI r64, UPPER(constant)
ADDI r65, r64, LOWER(constant)
LUI r66, UPPER(constant)
ADDI r67, r66, LOWER(constant)
LUI r69, UPPER(constant)
ADDI r70, r69, LOWER(constant)

(b) The generated instructions



(c) The abstract syntax tree

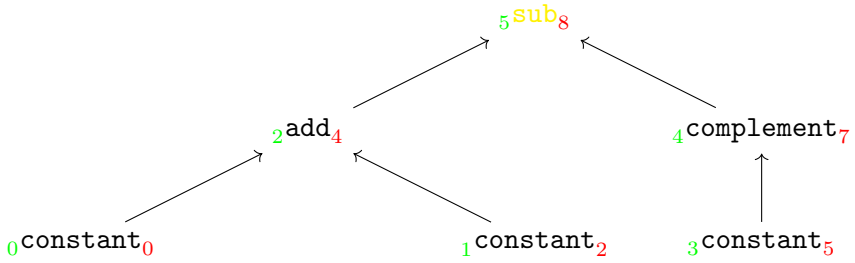
Figure 4.2. An example compilation using the tree walk after compiling the bottom layer of the tree.

0	0	0
0	0	0
65	67	0
0	0	0
70	0	0
68	71	0

(a) The buffer used to propagate data to the parents

LUI r64, UPPER(constant)
ADDI r65, r64, LOWER(constant)
LUI r66, UPPER(constant)
ADDI r67, r66, LOWER(constant)
ADD r68, r65, r67
LUI r69, UPPER(constant)
ADDI r70, r69, LOWER(constant)
XORI r71, r70, -1

(b) The generated instructions



(c) The abstract syntax tree

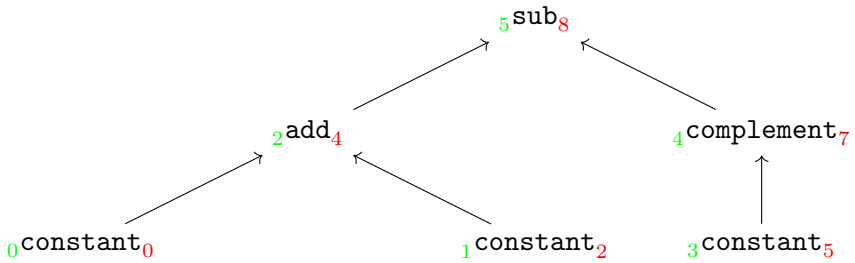
Figure 4.3. An example compilation using the tree walk after compiling the second layer of the tree.

0	0	0
0	0	0
65	67	0
0	0	0
70	0	0
68	71	0

(a) The buffer used to propagate data to the parents

LUI r64, UPPER(constant)
ADDI r65, r64, LOWER(constant)
LUI r66, UPPER(constant)
ADDI r67, r66, LOWER(constant)
ADD r68, r65, r67
LUI r69, UPPER(constant)
ADDI r70, r69, LOWER(constant)
XORI r71, r70, -1
SUB r72, r68, r71

(b) The generated instructions



(c) The abstract syntax tree

Figure 4.4. The conclusion of compiling an example tree using the tree walk.

4.4.2. Per-node compilation

To compile a node, the compiler makes use of the fact that there is a fixed upper bound to the number of instructions. For each index under this upper bound, the compiler will perform a check whether or not a node has that many instructions and if so perform the actual per-instruction compilation routine, if not, the indices in the return value of the current instruction will be set to an invalid instruction and an invalid parent data buffer index. Since these indices are used in a bounds-checked scatter operation by the tree walk, setting these to invalid values will simply ignore these entries, effectively meaning nothing is returned. To check if a node has a given instruction, a lookup table containing boolean values and indexed by node type, resulting data type and index is used. This table is similar to the lookup table used for the base size function during the instruction counting stage, given in Section 4.3.1. This table however does specify that conditional branches and argument nodes have instructions, unlike the table used during the instruction counting stage where the table contained that these nodes had zero instructions since their instruction counts were handled separately.

Since most GPU architectures will have to perform both branches of a conditional regardless of the result of the condition, we need to make sure that the upper bound on the number of instructions is as low as possible to avoid having a lot of GPU threads idle. To accomplish this, nodes that would generate large amounts of instructions have been split into multiple smaller nodes, each of which generate only a part of all instructions normally generated by the single larger node. This allows us to keep a tighter upper bound and allowing both parts of this node to be run in parallel with other nodes.

4.4.3. Per-instruction compilation

Each instruction will now determine its own location, destination register, opcode, operand register and jump target and immediate, as specified in Section 3.3.2. The actual implementation of the calculation of each of these components is described in the sections below.

Instruction location

The instruction location calculation for most nodes is simply the node's base instruction location, with the offset of the current instruction added to it. There are however two types of exceptions: branching statements and function call arguments.

In the case of function call arguments, the second instruction of the node will have to be placed after the call instruction, with instructions setting up the arguments themselves between the location of the first instruction of the call argument list and the call itself. Thus an offset needs to be added to the initial location. As described in Section 4.3.2, this setup requires one instruction per argument. Therefore we obtain the number of call arguments from the node before the call argument list, which should be the final call argument and use its child index as the total number of argument. We then add this to the instruction location of the first instruction and add an offset describing the number of instructions the call itself takes up, thus putting the location after the call.

In the case of branching instructions, the instruction location is stored in the data array used to pass data from child to parent. The instruction location returned is therefore directly obtained from this buffer. To set up these elements, the compiler checks whether the parent node was one of these conditional nodes and if so sets the propagated data element returned by the per-instruction compilation function to the instruction location of the child element. Since all children of conditional nodes are statements themselves, which do not otherwise return any data to their parent, these slots are guaranteed to be free and can thus be used to pass the child's instruction location to the parent.

Opcode

Opcodes are uniquely identified by their node type and resulting data type. To obtain the opcode, the compiler simply performs a lookup in a lookup table indexed by these two fields to obtain the opcode. This can in turn be translated into a single gather operation, allowing us to efficiently retrieve the opcode for many instructions at once.

Operand registers

The calculation used to retrieve operand registers can vary between nodes. For the vast majority of expression nodes, these operands are simply the results of their children, with the first operand being the result of the first child and the second operand being the result of the second child if it exists. In this case, the operands can be directly accessed from the buffer containing data propagated from child to parent, using the operand number as the offset from the start of the parent's part of this buffer to determine operand register. A few expressions use a similar method but require the two operands to be reversed, so the second operand to the instruction should be the result of the first child and vice versa.

For a small number of multi-instruction nodes, the operands are the result of the previous instruction. In this case, due to the convention that virtual register identifiers should simply be the instruction location of the instruction with a fixed constant added to them, the register can easily be obtained by simply subtracting one from the constant that would be generated for this instruction, to obtain the output register of the previous instruction.

Some instructions operate on a predetermined register, such as the stack pointer or base pointer for operations such as access to local variables, or the accumulator register for instructions moving the return value of a function into another register for later usage. In this case, the register number of this fixed register is returned.

Finally, a large set of statement nodes and other nodes have no operands. Since the final operands are later inserted into the instruction using a bitwise or operation during the post processing stage, setting the operand to zero here would make sure nothing is changed by this bitwise or operation. Thus all these nodes simply return a operand register with identifier zero.

To efficiently deal with all five of these options, the compiler uses a lookup table indexed by node type, node return data type, operand number and instruction number to determine

which of these five calculations is required for the current instruction. The compiler then uses a number of select statements to determine the final result based on the value in the lookup table. This approach reduces the number of branches required significantly by grouping all instructions with the same type of operand together. By keeping the number of options as low as possible, we can ensure that all operands can correctly be determined without incurring a large performance penalty.

Destination register

The destination register is determined using a fixed naming scheme to simplify access to instruction results and to allow each instruction to generate a unique output register without requiring any form of synchronization between threads. The output register is given using the calculation in Equation 4.4.

$$R_o(i) = I_l(i) + 64 \tag{4.4}$$

In this equation, $I_l(i)$ describes the instruction location of the given instruction. The constant of 64 corresponds to the total number of registers available on the RISC-V platform. Our compiler uses an internal register naming scheme where registers 0 to 31 correspond to integer registers x0 to x31 and registers 32 to 63 correspond to floating point registers f0 to f31. Since some instructions in later stages may generate hardcoded register numbers, these first 64 values are skipped when assigning output register numbers.

Similar to the method used to determine operand registers, the output register calculation uses another lookup table to determine the final type of calculation performed. This can be either the default register naming scheme for most expressions, a hard-coded output register such as the accumulator for return values, or zero for instructions that do not produce any results. A set of select statements is then used to obtain the final result.

Jump target

The jump target is zero for most instructions and it is only set for instructions performing jumps. In this case, for each of the branching nodes, it is set to the absolute instruction location to jump to, which is located in the data buffer used to pass information up the tree during the tree walk, similar to the instruction locations for these nodes. The only exception are jumps resulting from function calls and return statements, whose targets can be obtained from the function offset table. Both function call expression nodes and return statement nodes contain the function id of the node to be called, or the function the return statement is in, in the node data of the node respectively. The value stored in the node data of these nodes can be used as an index in to the function offset table to obtain the location to jump to.

Similar to operand registers and destination registers, a lookup table paired with select statements is used to determine the final result of the jump target field.

Immediate

The immediate field is set for instructions that have to load constants. Aside from the instructions generated by constants appearing in the program themselves, which have the constant to use located in their node data, this is also used to generate stack offsets for accessing local variables. In the case of local variables, each local variable has an unique per-function identifier identifying it, with the first symbol in the function having an identifier of zero. This identifier is used to calculate the actual stack offset by multiplying it by the size of a single element to obtain the offset in bytes. For any instructions without an immediate, the value it set to zero.

A similar lookup table and select statement combination as is used to calculate the values of the previous fields, is used to compute which of the above operations to perform.

Propagating data to the parent

To compute the data to propagate to the parent, another lookup table is used. In this case, the lookup table is indexed using the node type, resulting data type and instruction offset of the node. This lookup table contains an integer identifying the type of computation to perform to obtain the location where write the data to be propagated to the parent should be written.

All instructions either write to a fixed parent location which depends solely on the node the instruction came from, or do not write any results at all. The complete equation is given in Equation 4.5.

$$D_l(n) = P_i(n) * P_m + C_i(n) \tag{4.5}$$

In this equation, $P_i(n)$ gives the parent index of the current node, P_m is a constant describing the maximum number of slots per parent, which is three in our implementation and $C_i(n)$ is the child index of the current node. This effectively writes the data of the first child of a node to the parent's first slot and the data belonging second child of a node to the parent's second slot.

A lookup table is used to determine whether the equation has to be used and an invalid parent index is returned if no results have to be written. Since the index is propagated to a scatter operation and Futhark's implementation ignores invalid values, no results will be written if no results are expected.

The actual data to be propagated is always either the destination register, or an instruction location for the exceptions listed under Section 3.3.2. A single select statement is used to determine the result, which is then passed back together with the parent location as one of the results of the per-instruction compilation routine.

4.5. Optimization

After all instructions have been compiled, a number of significant optimizations can be performed. The initial instruction generation routine generates optimized code solely dependent on the node type and resulting data type of a node. In this stage, optimizations are performed over the generated set of instructions in order to produce more efficient code.

Before any optimization is performed, the instruction mask is allocated. The mask has one bit for every instruction, with bit n describing whether the n-th instruction should end up in the final executable. This mask is initially set to have all instructions enabled and later optimization stages can clear bits in this mask to disable instructions.

4.5.1. Dead expression elimination

Our current compiler performs one major optimization at this stage, dead expression optimization, which removes expressions that have no effect from the program.

First, it initializes a mask with the size of the array matching the total number of instructions. Since destination register numbers directly correspond to instruction locations, we can use this mask to keep track of both which registers to mark as unused and which instructions to remove. The initial array is set to be all zero, after which it is filled with fields to represent the initial usage of registers. To initialize the array, the compiler maps each instruction to its operand registers and then sets the element corresponding to each operand register it found to one. This is done by simply scattering the value of 1 into the buffer, using the operand registers as indices.

At this point, the compiler will know which registers are in-use by other instructions and which are not. Any instruction for which the destination register has a value of zero in this buffer will effectively be unused, as it represents an expression for which the value is not used by any other expression.

The compiler now enters a loop to find sub-expressions of these unused expressions. At the start of each loop, the compiler creates a copy of the usage data, which is used at the end of the loop to determine if any values have been changed. The detection of change is done by performing equality checks for all elements in this buffer in parallel, then reducing the results over the and-operator, thus comparing in parallel if all elements are equal. If all elements are equal, the compiler determines that all sub-expressions have been located, as no new sub-expressions have been marked as inactive and terminates the loop, finishing the optimization.

Within the loop, a similar operation is performed as during initialization. The compiler maps each instruction to its registers, but this time only doing so if the value in the usage data buffer of the instruction is zero. Subsequently found registers are set to zero using a scatter operation, marking the instructions generating them as unused. After this, the compiler performs a correction for side effects, setting the usage field for both the instruction with the side effect, as the instructions corresponding to its operands to true.

This operation is not performed during initialization, as the first iteration of the loop will correct any instructions with side-effects marked as unused during initialization and any operands that might have been marked as unused as a result of this are immediately set as used again. The results are scattered using the same operation as was used to set instructions as unused earlier.

After the loop finishes, the compiler has an array describing which instructions can be removed and which cannot. The compiler now converts this buffer into a bitmask and uses a bitwise or operation against the already existing instruction enabling mask to create a new one taking dead expression elimination into account, thereby finishing the optimization.

As an example of how dead expression elimination works, consider the set of instructions given in Listing 4.2. In this example, a register marked green denotes a register which is marked as in-use, whereas a register marked red is a register determined to be inactive. The stage denoted by this listing is the state after the initialization of the usage array, immediately before the loop starts executing. Note that only register `r66` is marked as inactive, since it does not appear as an operand to any other instructions.

Listing 4.2 An example set of instructions to optimize

```
LUI r64, 0
LUI r65, 1
ADD r66, r64, r65
LUI r67, 2
LUI r68, 3
ADD r69, r67, r68
STORE 0(sp), r69
```

After this initialization step, the loop starts executing. The first iteration will result in the instruction marking shown in Listing 4.3. Here we see that registers `r64` and `r65` have been marked as inactive, due to these being the operands to register `r66`. In the next iteration, nothing will change and only the first three registers will be marked as inactive, causing the loop to terminate. The instructions having registers `r64`, `r65` and `r66` will then be marked as inactive, since they are deemed part of a dead expression. Later during the instruction removal stage, described in Section 4.7, these instructions will be removed from the program.

Listing 4.3 An example set of instructions to optimize

```
LUI r64, 0
LUI r65, 1
ADD r66, r64, r65
LUI r67, 2
LUI r68, 3
ADD r69, r67, r68
STORE 0(sp), r69
```

4.6. Register Allocation

The next stage in the compilation process is register allocation. Register allocation is responsible for transforming the current infinite set of registers, which we will refer to as virtual registers in this section, to a finite set of registers matching those available on the target platform, which we will refer to as physical registers. Register allocation goes through multiple stages, starting with the initial lifetime analysis and register assignment, then moving on to stack space analysis. After that, the compiler determines if it needs to generate load and store instructions for spilled registers, before finally inserting stack frames to finish the procedure. Each of these components is implemented as a separate sub-stage, with the following subsections describing how each stage operates.

4.6.1. Lifetime analysis

To perform lifetime analysis, the compiler will have to visit each instruction of every function. Although separate functions can be analyzed in parallel, the instructions within a function themselves need to be analyzed sequentially, as the results of the analysis of previous instructions impact what registers are available for use for the next instruction. As a consequence, the lifetime analysis stage is a rather expensive stage in the compilation process. To mitigate the performance penalty of this stage, some optimizations have been implemented.

Initially, the lifetime analysis stage will set up the mask describing which registers are free and which are occupied. One mask per function is required, so the compiler allocates an array of masks and initializes each of these with a mask marking only registers the compiler is allowed to use as a target register as free. The RISC-V architecture assigns some registers a special meaning, such as a stack pointer, a base pointer, the function return address. The RISC-V architecture also defines register x0 to be a hard-wired zero constant, meaning that this register cannot be used for computation. Each of these registers are marked as occupied during initialization, preventing the algorithm from allocating these registers.

Secondly, the algorithm requires two tables to keep track of virtual and physical register stages. One table has one entry per virtual register and stores the physical register it is assigned to, together with a swap bit describing if the register should be moved to the stack after its value has been generated. This table is used as the result of the analysis and will at the end contain how a register is to be used. The second table is a table used to map physical registers to the virtual registers that currently occupy them. This second table is required to correctly mark virtual registers that need to be swapped out if their physical register has to be reused.

Once all tables have been initialized, the compiler starts looping through instructions and analyzes one instruction per function in parallel. This function will allocate a physical register for the instruction's destination register, deallocate operand registers and update the lifetime masks to reflect the new register state. If no registers are available, this function will determine a physical register it wants to reuse and return this physical register number.

Once the per-instruction analysis finishes executing, the lifetime analysis will take the set

of physical registers it determined should be swapped out and perform a reverse lookup in the physical register table to obtain the virtual registers currently occupying those registers. It will then set their swap bits to mark that they should be moved to the stack. Afterwards, the lifetime analysis updates both tables by scattering the new data returned from the per-instruction analysis function into the tables, after which it moves on to the next instruction, or stops if no further instructions are available.

Per-instruction analysis

Per-instruction analysis takes an instruction and the current register state of a function and determines which physical registers should be used by that instruction. To achieve this, it makes use of the three restrictions imposed on the register set described in Section 3.3.2.

First, it deallocates the physical registers currently in use by its two operands. Even though a virtual register may be read from more than once, the assumptions guarantee that no new registers will be allocated before the final usage of this virtual register. So even if it is marked as inactive the moment it is first read from, this will not have any effect on the actual allocation itself, as the physical register the operand occupies will not be reused until the virtual register is no longer used. If the swap bit of the operand is set, this means that the operand is not currently in a physical register, causing the allocator to assign a predetermined register to the operand. The per-instruction analysis will add this register to the set of physical registers to be swapped out, which signals to the main lifetime analysis loop that the swap bit of the virtual register currently occupying the physical register will be set. Different registers are used for the two operands an instruction may have to prevent overwriting the first operand when the second operand is loaded if both operands are currently swapped out.

After both operand registers are deallocated, it now allocates a register for the destination register. To do this, the allocator will first determine if an integer register or a floating-point register is required. In the RISC-V architecture, the last seven bits of the instruction opcode signify the instruction class. For any instructions requiring floating-point registers, this class is set to the floating-point instruction set class, allowing us to easily detect if a floating-point register has to be used by simply comparing the final seven bits of the opcode to the constant specifying the floating-point instruction class. The bits in the lifetime mask corresponding to the other register type are set to occupied using a bitmask, which then forces the compiler to allocate a register in the right set of registers.

To allocate a physical register, the compiler tries to find the first bit in the lifetime mask that is not set, as this would now indicate a valid non-occupied register. To do this, a bit count is performed to find the first zero bit in the lifetime mask. If no such bit was available, this means all valid registers are occupied and the compiler will take one predetermined register and swap it out. Similar to how swapping works for operand registers, the reused physical register is then added to the swapped register set so the swap bit of the virtual register currently occupying it can be set by the main lifetime analysis loop. After allocating the register, the lifetime mask is updated to reflect the now occupied register.

A special exception is formed by instructions requiring specific predetermined registers.

In this case, no allocation is performed and the lifetime mask is simply updated to set the bit corresponding to the requested register. If the register was previously occupied, the register is added to the set of swapped physical registers. Virtual register numbers only start from 64 to allow the instruction generation phase to use the lower constants to request predetermined registers, with values from 0 to 31 signaling integer registers x0 to x31 and constants 32 to 64 meaning floating-point registers f0 to f31.

The final exception to the rule are call instructions. These instructions need to mark any scratch register in the calling convention as free and possibly swap any virtual registers currently stored in these scratch registers. To detect a call, the compiler assumes that any jump instruction that jumps to a location outside of the function boundary is a call. The first instruction of the function is marked as outside of the function boundary to correctly handle recursive calls. Since each function starts with a function prologue that is only executed once per function call, no other jump instructions can jump to the first instruction of a function. The compiler checks for jumps by once again making use of the fact that the last seven bits of the instruction opcode on RISC-V signify the instruction class and this time verifies that it matches the class representing jump instructions. It then checks if the jump target field of the instruction is outside of the function range given by the function offset table to determine whether it has encountered a call instruction. If it found a call instruction, it will simply add all scratch registers to the swapped physical register set and clear any bits in the lifetime mask corresponding to these bits. Since arguments are set up with instructions whose target is a predetermined register, the entries in the physical register table for the arguments will be empty, meaning the argument registers are thus consumed by the function call, clearing the registers they were occupying as a result.

Finally, in case an instruction is not active according to the mask generated by the optimizer, the per-instruction analysis performs no action, effectively ignoring the instruction.

4.6.2. Stack space analysis

To perform stack space analysis, the compiler will have to determine a unique stack offset for every spilled register on a per-function basis. To do this in parallel, the compiler makes use of a segmented scan operation. A segmented scan performs the function of a scan operation within an array of elements, however it also takes a second array describing where various segments start and the results of the scan are reset at the start of each segment. A segmented scan over the addition operator leads to a segmented prefix sum, which allows us to perform a prefix sum over various sub-sequences of an array in parallel. Rotating these elements by one position afterwards yields us the results of an exclusive prefix sum.

By making use of the function offset table and the fact that virtual register numbers correspond to instruction locations, the compiler generates the segment descriptor array by simply scattering the start of segment indicator using the function offset table's start of function element as its index. This allows us to split the virtual register table into parts where each part contains only the virtual registers present within one function. We then map each entry in the virtual register table to either the size of the register if its swap bit is set, or zero if it is not and perform a segmented scan over this array. The result is an

array where each entry in the array corresponding to a swapped register contains a unique offset, which starts from zero at the start of each function. To each element in this array, we then add the size of the stack required to hold the local variables of the function the corresponding virtual register belongs to. This gives us usable stack offsets for storing the spill locations.

By taking the last element of each function, which can be found by taking the elements immediately before the segment starts or the end of the array, we now have a the stack size each function requires to hold all its local variables and spilled registers. Thus completing the stack space analysis sub-stage of register allocation.

4.6.3. Determining spill instructions

The next stage of the process is determining the number of load and store operations required to correctly swap out spilled registers. Using the rules given in Section 3.5.3, we end up with the Equation 4.6.

$$N_i(i) = 1 + S_{op1}(i) + S_{op2}(i) + S_{dest}(i) \tag{4.6}$$

In this equation, $S_x(i)$ returns 1 if operand x of instruction i has its spill flag set and 0 otherwise. In effect, this means we add one instruction to each existing instruction for each virtual register that has its spill flag set. Both reading a value from the stack as well as storing a value to the stack can be done in a single instruction on the RISC-V architecture, therefore these counts are all set to 1.

The actual insertion of the instructions is not performed until the instruction removal stage found in Section 4.7 in order to re-use the recalculation of instruction locations performed during that stage.

4.6.4. Inserting stack frames

To finish register allocation, the instructions to set up the stack frames are inserted. Because room for these instructions has already been allocated during the instruction counting stage, we only need to overwrite the previously invalid instructions with the correct instructions to generate the stack frame. Note that these instructions are always located at the same location relative to the start and end of the function. So, we use the function offset table to determine where each function starts and ends and map each function id to the instructions required to set up the stack frame belonging to that function and the corresponding instruction location. Since the stack sizes for each function were already determined during the stack space analysis, this stage only has to insert the computed values as immediate values into the stack pointer subtraction or addition instructions required for the prologue and epilogue respectively. Finally, we scatter these instructions into their correct locations for all functions at once, inserting the stack frames.

4.7. Instruction Removal

The compiler now moves on to instruction removal, where it removes any instructions optimized away during earlier stages from the array of instructions. This stage is also responsible for inserting the spill load and store instructions the register allocation determined were necessary.

Our implementation first performs an exclusive prefix sum to determine the new instruction locations. The exclusive prefix sum is performed over the array generated by performing Equation 4.6 over every instruction in the program. This array was already created during the third stage of register allocation, see Section 4.6.3. One change is made to this array however: every entry corresponding to an instruction for which the mask describing whether an instruction is enabled or not says the instruction should not be enabled, is set to zero so it is removed from the program.

Next, the compiler determines the actual instructions and instruction locations for each spill load and store instruction it needs to insert. If an operand to an instruction has the swap bit set, it will need to load the value from memory and if the destination register has the swap bit set, it will need to write the result of the operation to memory. The stack offsets for all registers are obtained from the array of stack offsets generated during the register allocation's stack space analysis subroutine, described in Section 4.6.2. To determine the final locations of each instruction, the compiler uses the set of equations found in Equation 4.7.

$$\begin{aligned} L_{op1}(i) &= B(i) \\ L_{op2}(i) &= S_{op1}(i) + L_{op1}(i) \\ L_{instr}(i) &= S_{op2}(i) + L_{op2}(i) \\ L_{dst}(i) &= L_{instr}(i) + 1 \end{aligned} \tag{4.7}$$

In this equation, $L_x(i)$ is the instruction location for element x of instruction i, where x may either be loading an operand, the main instruction, or storing the result and $S_x(i)$ returns 1 if the swap bit for component x of instruction i is set. These functions set the location of each instruction to be immediately after the previous instruction, taking into account the fact that there may not be a previous instruction. If an operand or destination register does not need to be swapped, the instruction location corresponding to that instruction is set to an invalid value by the compiler.

Since all instructions in the program need to be mapped in this stage to determine their new locations, this stage makes a slight change to the instruction encoding. At this point, the register fields in the instruction data structure still encode virtual register numbers, however since register allocation has already finished, the virtual register table can be used to resolve these virtual registers to physical registers. All registers appearing in instructions in the program are thus resolved to their physical register numbers, which are now stored in the register field of the instruction.

Afterwards, the compiler scatters all newly generated instructions into the newly allocated instruction buffer, removing any instructions that have been optimized away and insert-

ing any spill operations if required. To determine whether an operation requires a spill instruction, the compiler consults the virtual register table. If the swap bit is set for a virtual register used as an operand, a load instruction is generated. If the swap bit is set for the destination register, a store instruction is generated.

4.8. Jump Linking

After instruction removal has been performed, the compiler needs to fix any jumps that have been invalidated due to the removal of instructions during the previous stage and possibly insert extra instructions to calculate jump targets if the jump target is too far away to store in an immediate on the target platform.

The compiler first tries to allocate room for jump instructions requiring more than one instruction by performing an exclusive prefix sum similar to the one performed during the instruction removal stage. Each instruction is mapped to a size, that is either one for normal instructions and jump instructions jumping to targets that are close enough to fit in a single instruction, or two for jump instructions that require an extra instruction to setup the jump target address. A subsequent exclusive prefix sum is performed to compute new instruction locations. To determine whether an instruction is a jump, the compiler performs a similar operation to the one performed during the lifetime analysis stage of the register allocation stage, described in Section 4.6.1, exploiting the way opcodes are encoded on the RISC-V platform. By checking the lower 7 bits of the opcode, the compiler can determine whether an instruction is a jump type instruction and thus handle it accordingly.

To determine if an extra instruction is required, the compiler uses the new instruction locations computed during the instruction removal stage. However, since additional instructions can still be inserted due to other jumps that might require more than one instruction, the compiler cannot calculate the exact results in parallel. Instead, it assumes the worst case scenario, in which case every instruction between the current instruction and the jump target takes up two instructions. If the target is within this threshold, the compiler will generate a single jump instruction and otherwise it will generate a two instruction sequence.

After recalculating the new instruction locations, the instructions are immediately scattered to their new correct locations. After this, the compiler will perform a substitution for any jump instruction. Each jump instruction is mapped to an array of two pairs of instruction locations and instruction in parallel. If a jump has only one instruction, only the location for the first instruction will be valid and the instruction itself will be the new jump instruction. If two instructions need to be generated, the two pairs will represent the locations and instructions for the two generated instructions.

During this mapping, all jump targets are updated to point to their correct location. This is done in two steps. Since the jump targets are absolute addresses based on the instruction locations before the instruction removal stage, they can also be used as indices into the instruction location array to find the new location of those instructions after the instruction removal stage. Thus all jump targets are first resolved against the instruction location array generated by the instruction removal stage. These new locations can then

be used as indices into the instruction location array generated by the exclusive prefix sum performed at the start of the jump linking stage to obtain the final locations of the original instructions after optimization. This two-step process is required as the jump linking stage's instruction location buffer does not contain entries for every instruction that appeared during the instruction generation stage, as some of them may have been removed during the instruction removal stage.

The same operations are used to update the function offset table. Function start and end locations are resolved against the two instruction location buffers to update the contents of the function offset table.

Finally, all jump targets are translated from absolute locations to relative addresses. On the RISC-V architecture, jumps are generally encoded as PC-relative jumps, thus a translation needs to be performed. This is done by simply subtracting the jump instruction's location from its jump target. The resulting offset is then encoded in accordance with the instruction encoding of the RISC-V architecture and stored as an immediate. The jump target field is set to zero, as it is no longer used.

4.9. Postprocessing

The final step of the compilation process takes all the data generated by previous stages and converts this into encoded instructions. At this point, the opcode field of the instruction structure contains the instruction opcode and immediate and jump targets have already been converted to immediate values. The register fields for the instructions are still set to zero within the opcode itself, but the physical registers are present in the three register fields in the instruction data structure. Therefore, a bitwise or operation is used to merge the opcode and register fields of all instructions, finalizing the encoded instruction. This array of opcodes is now returned as the result of the compilation, together with the function offset table. This finalizes the compilation process.

The final result of this process is an array of RISC-V instructions and the function offset table, containing the start address and size of each function in this array of instructions. Because all code generated by our implementation is position independent, the array of instructions can be placed at any address in memory without requiring the instructions to be modified. The function offset table can be used to add symbol definitions for each function in the program. Providing the CPU with a relocatable set of instructions and a table containing the location and size of all functions within the array of instructions provides it with sufficient information to encode the generated instructions in most common object file formats.

5. Experiments

To evaluate the effectiveness of the parallel implementation and to determine the performance characteristics of our implementation, three sets of experiments have been performed. These experiments are designed to measure the scaling of the runtime of the program under different input abstract syntax trees. Each test case measures the runtime of the various stages of the compiler when it is provided with a valid abstract syntax tree. The size and shape of the provided abstract syntax trees were varied among test cases to measure the performance of the compiler under various scenarios. The runtime of each stage of the compilation process was measured. The register allocation and instruction removal stages were merged since a significant part of the instruction removal stage is the insertion of the load and store instructions generated by the register allocation stages.

Three main experiments have been performed. The first experiment was designed to show the scaling of our implementation when the size of the abstract syntax tree is increased. The second experiment was used to determine the performance of our implementation when the length of functions was varied for a source file of constant size. The third experiment was designed to measure how our implementation performs when the shape of the syntax tree is varied.

All experiments were performed on the same machine. The properties of the machine the experiments were performed on can be found in Table 5.1. A full description on how to compile our implementation and run our experiments can be found in Appendix B. For all experiments, the runtime of each separate stage of the compiler was measured. For these measurements, we decided not to include the overhead of the initial memory transfer of the abstract syntax tree to the GPU in our measurements, as in our full compiler implementation this tree would be created on the GPU by the frontend.

GPU Type	NVIDIA RTX 3090
GPU Memory	24GB
CPU Type	Intel Xeon Silver 4214
RAM	256GB

Table 5.1. Hardware properties of the machine used for our experiments.

5.1. Abstract Syntax Tree Generation

To create the test cases used for our experiments, various abstract syntax trees had to be generated in order to provide the compiler with a valid input. Since our compiler has a custom input language, no preexisting code was available to use as input dataset. Instead, for our experiments, random abstract syntax trees were used. These random abstract syntax trees represent the syntax trees of an imperative, procedural programming language. These trees were generated by starting with the root node of the syntax tree to generate and then recursively selecting valid children randomly until the tree is complete. A description of how valid syntax trees for our compiler are formed, is given in Appendix A.

Our test case generation program allows us to specify certain properties of the syntax tree to generate, such as the maximum width of the syntax tree, the maximum height of the syntax tree, the number of functions to generate and the maximum length of variable-length nodes such as statement lists and function argument lists. By modifying these parameters it is possible to create abstract syntax trees of various sizes and shapes, allowing us to effectively test our implementation.

The syntax trees generated by our test case generation program are guaranteed to be semantically correct. All generated programs should compile without failure. However, the generated syntax trees may not represent valid programs, as it might generate a set of operations that would cause the compiled program to crash if executed. For example, the generated code may attempt integer division by zero or trigger infinite recursion which would result in a stack overflow on the RISC-V architecture. The generated programs may also contain infinite loops, which would cause the generated programs to hang indefinitely if executed. However, since these programs can still be compiled successfully by our compiler, they are still suitable as test cases used to benchmark the various stages of the compilation process.

5.2. Source File Size

The first experiment performed measures the performance of our compiler when the total source file size and thus the number of nodes in the abstract syntax tree, is increased. The intent of this experiment is to determine how our implementation scales under the input size. The source files in this experiment are designed to represent realistic source files of various sizes.

To obtain an abstract syntax tree resembling a real source file as closely as possible, we decided to generate syntax trees with a fixed depth of nineteen nodes and to increase the width and number of functions when increasing the source file size. Real source files tend to be split up into multiple functions, with a large source file having more functions than a smaller source file, instead of having larger functions overall. The depth of nineteen nodes was chosen to allow a decent number of nested expressions and statements to appear in the source code, while not creating so many nested blocks that the source file would become unrealistic. This way we can measure the performance of the compiler using random source

Source File Size	Nodes	Width	Height	Functions	Maximum Function Nodes
5kB	1155	234	19	5	397
10kB	3078	697	18	9	667
100kB	25793	5454	19	80	846
500kB	128321	27053	19	421	1108
1MB	275852	58642	19	895	1006
10MB	2550855	541224	19	8387	1295
50MB	12301146	2619673	19	40112	1296

Table 5.2. Properties of the abstract syntax trees for the source file size experiment.

Test Case	Minimum Runtime (ms)	Maximum Runtime (ms)	Standard Deviation (ms)
5kB	52.238	53.444	0.452
10kB	86.748	89.757	0.582
100kB	125.37	142.89	4.128
500kB	175.79	192.90	3.638
1MB	164.20	193.60	5.141
10MB	410.03	419.91	2.541
50MB	1342.2	1364.5	4.804

Table 5.3. Minimum and maximum runtimes and the standard deviation on the runtime for the source file size experiment.

files that most closely resemble real programs.

A total of seven different randomly generated abstract syntax trees were tested, each varying in the total number of nodes in the abstract syntax tree. An overview of the properties of the files can be found in Table 5.2, listing the number of nodes, the width and height of the abstract syntax tree, the number of functions appearing in the source file and the number of nodes of the largest function. This table also contains the source file size, which is the size of the source file represented by the abstract syntax tree. Each of the seven test cases was compiled 31 times using our compiler and the runtime of each substage of the compiler was measured and the results were averaged to obtain the final result.

A graph containing the total runtime of the GPU stages is given in Figure 5.1 and Table 5.3 contains the minimum and maximum runtime and the standard deviation on the runtime in milliseconds for each test case. In Figure 5.1, the number of nodes in the syntax tree is plotted against the total runtime of the compiler. From this figure, we can conclude that even as the number of nodes in the abstract syntax tree increases significantly, the total runtime appears to increase sub-linearly. This sub-linear increase is especially noticeable for source files with sizes under 1MB. On traditional compilers we would expect to see a linear increase in runtime when the source file size increases, which illustrates the performance of our GPU implementation. However, to more accurately obtain information on what stages exactly are responsible for this increased runtime, we measured the total runtime for each stage separately, the results of this experiment can be found in Figure 5.2. From this graph, we can conclude that the combined register

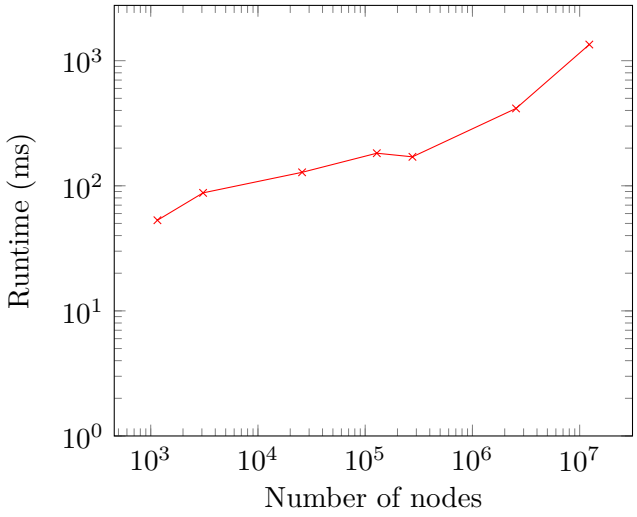


Figure 5.1. The total runtime of the source file size test cases plotted against the number of nodes.

allocation and instruction removal stage is the stage that has by far the most impact on the performance. For most test cases, the register allocation and instruction removal stage takes up nearly all of the runtime of the program.

The register allocation stage starts with a largely sequential lifetime analysis substage. This substage loops through all instructions in the function and performs an initial greedy assignment of physical registers to virtual registers. Since this substage can only be run in parallel for each function and has to sequentially process each instruction of a function, it can take a significant amount of time when functions grow larger. Furthermore, since it can only be run in parallel for each function instead of for each instruction or node as is applicable for all other stages, it cannot be parallelized as well as the other stages, leading to a significant runtime overhead compared to the other stages of the program. The next experiment in this chapter shows the relation between function length and the overhead of the register allocation stage.

5.3. Function Length

The second experiment was performed to measure the impact of the length of the functions on the compilation process. One significant stage of the compilation process, register allocation, starts with a lifetime analysis substage which goes through each instruction in a function sequentially. As the size of a function grows, so does the number of instructions it generates, which would increase the runtime of the register allocation stage. To determine the relation between function length and the performance of this stage, an experiment was performed.

A total of eight source files were generated. Each abstract syntax tree has a similar depth and width and varies only in the number of functions present in the source file. If more functions are present in a source file without the size of the source file changing, the length of those functions must go up. Each of the eight source files thus has functions of increasing

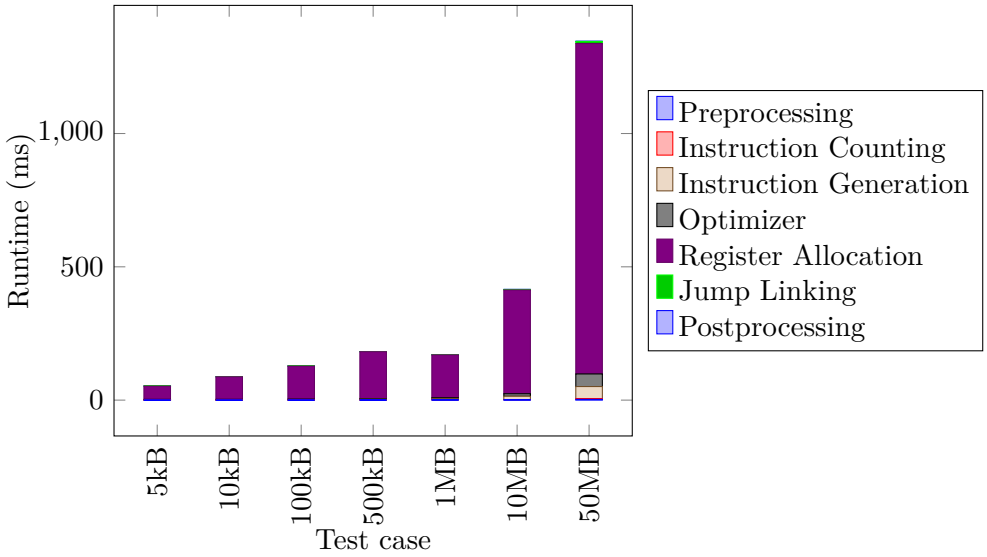


Figure 5.2. A breakdown of the runtime of the various major substages of the compiler for the source file size experiment.

size. An overview of the generated syntax trees can be found in Table 5.4. Similar to the previous experiment, each test case was run 31 times and results were averaged to obtain the final result.

The results of the experiment can be found in Figure 5.3, with an overview of the minimum runtime, maximum runtime and standard deviation on the runtime in milliseconds found in Table 5.5. While we can see that the runtime of most stages remains nearly identical, there is a significant increase in the time spent on the register allocation stage when the function length is increased beyond a certain point. The runtime overhead even seems to increase if the total number of nodes in the program decreases slightly, meaning the only factor that could cause this significant increase in runtime is the length of the functions themselves. This performance impact is significant enough that it becomes the main bottleneck of the compilation process once functions become large enough. Future implementations should attempt to find alternative methods of register allocation if they will encounter significantly large functions, as the current implementation is ill-suited to compiling large functions. One proposal for future implementation can be found in Section 6.1.1, where a possible method of parallel register allocation within functions is proposed for statement-based

Test case	Nodes	Width	Height	Functions	Maximum Function Nodes
1	307141	77058	9	25118	53
2	228263	61668	9	12495	79
3	199753	54731	9	6181	90
4	173856	41508	9	2513	164
5	164527	38851	9	1539	214
6	123281	31352	9	497	402
7	121211	31849	9	251	657
8	141181	37605	9	100	1740

Table 5.4. Properties of the abstract syntax trees for function length experiment.

Test Case	Minimum Runtime (ms)	Maximum Runtime (ms)	Standard Deviation (ms)
1	44.030	45.134	0.498
2	37.112	40.007	0.649
3	32.998	34.962	0.530
4	38.122	42.801	0.900
5	41.598	43.011	0.433
6	67.680	68.961	0.442
7	105.87	109.50	0.703
8	216.65	230.40	4.552

Table 5.5. Minimum and maximum runtimes and the standard deviation on the runtime for the source file size experiment.

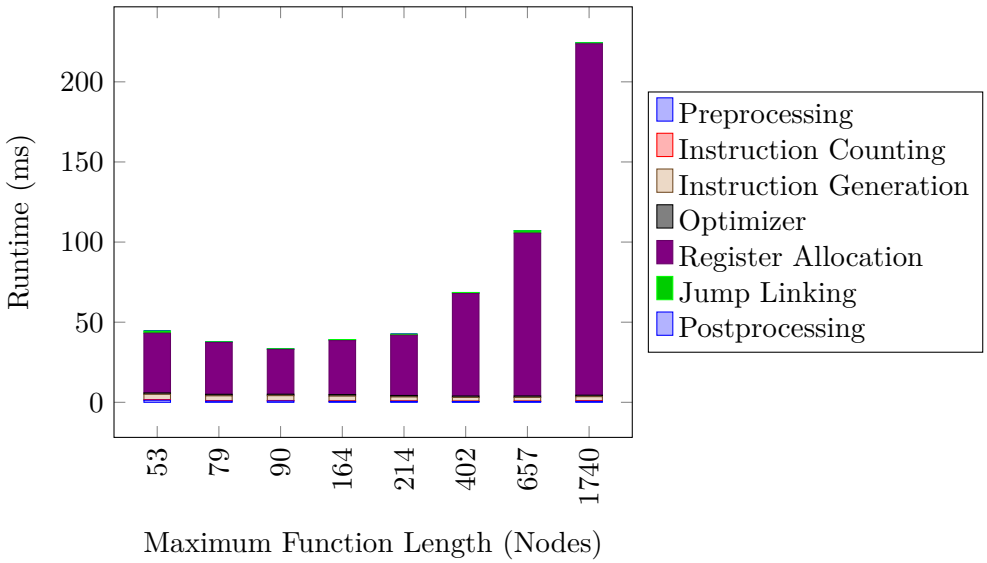


Figure 5.3. A breakdown of the runtime of the various major substages of the compiler for the function length experiment.

imperative languages.

5.4. Abstract Syntax Tree Shape

The final experiment attempts to classify the performance of the implementation when the shape of the abstract syntax tree is changed. The tree walk performed in the instruction generation stage can only run nodes on the same layer of the syntax tree in parallel. If a tree becomes too narrow, there might not be enough nodes for the tree walk to run in parallel, which could cause an increase in runtime for this stage.

To measure this impact, six source files have been generated. Each source file varies in tree width and height, but has a similar number of nodes. The properties of these source files can be found in Table 5.6. Due to the randomized nature of the source file generation,

Test case	Nodes	Width	Height	Functions	Maximum Function Nodes
1	939825	247606	9	25089	110
2	825604	156257	19	40014	1190
3	809010	144621	34	16474	115015
4	669149	124725	36	25087	243973
5	618107	114432	42	24718	218264
6	564276	105432	49	25191	185176

Table 5.6. Properties of the abstract syntax trees for the abstract syntax tree shape experiment.

the number of nodes are not exactly the same, they are however similar enough that the experiment to indicate whether the shape of the abstract syntax tree has an effect on the runtime of the implementation can be performed.

As the depth of a tree increases, so do the sizes of the functions in that tree, as function declaration nodes are always located in the second layer of the abstract syntax tree. For deeper trees, the runtime of the register allocation stage would thus increase significantly, as was also found in the experiment described in the previous section. To see the influence of abstract syntax tree shape on the instruction generation stage, measuring the total compiler runtime as was done for previous experiments would not give us any usable measurements with which we can determine the effect of the shape of the abstract syntax tree on the runtime of the various substages. Measuring the total runtime would only show a significant increase in runtime due to the increase in function length causing the register allocation stage to perform significantly worse. Instead, a detailed breakdown per stage was generated. These results can be found in Table 5.7, which contains the runtimes of the various substages for each of the test cases run for this experiment in milliseconds. Each test case was run 31 times and results were averaged in order to obtain the final results. A table containing the maximum runtime, minimum runtime and the standard deviation on the total runtime in milliseconds can be found in Table 5.8.

From Table 5.7 we can see that the register allocation stage's runtime increases significantly for these files and is mostly dependent on the number of nodes in the largest functions. We can also see that most of the other stages seem to more closely follow the total number of nodes in the abstract syntax tree as an indicator for their runtime. Two stages stand out among these results, the instruction generation stage and the optimization stage, whose runtimes increase for the deeper trees.

The runtimes of the instruction generation and optimization stages have been plotted in Figure 5.4. For both stages, the runtime increases when the depth of the abstract syntax tree increases. For the instruction generation stage, this can be explained due the the design of the tree walk. The tree walk performed during the instruction generation stage can only run instructions at the same layer of the tree in parallel. If a tree becomes deeper and narrower, fewer nodes can be processed before the tree walk needs to synchronize all threads in order to prepare for the next iteration of the tree walk. This imposes an overhead on deeper trees.

For the optimization stage, the increase in runtime can be explained due to the algorithms performed for dead expression removal, which was the only optimization performed during

Test Case	Total (ms)	Preprocessing (ms)	Instruction Counting (ms)	Instruction Generation (ms)
1	78.87	1.401	0.376	5.45
2	957.584	1.37	0.362	6.114
3	45568.068	1.35	0.363	8.014
4	121399.471	1.397	0.356	7.706
5	110620.601	1.364	0.349	8.057
6	97193.301	1.368	0.4	8.721
Test Case	**Optimization (ms)**	**Register Allocation (ms)**	**Jump Linking (ms)**	**Postprocessing (ms)**
1	1.959	68.639	0.983	0.063
2	3.938	944.869	0.874	0.057
3	6.995	45550.568	0.729	0.05
4	7.544	121381.801	0.623	0.044
5	6.634	110603.524	0.629	0.044
6	7.008	97175.172	0.591	0.041

Table 5.7. The runtime for each of the substages for the abstract syntax tree shape experiment.

Test Case	Minimum Runtime (ms)	Maximum Runtime (ms)	Standard Deviation (ms)
1	78.493	79.692	0.475
2	955.48	959.49	1.001
3	45490.1	45659.5	43.64
4	121274	121513	55.31
5	110488	110720	47.59
6	96914.9	97273.1	66.17

Table 5.8. Minimum and maximum runtimes and the standard deviation on the runtime for the source file size experiment.

this stage. The dead expression removal algorithm performs a loop until no new virtual registers can be marked as unused. At each iteration, the dead expression elimination algorithm will mark any sub-expressions of an expression that was marked as unused in an earlier iteration as invalid and the first expressions to be marked as unused are expressions that are not used as an operand to any other expression. This corresponds to performing a top-down walk through the abstract syntax tree starting at each top-level expression and marking any expression node that is the child of an unused expression as unused. If the abstract syntax tree gets deeper, the depth of expressions will also get deeper, which would thus increase the number of iterations the optimizer has to perform before it has observed all expressions, increasing the runtime for deeper trees.

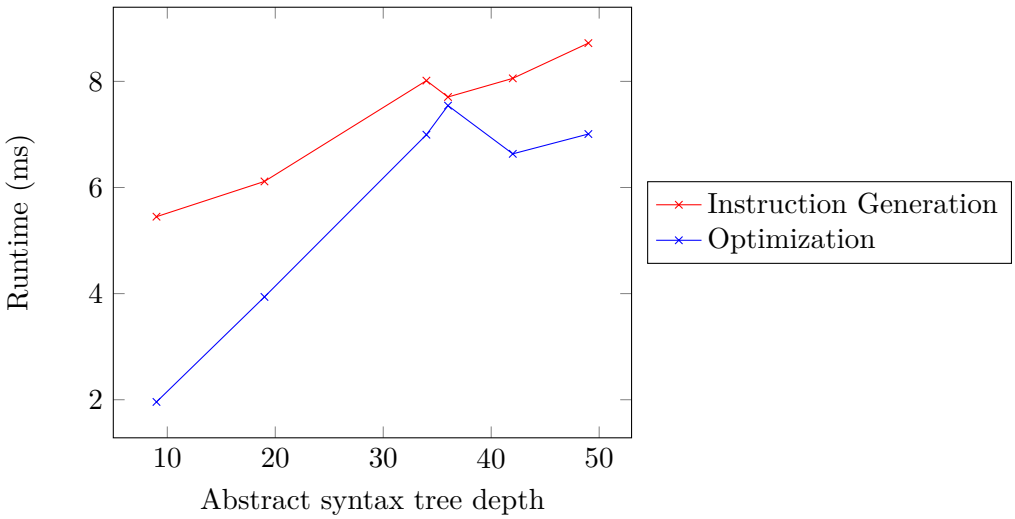


Figure 5.4. The runtime of the instruction generation and optimization stages plotted against the abstract syntax tree depth.

6. Conclusions

In this thesis we proposed a novel method for code generation on GPU architectures, which processes an abstract syntax tree in parallel contrary to sequential processing as is done by traditional compilers. To accomplish this, we proposed a design consisting of a set of new stages capable of transforming an abstract syntax tree into a set of instructions using only parallel GPU primitives. We then implemented our proposed design in the Pareas compiler, which was shown to be capable of compiling an imperative programming language into a list of RISC-V instructions, demonstrating the feasibility of our design.

We measured the performance of our implementation and found that our implementation works the best for wide, shallow abstract syntax trees, which are similar to real-world source files, but performs significantly worse on deep, narrow trees. We found that for these files a significant part of the runtime was the register allocation stage, which has a large overhead for the long functions present in these source files. Since the implementation performs best on abstract syntax trees that are similar to real-world source files, we believe this novel method of compilation can be effective for speeding up the compilation of large projects.

6.1. Future Work

A number of additions and improvements to the current implementation can still be made to further optimize the performance and increase the applicability of the compiler. This section lists a number of significant improvements that can be made to the implementation.

6.1.1. Register Allocation Optimization

As found during the experiments performed in Chapter 5, the register allocation stage of the compiler takes up a significant portion of the total runtime. Furthermore, as described in Section 5.3, the length of a function has a significant impact on the performance of the register allocation stage. Since the current implementation of the register allocation stage goes through functions on an instruction-per-instruction basis, only executing separate functions in parallel, the length of a function has a significant impact on the performance of this stage.

The current register allocator is still rather generic and does not make use of some properties of the language being compiled. By using language-specific properties the register allocator could be significantly optimized. One possible implementation might make use of

Expression Start	Expression Size
0	6
6	6

Table 6.1. Instruction offset and size for the example function

the fact that in statement-based languages the register state at the end of a full expression will return to its default state of not having any registers allocated for calculations.

We could use this property to optimize the register allocation stage. By having the instruction counting stage keep track of where full expressions begin and end in a table similar to the current function offset table, we could instead partition instructions on a per-expression basis instead of a per-function basis, allowing longer functions to be split up into smaller parts that can be allocated in parallel.

To then determine the set of registers used by a function, the masks for each expression can be combined using a bitwise or operation, thus yielding a set of all registers used by the various expressions of a function, which would allow the later stages of the register allocator to generate stack frames using the same method the current implementation uses.

As an example of this proposed modification, suppose we have a program given by Listing 6.1. In our current implementation, register allocation for all instructions in this function would be performed sequentially. However, using the newly proposed method, the instructions can be partitioned into two separate full expressions, which correspond to the two statements of the function. The compiler can then use the instruction locations of these statement nodes to determine the start and size of these full expressions, resulting in the table found in Table 6.1. Finally, this would result in the instructions being partitioned as seen in Listing 6.2, where red instructions correspond to the instructions belonging to the first full expression and blue instructions correspond to the instructions belonging to the second full expressions, which can then be allocated concurrently.

Listing 6.1 An example function

```
function f() {
    a : int = 2 + 3;
    b : int = 1 - 4;
}
```

Listing 6.2 The instructions of the example function partitioned into two full expressions

```
LUI r64, 0
ADDI r65, r64, 2
LUI r66, 0
ADDI r67, r66, 3
ADD r68, r65, r67
STORE a(sp), r68
LUI r70, 0
ADDI r71, r70, 1
LUI r72, 0
ADDI r73, r72, 4
```

```
SUB r74, r71, r73
STORE b(sp), r74
```

Further research can also be done into implementing alternative forms of register allocation, which do not rely on a greedy approach of register allocation.

6.1.2. Basic-Block Level Optimization

Currently, our compiler only performs a small number of local optimizations. However, modern compilers support a wide variety of increasingly complex optimizations. Future implementations could examine the possibility of efficiently performing these optimizations on a GPU architecture.

Many optimizations take place within a basic block, a set of instructions always executed as a single block, with no transfers of control flow from within the block, or from outside the block into the block. Control flow may only be transferred into the first instruction of a basic block and control flow may only be transferred from the last instruction of a basic block. By inserting an extra stage after the instruction generation stage, these basic blocks could be retrieved efficiently.

After the instruction generation stage, any control transfer instruction will have its jump target field set to the target of the jump, which can be used to identify the start of any basic block targeted by a jump instruction. Jump instructions themselves form the end of basic blocks and adding a value of one to their location should give the start of the basic block immediately following the basic block ending with the jump instruction. By merging this set of of starting locations with the list of jump target locations, the start of the program and the end of the program and performing a radix sort to sort all targets by their instruction location, we thus obtain a sorted list of all instruction locations where basic blocks would start. The size of each basic block can then be determined by the instruction location of the next basic block, which should be the next element in the array of basic block start locations.

For an example of this proposed method, we can use the instruction found in Listing 6.3. In this example, one jump is performed from the third instruction in the program to the sixth instruction in the program. The program itself would thus contain three basic blocks, one from the start of the program to the first jump, one from the instruction immediately after the jump until the fifth instruction in the program and one containing only the sixth instruction.

Our proposed method of finding basic blocks would first yield an array containing all jump targets, which would be the set 6. It would then create the set of all instruction locations containing jump instructions and add one to them, this would result in the set 3, as the only jump instruction in this program is located at location 2. The final set would contain only the start and end locations of the program, which would be the set $0, 7$ in this instance. Combining the three sets and sorting them would give us the set $0, 3, 6, 7$, which corresponds to the basic blocks of this program.

Listing 6.3 An example set of instructions for basic block analysis

```
LUI r64, 1
ADDI r65, r64, 2
JAL x0, 6
LUI r67, 3
ADDI r68, r67, 4
LUI r69, 5
```

Further research could then be done to determine whether basic-block level optimization can effectively performed using this table of basic blocks. One possible implementation would be to then run a sequential optimization routine, similar to the lifetime analysis substage of our register allocation stage, which can be run for all basic blocks in parallel. However, for certain optimizations it might be possible to find a more efficient implementation.

6.1.3. Tree Walk Scheduling

The current implementation of our instruction generation stages performs a tree walk that schedules nodes location on the same level of the syntax tree to be compiled at the same time. While this works well for most programs, where the abstract syntax tree tends to be wide and relatively shallow, this does mean that for cases where the syntax tree is very deep and narrow the implementation will not be able to efficiently schedule many nodes to be compiled in parallel.

One possible method to resolve this issue is to split out the assignment of virtual registers from the rest of the instruction generation process. The main dependency in the tree walk preventing nodes at different layers from being executed in parallel is the requirement of parent nodes that the operand register numbers of their child nodes are known before they begin their compilation process. Instead, we could delay the generation of operand registers until after all nodes have been compiled once.

In this case, there is no dependency between parent and child nodes for the majority of nodes in the syntax tree, the only exception being the branching nodes who need to know the instruction locations of their children to determine the locations of their own instructions. A second pass solely dedicated to determining operand registers could then be used to fill in the operand register fields of all instructions by using the target register numbers generated during the first pass. This would allow both passes to schedule parent and child nodes at the same time for most node types, allowing most deep and narrow trees to still be compiled efficiently.

Research would need to be done into an effective scheduling method which can take these branching nodes into account and to see if the alternative tree walk algorithm would be as efficient for wide trees as the current tree walk algorithm.

Bibliography

[Voe21] R Voetter. "Parallel Lexing, Parsing and Semantic Analysis on the GPU". MSc Thesis. Leiden University, 2021.

[Aho+06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006. ISBN: 0321486811.

[Hen+17] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. "Futhark: Purely Functional GPU-programming with Nested Parallelism and In-place Array Updates". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. Barcelona, Spain: ACM, 2017, pp. 556–571. ISBN: 978-1-4503-4988-8. DOI: 10.1145/3062341.3062354. URL: http://doi.acm.org/10.1145/3062341.3062354.

[Hsu19] Aaron Wen-yao Hsu. "A data parallel compiler hosted on the GPU". PhD thesis. Indiana University, 2019.

[GNU] GNU. *Parallel GCC*. https://gcc.gnu.org/wiki/ParallelGcc. Accessed: 2021-07-19.

[Süß+18] Tim Süß, Nils Döring, André Brinkmann, and Lars Nagel. "And now for something completely different: running Lisp on GPUs". In: *2018 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE. 2018, pp. 434–444.

[KD13] Andrew W. Keep and R. Kent Dybvig. "A Nanopass Framework for Commercial Compiler Development". In: *SIGPLAN Not.* 48.9 (Sept. 2013), pp. 343–350. ISSN: 0362-1340. DOI: 10.1145/2544174.2500618. URL: https://doi-org.ezproxy.leidenuniv.nl/10.1145/2544174.2500618.

[Ghu06] Abdulaziz Ghuloum. "An incremental approach to compiler construction". In: *Proceedings of the 2006 Scheme and Functional Programming Workshop, Portland, OR. Citeseer*. Citeseer. 2006.

[Gro+11] Andre Vincent Pascal Grosset, Peihong Zhu, Shusen Liu, Suresh Venkatasubramanian, and Mary Hall. "Evaluating graph coloring on GPUs". In: *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*. 2011, pp. 297–298.

A. Abstract Syntax Tree

This appendix contains an overview of all node types present in the abstract syntax tree. Various similar node types have been grouped together based on function and possible positions in the syntax tree relative to other nodes. Each of the following sections describes the members of one category of nodes.

In our implementation, a node is represented as a tuple of 6 fields: the node type, the resulting data type, the index where the parent node can be found, the depth of the node in the abstract syntax tree, the index of the node in its sibling list and the node's node data.

A.1. Statements

This section contains the various statement nodes of the programs. Statement nodes do not yield any return value and thus their data type is always set to invalid.

A.1.1. Control Flow Statements

Control flow statements are statements that change the control flow of the program, such as branches and loops. These statements may have either statements or statement lists as children, representing the statements to be executed. The first child of the conditional statements and the second child of the while loop must be an expression returning an integer, which is used as the conditional. The various conditional node node types are listed below:

`if_statement`
`if_else_statement`
`while_statement`

A.1.2. Expression Statements

Expression statements take an expression, described in Section A.2, as child. This expression may return either integer, float or void. Any non-void result will be discarded.

`expression_statement`

A.1.3. Return statements

Return statements end the current function and return control to the caller. They may take one operand of which the data type must match the data type returned by the function. If the function return type is void, a return statement may have no arguments.

```
return_statement
```

A.1.4. Empty Statements

These statements represent statements that perform no operation. They may be generated by the parser to represent statements that should perform no action. For example, an infinite loop, as would belong to the source code `while(1);` would have an empty statement as a child of the while node.

```
empty_statement
```

A.2. Expressions

Expressions represent computations and therefore always return either integer, float or void. Expressions may have other expressions as children.

A.2.1. Binary Expressions

Binary expressions operate on two operands of the same type and return a result of the same data type as their operands. The children of the four arithmetic expressions, addition, subtraction, multiplication and division may return either integer or float values, whereas all other binary expressions allow only integers as operands.

```
add_expression
sub_expression
mul_expression
div_expression
mod_expression
bitwise_and_expression
bitwise_or_expression
bitwise_xor_expression
left_shift_expression
right_shift_expression
logical_and_expression
logical_or_expression
```

A.2.2. Unary Expressions

Unary expressions have only one operand of the same type as their result. Negations may take either operands returning integers or floats and all other unary expressions may take only operands returning integers.

```
negation_expression
logical_not_expression
bitwise_complement_expression
```

A.2.3. Comparison Expressions

These expressions are used to compare two values of the same data type. The return type of these nodes is always an integer, until the preprocessing stage of the compiler modifies the return type field of the nodes to match the operand types. This does not affect their location in the syntax tree and they may appear in any location where an expression returning an integer is expected.

```
equal_expression
not_equal_expression
less_than_expression
greater_than_expression
less_equal_expression
greater_equal_expression
```

A.2.4. Function call expressions

The function call expression is used to call other functions in the program. Their node data will contain the integer identifier of the function to be called, which should match the identifier contained in the node data of the function declaration declaring the function to be called. Function call expressions may return either integers, floats or void, depending on the return type of the called function. They will only have one child node, which must be a function call argument list.

```
function_call_expression
```

A.2.5. Assignment expressions

Assignment expressions will store the result of their right operand in the reference represented by their left operand. Their right operand must be an expression returning either integer or float and their left operand must be an expression returning either integer reference or float reference, with the type of the left operand matching the reference type of

the right operand. The expression itself will return the value stored and will thus match the data type of the right operand.

`assignment_expression`

A.2.6. Literal Expressions

Literal expressions represent constants in the program. Their data type will be set to either integer or float, depending on the type of the constant. Their node data will contain the value of the constant.

`literal_expression`

A.2.7. Symbol Expressions

Symbol expressions define the usage of symbols. Their return type will be a reference type, either integer reference or float reference and their node data will contain the integer identifier of the referenced symbol.

`declaration_expression`
`identifier_expression`

A.2.8. Cast Expressions

Cast expressions convert expressions returning one data type to an expression returning another data type. The `cast_expression` node may be used to convert integers to floating-point values and vice versa. The `dereference_expression` may be used to convert references into their underlying values.

`cast_expression`
`dereference_expression`

A.3. Other Nodes

These nodes are used to represent various other parts of the program that do not classify under any of the previous categories.

A.3.1. Function Declarations

The function declaration node is used to declare a function. These nodes are children of the list of function declarations that is the root of the abstract syntax tree. Their node data will contain an integer identifier for this function and they will have a list of arguments and a statement or statement list as children.

`function_declaration`

A.3.2. Function Call Argument Nodes

These nodes represent arguments passed to a function call. The function call argument list node must appear directly below a function call expression node. A function call argument list node will only have function call arguments as children, who may have expressions returning integers or floats as children. Two extra node types are generated by the compiler to differentiate nodes based on the calling convention. The process used to split function call argument nodes into these two extra node types can be found in Section 3.1.1.

`function_call_argument_list`
`function_call_argument`
`function_call_argument_float_in_int`
`function_call_argument_stack`

A.3.3. Function Argument Nodes

These nodes are used to represent arguments to a function declaration. The function argument list appears as a child of a function declaration. The function argument nodes themselves are children of this argument list and may only have declaration nodes as children. The argument nodes are split by the preprocessing stage of the compiler in order to differentiate the nodes based on the calling convention. The procedure used to split function argument nodes into the other node types is described in Section 3.1.1.

`function_argument_list`
`function_argument`
`func_arg_float_in_int`
`func_arg_stack`

A.3.4. Statement Lists

Statement lists hold a list of statements. They may have a variable number of children, all of which must be either statements or other statement lists.

`statement_list`

A.3.5. Compiler Internal Nodes

These nodes are used only internally by the compiler and do not correspond to any actual source code. They are used to support certain algorithms within the compiler.

`invalid`
`while_support_node`
`function_declaration_support_node`

B. Reproducibility

This appendix describes how to reproduce the results we obtained during the experiments performed in Chapter 5. All source code and datasets used during our experiments are available from the git repository located at `https://github.com/Snektron/pareas.git`.

The Pareas compiler was written in a combination of Futhark and C++. It therefore requires both the Futhark compiler and a modern C++17 compatible compiler to be present on the build path. For our experiments, we used version 0.20.0 of the official Futhark compiler and version 10.2.0 of g++. Our implementation uses the Meson build system. Meson requires an additional build system to be installed in order to compile code. We have tested both the ninja and samurai build systems, both of which can be used to compile our project.

After meson and either ninja or samurai have been installed, the project can be compiled by issuing the commands listed in Listing B.1. This instructs meson to set up the compiler using CUDA as a backend for Futhark and to initialize a build directory in the folder `build`, which will be created automatically. The C++ standard library was linked statically to prevent a library version mismatch that might occur at runtime if multiple versions of the C++ standard library are installed on the same system and the version of the default C++ standard library is outdated.

Listing B.1 Compiling the Pareas compiler

```
LDFLAGS="-static-libstdc++" meson build \
    -Dfuthark-backend=cuda \
    -Dbuildtype=release
cd build
# Alternatively samu can be used if samurai is used instead of ninja
ninja
```

The compiler will now be present in the build directory as an executable named `pareas`. The compiler can be invoked using a path to a source file as an argument to compile a single file. The compiler will write the timings of the various substages to standard output. All source files used to perform the experiments listed in Chapter 5 are present in the `test` directory.

To simplify the testing process, scripts have been provided to automatically run all tests once the compiler has been compiled. To automatically run all tests, the script `run_tests.sh` can be used. This script, which takes no arguments, will execute 31 runs for all valid tests present in the directory `test` and write the results to a directory named `results`, automating all testing.