Master's thesis

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

Sigve Sjømæling Nordgaard

Program feature impact on the treewidth of the RVSDG IR

Master's thesis in Computer Science
Supervisor: Jan Christian Meyer
June 2020

Norwegian University of Science and Technology

Sigve Sjømæling Nordgaard

Program feature impact on the treewidth of the RVSDG IR

Master's thesis in Computer Science
Supervisor: Jan Christian Meyer
June 2020

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

Problem Description

This study will investigate the relationship between particular program properties and the bounds of the treewidth metric of its representation as a Regionalized Value State Dependence Graph.

Supervisor: Jan Christian Meyer

Abstract

In this thesis we investigate program feature impact on the treewidth of the Regionalized Value State Dependency Graph (*RVSDG*) intermediate representation. We look at how properties of programs translate to changes in the treewidth of their corresponding RVSDGs. This includes measurements of how different optimizations affect the structure of the programs, including optimizations such as loop unrolling and function inlining. We also create programs to induce features that change the treewidth, including different ways of passing arguments to functions and liveness range of variables. Where we are able to change the structure of a program without changing its semantics, we also look at the runtime of the program with different treewidths to try to determine if any relationship between the treewidth and the runtime of the program exists.

Acknowledgements

I would like to express my gratitude towards my supervisor Jan Christian Meyer and Nico Reissmann for their assistance with this project. Thanks to Jan Christian for guiding me through the quirks of academic writing, and mentoring me in accomplishing the thesis. Also many thanks to Nico for assisting me with my technical challenges, introducing and helping me understand the RVSDG and compiler technology in general. Thank you both again for all proofreading, correction and discussion done in the writing of this thesis.

I would also like to thank Magnus Hetland and Magnus Själander for fruitful discussions about this project. Finally, credit is due to Nico, Magnus S. and Asbjørn Djupdal for developing the RVSDG software which made this project possible.

Table of Contents

Problem Description i

Abstract ii

Acknowledgements iii

Table of Contents vi

1 Introduction 1

2 Background 3
- 2.1 Intermediate Representations 3
- 2.2 Data-flow Centric IRs 4
- 2.3 Tree Decomposition 4
- 2.4 Heuristics 5
- 2.5 Related work 6
  - 2.5.1 Finding the treewidth 6
  - 2.5.2 Creating the tree decomposition 6
  - 2.5.3 Algorithms on graphs of bounded treewidth 7

3 Theory 9
- 3.1 The Regionalized Value State Dependency Graph 9
- 3.2 Tree decompositions 11
- 3.3 Heuristics 12
  - 3.3.1 The min-fill heuristic 12
  - 3.3.2 The minor-min-width heuristic 14
- 3.4 Program Features 16
  - 3.4.1 Functions 16
  - 3.4.2 Liveness Analysis 17

4 Method 19
4.1 Representing the RVSDG as a dotfile 19
4.2 rvsdg-treedc framework 20
4.2.1 Parser and data structures 20
4.2.2 Heuristics 22
4.3 Benchmarks 22
4.4 Metrics 22
4.5 Optimizations 23
4.6 Function arguments 25
4.7 Variable Liveness 25

5 Results & Discussion 27
5.1 Results 27
5.1.1 General treewidth results 28
5.1.2 Optimizations 29
5.1.3 Impact on program runtime 38
5.1.4 Functions 38
5.1.5 Variable liveness 40
5.2 Discussion 44
5.2.1 General treewidth results 45
5.2.2 Analysis of dependencies in functions 46
5.2.3 Analysis of dependencies with respect to variable liveness 50

6 Conclusion 57
6.1 Conclusion 57
6.2 Future work 58

Bibliography 59

A Result Tables 63

Chapter 1

Introduction

Compilers translate from a source to a target language and perform a set of optimizations, creating faster and more efficient programs. An *Intermediate Representation* (IR) is an abstract representation of a program, which the compiler uses to aid in both translation and optimization. The IR is language independent and allows us to separate the language parsing and code generation sections in the compiler.

Data flow based IRs have been developed due to the limitations of traditional control flow based IRs, and attempt to raise the abstraction level of the representation by modeling data flow explicitly rather than implicitly. This both simplifies optimizations and allows the exposure of parallelism in programs. These methods have not been extensively evaluated in practical applications. For an IR to be practical, it has to implement a range of optimization algorithms such as *Constant Subexpression Elimination* (CSE) and *Register Allocation* (RA). Since most optimizations are computationally hard, real world compilers apply algorithms that are not provably optimal, using a range of heuristics [21, 1].

In this thesis we investigate the tree decomposition properties of the RVSDG data flow graph IR. The *tree decomposition* is a generalization on classes of tree-like graphs which allows for solving problems efficiently. We know that if a control flow graph has a low and bounded *treewidth*, we can find provably optimal, efficient algorithms for many optimization problems.

We implement a framework for parsing and calculating the upper and lower treewidth bounds for a RVSDG-based `C` compiler. We apply this to select benchmark programs, and evaluate the feasibility of applying the results to compiler optimizations. We find that these treewidths are low and bounded, opening several avenues of possible future research.

Further, we investigate how different program features impact the graph structure and resulting treewidth. This is done by using either benchmarks, inducing structural changes to the program by applying different optimizations, or by using custom programs that manually induce these changes. We show how different optimization passes affect the program treewidth, and find that while there is no correlation between the runtime changes of these individual passes and the corresponding treewidth, there is a correlation between the increased efficiency of the collection of optimizations and the reduced treewidth this

produces. We present a detailed analysis of this optimization process, and identify a small subset of optimizations that affect the treewidth.

We also analyse different aspects of function program features, finding that both the method of argument passing, and the function call order affects the resulting program treewidth. We present an analysis of variable liveness and variable allocation, finding that for an increasing amount of variables allocated or an increasing number of references to these variables, the treewidth increases. While only one of the parameters increases this growth is bounded. However, when both parameters increase, the treewidth grows indefinitely.

Finally, we conduct a detailed analysis of the RVSDGs generated by these programs, relating the changes in treewidth to structural changes in the corresponding graphs. This breakdown focuses on how the program features affect the data flow and state dependencies between the nodes in the graph, and how this results in different types of cycles that scale with the number of nodes or operations.

In **Chapter 2**, we describe the difference between control- and data flow based IRs, the RVSDG specifically, and review why the tree decomposition properties are useful in finding optimal optimization algorithms. We also introduce the heuristic approach to finding the treewidth of the tree decomposition, and give an overview of alternative approaches and related work in the field. In **Chapter 3**, a theoretical background for the RVSDG and tree decomposition is given, along with a description of the heuristic algorithms used for the treewidth evaluation. Further, we define the function and liveness program features we investigate in this thesis. **Chapter 4** describes our experimental setup. This includes the developed parser and treewidth framework, along with both implementation and benchmarking of treewidth heuristics on the graphs produced. We also describe how different optimizations are tested, which program features are induced, and how these are implemented. We present and discuss the results from our benchmarks and program feature investigations in **Chapter 5**, and finally summarize our findings and discuss possible future work in **Chapter 6**.

Chapter 2

Background

Compilers perform a set of optimizations on the source program to create a faster and more efficient target. Examples of two important categories of optimizations are

1. *Redundancy elimination*: The abstraction level of high-level programming languages results in overheads when compared to the efficiency of low-level languages. Redundancy elimination is a set of optimizations that mitigates some of these effects, such as pruning duplicate and unnecessary computations done in the program.

2. *Register allocation*: Processor architectures are becoming more complex, making them harder to optimize programs for. This complexity also leads to more opportunities to improve the way the code executes, such as when mapping program variables to a set of registers in the computer.

The compiler optimizer is rarely able to create optimal code. This is both because optimization is a CPU- and memory-intensive process, and because many of the problems the compiler tries to solve are computationally hard. For practical use, the compiler has to balance the need for correctness, efficiency, and compile time. Most optimizations are therefore heuristic methods, used to improve performance and resource use in common program patterns [1].

In this chapter we give an introduction to the concept of Intermediate Representations, which lie at the heart of every optimizing compiler. We then present the intermediate representation we are investigating in this thesis, the RVSDG. Finally, we introduce the treewidth properties we are looking for, discuss how we find these properties, and alternative methods for doing this.

2.1 Intermediate Representations

The *Intermediate Representation* (IR) is an abstract representation of a source program, often a graph, generated by the front end of the compiler. This representation is used as

a data structure for various optimization algorithms, simplifying the process by exposing data- and control-flow.

The most common IR used in modern compilers is the Control Flow Graph (CFG), which is a graph representation of the possible control flow between basic blocks in a program. The CFG is simple to construct and destruct, but is restricted in several ways. It has a low abstraction level, providing no structural information about procedures bodies, and cannot explicitly represent constructs such as loops and functions. Data flow is represented implicitly and in a specific execution order, making the CFG "an inherently sequential IR" [32].

Because of the limitations of this approach, IRs have been developed based on the flow of data rather than the flow of control. Many optimizations require data flow, and explicitly representing this raises the IRs abstraction level. These *Data Flow Graphs* represent programs in demand-dependence form, modeling the flow of data and state, with only an implicit and restricted form of control flow. This simplifies data flow optimizations, and makes the representation and exposure of parallelism embedded in programs possible.

2.2 Data-flow Centric IRs

Compared to the CFGs, data-flow centric IRs have not been extensively evaluated for feasibility and usability in practical implementations. This thesis will focus on the *Regionalized Value State Dependence Graph* (RVSDG), presented in Bahmann *et al.* [4]. The RVSDG is a data centric IR representing programs in demand-dependence form, modeling data flow between operations as edges between nodes. This is opposed to the CFG, which needs supporting data structures such as call graphs and loop trees to perform complex optimizations. It is also able to encode structured control flow, as the CFG does, and model the entire program within a single IR.

The Value Dependence Graph (VDG) represents control flow as data flow and is an early Data Flow Graph described by Weisse [41]. Johnson [19] notes that, "This representation removes any specific ordering of instructions (nodes), but does not elegantly handle loop and function termination dependencies". As a solution, Johnson proposes the Value State Dependence Graph (VSDG) as an extension of the VDG, introducing state dependency edges to model sequentialised computing.

Mapping a language with possibly unstructured control flow requires efficient *construction* algorithms from explicit control flow, and *destruction* algorithms to reestablish control flow before generating the desired machine code. The RVSDG permits the complete recovery of the original control flow from a procedure represented in a demand-dependence graph. It has an advantage over earlier methods using the VSDG, which have potential overhead in code size growth and/or sub-optimal control flow recovery.

2.3 Tree Decomposition

We will look at the tree decomposition and treewidth properties of the RVSDG. A desirable property of trees is the restricted amount of interaction between the nodes. This allows the separation of the tree into two disconnected components by removing a single

vertex. A *tree decomposition* is a mapping of a graph into a related tree that captures the possibility of decomposing the graph into disconnected pieces by removing a *small number of the nodes*. The *treewidth* is a measure of the tree decomposition of the graph, and thus captures the graphs property of being tree-like [20]. A formal definition of these concepts is given in Section 3.2.

As with trees, many NP-complete problems are tractable on graphs of bounded treewidth [20]. One such application is in the implementation of efficient optimization algorithms. Using tree decompositions for CFGs, significant improvements in compiler optimizations have been found, such as providing algorithms in polynomial time for band selection, redundancy elimination, and register allocation. The use of tree decompositions allows efficient and provably optimal algorithms [21].

It is important that these tree decompisitions have small treewidths to achieve an efficient run-time and result quality of these algorithms. This is because the treewidth denotes how many nodes are needed to remove to separate the graph, which is the main property of trees we want to capture. Specifically, many problems on graphs can be solved in single-exponential time for the treewidth of the graph, and linear time for its number of nodes [8]. Thorup found that all `goto`-free programs have exactly this, and Throups heuristic has *e.g.* been used to obtain the tree decompositions of the CFGs in the SDCC `C` compiler [38].

This work aims to investigate the tree decomposition properties of RVSDG IR by measuring the treewidth of a set of benchmark programs. If these treewidths are shown to be low and bounded, further work can be put into either determining an upper bound analytically, or finding the tree decomposition itself. This would also indicate that we could use the tree decomposition in finding faster and more efficient optimization algorithms for the RVSDG.

By surveying what program features affect the treewidth of the RVSDG, we also gain insight into how to write programs with lower treewidths. If a correlation between the treewidth of the program and its runtime efficiency is established, this further opens avenues of research. Either by directly connecting the graph properties of programs features to runtime efficiency, or by expanding the area of application beyond algorithmic efficiency to properties of the programs themselves.

2.4 Heuristics

To find the tree decomposition of a graph G, most algorithms start by finding the treewidth, k. Given k, the next step will then either report that the treewidth of G is more than k, or it will construct the tree decomposition with maximum width ck for a constant c. These algorithms also outputs the tree decomposition in time proportional to k [8]. Finding the treewidth thus lets us know if we can find a small tree decomposition, what k to input to the tree decomposition algorithm, and the runtime of the algorithm.

Finding a decomposition with low treewidth for a graph means that we can find efficient solutions to computationally hard problems on the graph. Specifically, for a graph of n vertices and a treewidth of k, we can find dynamic programming solutions that run in time $2^{\mathcal{O}(k)}n$. Determining the treewidth is also an important stepping-stone for most algorithms that find the tree decomposition. This is because these algorithms either find a tree

decomposition for a given k, or reports that the treewidth is higher than k. Finally, having a low treewidth tells us that the tree decompositions can be found in time $f(k) \cdot g(n)$ [8].

Deciding if the treewidth of a graph is at most k is an NP-complete problem. This means we need alternative methods to obtain the treewidth. Bodlaender found that there exists a linear time algorithm that solves this problem, but even with improvements found by other authors, the algorithm is not practical. This is because of a large hidden constant in the $\mathcal{O}$-notation, which makes the runtime "useless from a practical point of view" [6]. An alternative approach is using heuristics that approximate the treewidth. These approaches are desirable, since they can be found for polynomial runtime in the size of the graph. This work takes this approach. The specific heuristics implemented are presented in Section 3.3.

2.5 Related work

2.5.1 Finding the treewidth

As shown in a survey by Bodlaender [6], in addition to the exact fixed parameter cases and heuristics mentioned above, there exist several other methods for exactly finding the treewidth of a graph. These approaches are usually either restricted to special graph cases, or run in exponential time. They are therefore not relevant for our work since, as stated by Bodlaender *et al.* in [8], for most applications these properties cannot be derived. Also, as noted by Wersch & Kelk in [39], heuristic algorithms for finding the treewidth usually have more accessible and comprehensible studies behind them, while exact solutions are less widespread.

Techniques also exist for preprocessing the graphs, decreasing the size of the graph to speed up runtime. This is not required, as the heuristics we implement run in polynomial time on an efficient `C++`-implementation. Correspondingly, we can prostprocess the tree decomposition after it is generated to reduce its treewidth. This is desirable because the treewidth of the tree decomposition found by most algorithms is not minimal. As we only look at the treewidth itself, this is out of the scope for this work.

2.5.2 Creating the tree decomposition

There are several approaches to creating the tree decomposition. Firstly, we can use an approach from the class of algorithms mentioned in Section 2.4. These are the algorithms that are able to either find the decomposition for a given treewidth, k, or report that the treewidth is at least k. An overview of these methods can be found in Bodlaender *et al.* [8], given as the *constant factor approximation* algorithms.

Another approach is using satisfiability (SAT) based approaches, encoding the tree decomposition as a boolean satisfiability problem. An evaluation of this approach is given in Berg & Järvisalo [5], where it is found that this SAT-solvers can in some cases be competitive with dedicated algorithms. One can also set out to decide certain properties of the graph, allowing the use of specialized algorithms for finding the tree decomposition. An example of this approach is creating the perfect elimination ordering of a triangulated graph, and is described in Section 3.3 for the min-fill heuristic.

We can also try to determine the exact treewidth to possibly reduce the size of the generated tree decomposition. As we already have implemented the necessary heuristics required for the branch-and-bound algorithm presented by Googate & Dechter [17], an obvious approach would be to fully implement the given quickBB-algorithm. Otherwise, we can set out to determine any of the graph properties for the RVSDG that allow special case exact algorithms in polynomial time.

2.5.3 Algorithms on graphs of bounded treewidth

As mentioned in Section 2.4, we can use dynamic programming algorithms to solve computationally hard problems efficiently with the use of the tree decomposition for a graph of bounded treewidth. This is done by traversing the tree decomposition, computing partial solutions to the problem along the way, only considering the vertices in the current bag. Due to the properties of the tree decomposition, we know that these are the only vertices that will interact at any point, and thus we are able take the whole instance into account [10, 7].

To solve problems like register allocation or CSE on the RVSDG in polynomial time, we first need to define algorithms such that they can be solved using these dynamic programming techniques. If we can reduce these optimizations down to classic algorithms like graph coloring or independent vertex sets, we have known solutions using the tree decomposition [14]. However, the work done by Krause in [21] indicated that these problems are most likely complex enough to require specialized approaches, such as first finding the *nice* tree decomposition.

Chapter 3

Theory

In this chapter we give a brief introduction to the RVSDG IR. For a complete definition of the RVSDG, the reader is referred to [32, 33]. Further, we give a short graph theoretical background and define the tree decomposition and the treewidth parameter. We then describe heuristic algorithms for finding the upper and lower bound treewidth of a graph. Lastly, we define the program features investigated in the thesis. This includes function parameters, ordering of function calls, and variable allocation and liveness.

3.1 The Regionalized Value State Dependency Graph

The *Regionalized Value State Dependency Graph* (RVSDG) is an acyclic multigraph consisting of two different substructures. *Nodes* and *Edges* represent data flow in the graph, with edges connecting one nodes output to exactly one corresponding input of another node. Edges can also represent states, not transferring data but modeling a necessary ordering of operations. The nodes are one of two types: *Simple* – representing primitive operations, represented by nodes with a corresponding operation, or *Structural* – containing more complex operations, represented as sub-RVSDGs named *regions*.

Each region consists of arguments and results, corresponding to node inputs and outputs, the set of edges that connect these, and a set of nodes. Structural nodes contain a set of regions, used to model structural program constructs such as conditionals or loops. Six different types are defined to capture the necessary higher-level programming constructs, enabling the representation of the entire program in a single IR.

γ-*nodes* are decision points that model conditionals such as if and switch statements.

θ-*nodes* represent tail-controlled loops, which can be used in conjunction with γ-nodes as a basic building block to represent any loop structure.

λ-*nodes* model functions. Inputs to the node are the external variables the function depends on, and the output is a value representing the node itself. An apply node is used to invoke a function, inputting arguments and computing the actual function body.

δ-nodes similarly model a global variable, inputting external variables the node is dependent on, and outputs a single result representing the right-hand side value of the variable.

ϕ-nodes are required to express mutually recursive functions without introducing cycles. These are meta regions consisting of λ-nodes, containing all their definitions and corresponding inputs, outputting them as a single result.

ω-nodes are top-level nodes modeling translation units (TU). This is required to import and export data and functions between the different TUs in the program.

Using these structural nodes, we can represent all inter- and intra-procedural dependencies in the IR. Also, with the use of simple nodes, the RVSDG is able to represent both high- and low-level programming constructs. This normalization of programming constructs simplifies optimizations by providing a canonical form of the representation of loops, functions, and conditionals. Further benefits of using the RVSDG are that no ordering of independent expressions are added, and that the flow of data implicitly enforces the Single State Assignment form [32].

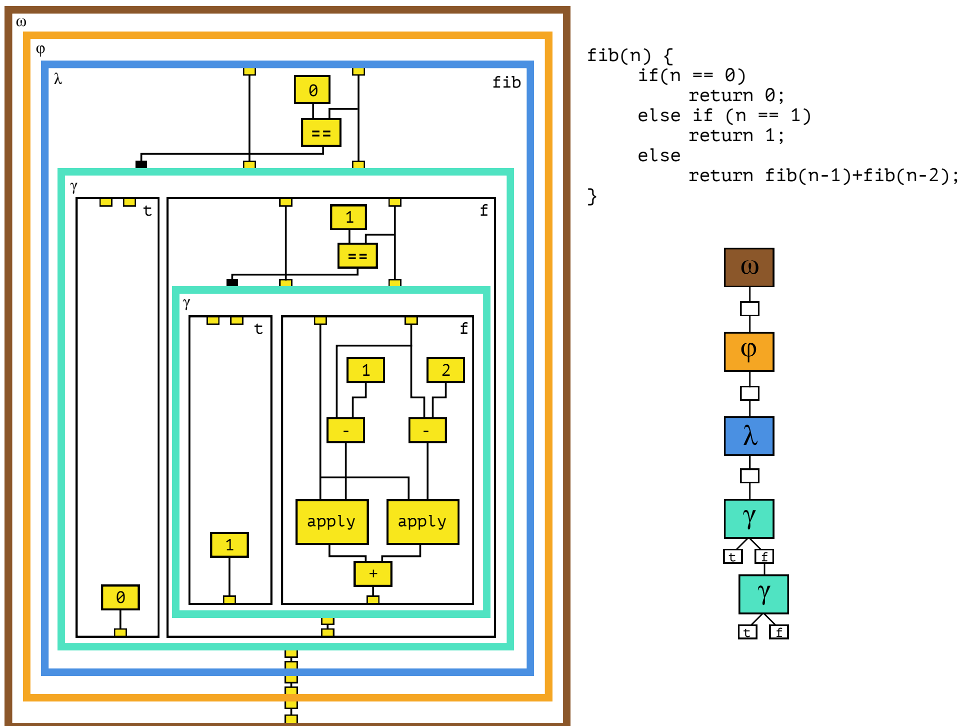


Figure 3.1: Example RVSDG for the Fibonacci function shown on the right. The `fib` function is modeled in the λ-node, which is contained in and exported by the ω-node. Since the function is recursive, it also needs to be contained in a ϕ-node to be able to call itself. The control flow in the function is handled by two nested γ-nodes.

3.2 Tree decompositions

Let $G(V, E)$ be an undirected graph where V is the set of vertices and E is the set of edges of the graph. Let $n = |V|$ denote the number of vertices of the graph. For vertices $u, v \in V$, u is a *neighbor* of v in G if $\{u, v\} \in E$. Two neighboring vertices are said to be *adjacent*. The set of all neighbors of v is the *neighborhood* of v. A set of vertices $C \subseteq V$ is a *clique* of G if each distinct vertex in C is pairwise adjacent.

A graph $H(V', E')$ is the *subgraph* of G if $V' \subseteq V$ and $E' \subseteq E$. H is an *induced* subgraph of G if, for every vertex in both graphs, every adjacent vertex in G is also adjacent in H. $G[X]$ is the induced subgraph of G with the vertex set $X \subseteq V$.

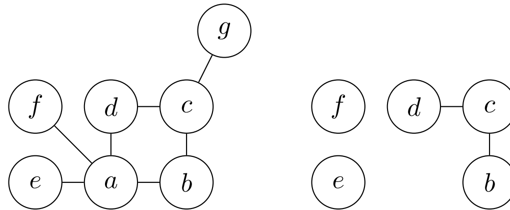


Figure 3.2: Graph G with the induced subgraph $G[X]$ for $X = \{b, c, d, e, f\}$.

Definition A *tree decomposition* of the graph $G(V, E)$ is a tree $T(I, F)$ and a collection of subsets $\chi = \{\chi_i, i \in I\}$ of V, called the *bags* of the tree decomposition. The pair (T, χ) must satisfy the following properties.

Vertex Coverage $\bigcup_{i \in \chi_i} \chi_i = V$

Edge Coverage For each $\{u, v\} \in E$ there exists an $i \in I$ such that $u, v \in \chi_i$

Coherence If t_2 is on the path from t_1 to t_3 in T, then $\chi_{t_1} \cap \chi_{t_3} \subseteq \chi_{t_2}$ for $t_1, t_2, t_3 \in I$

The *width* of a tree decomposition is $\max_{i \in I} |\chi_i| - 1$. The *treewidth* of the graph G is the minimum width over all tree decompoisitions of G. Due to the edge coverage property, trees have two vertices in each bag. To define treewidth such that trees have a width of 1, we therefore subtract one from the size of the largest bag in the definition of the treewidth.

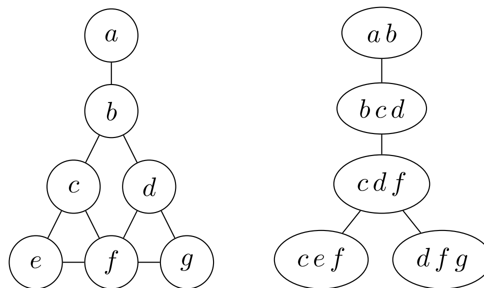


Figure 3.3: Graph with a possible decomposition of width 2.

3.3 Heuristics

Gogate & Dechter [17] describe an algorithm for calculating the treewidth based on a branch and bound search. This approach relies on a set of heuristics to prune branches in the solution state space tree. These heuristics provide either an upper or lower bound on the solution, and thus an upper or lower bound on the treewidth of the graph we are analyzing.

Gogate & Dechter present three existing heuristics for computing the upper bound, and one novel approach to compute the lower bound on treewidth. We implement these heuristics, and use them to evaluate the tree decomposition properties of the RVSDG. As one upper bound heuristic consistently finds a tighter bound, we only report this when presenting our results. In this section, we describe this best performing upper bound heuristic, and the lower bound heuristic used in the evaluation.

3.3.1 The min-fill heuristic

To define a heuristic algorithm for finding the upper bound on treewidth, we first present three lemmas from graph theory. A graph is *triangulated* if every cycle in the graph is not chordless. A *chord* is en edge between two vertices in a cycle that is not part of the cycle itself. A *chordless* cycle is a cycle of length $k > 3$ that has no chord.

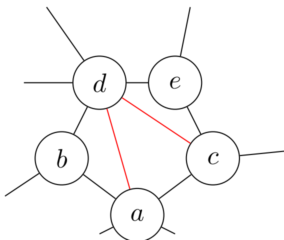


Figure 3.4: Section of a graph, where the vertices shown forms a cycle. The red edges are chords such that this section of the graph is triangulated. The graph containing all edges and vertices in the section is the triangulation of the corresponding graph containing only the black edges.

Triangulated graphs are connected to the tree decomposition with the following property [16]

Lemma 1. *If G is a triangulated graph and the maximum size of a clique in G is denoted by $\omega(G)$, the treewidth of G equals $\omega(G) - 1$*

The *triangulation* of $G(V, E)$, is the triangulated graph $H(V', E')$ such that $V = V'$ and $E \subseteq E'$. From this definition, we get the following property [34].

Lemma 2. *For every graph G there exists a triangulation H such that the treewidth of G equals the treewidth of H*

One approach to finding the treewidth of our input graph G is finding such a triangulation. This class of heuristics is based on finding the triangulation H of G that minimizes

the size of the maximum clique. This is an upper bound on the treewidth of G, as we know from Lemma 2 that the treewidth of G is at most as large as the treewidth found for H.

This triangulation can be built by constructing a *perfect elimination ordering* of the vertices in the graph. A vertex v of G is *simplistical* if its neighborhood induces a clique. The perfect elimination ordering is an ordering $\{v_1, v_2 \ldots, v_n\}$ where for every $i \in \{1, 2, \ldots, n\}$, v_i is a simplistical vertex in $G[X]$ for $X = \{v_i, \ldots, v_n\}$.

The triangulation can be constructed along the ordering as follows: make each v_i simplistical by connecting all its neighbors in $G[X]$, and then delete the vertex. Using Lemma 1 and 2 we then get our final lemma used to find the upper bound treewidth of G [11].

Lemma 3. *Given a perfect elimination ordering $\{v_1, v_2 \ldots, v_n\}$ of the triangulation of the graph G, the treewidth of G is given by*

$$\max(\textit{neighbors of } v_i \textit{ which is in the ordering at position } j, \textit{ for } j > i)$$

The *min-fill* heuristic creates the perfect elimination ordering by finding the vertex that adds the least number of edges when eliminated from the graph. This vertex is eliminated and placed in the ordering. Eliminating the vertex at each step lets us find the v_i defined in Lemma 3 for the current position in the ordering. The algorithm then repeats until all vertices are placed in the ordering. We implement the algorithm as shown in Listing 3.1.

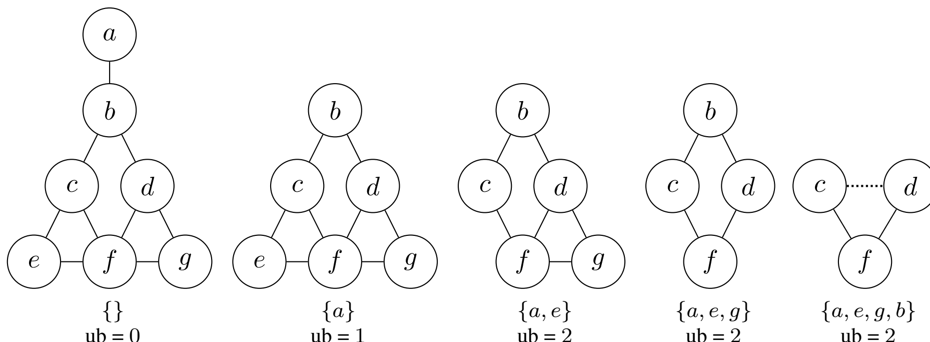


Figure 3.5: First steps of the min-fill heuristic. Vertices are shown below the graphs as they are added to the ordering, along with the current upper bound. The dotted edge denotes an edge added when eliminating a vertex.

```
min-fill-heuristic(Graph G) {

    ordering = [G.size]
    upper_bound = 0

    for i in 0 to G.size {
        /* Vertex in G that adds the fewest amount of edges
           when eliminated from the graph */
        v = minClique(G)
```

```
        /* Compares the current upper bound with the number of
           neighbors for v. This way each v in the ordering is
           compared against the degree of each element after it,
           and we get the upper bound according to lemma 3 */
        upper_bound = max(upper_bound, degree(v))

        /* Make v simplistic by making its neighborhood
           a clique, and then remove the vertex */
        eliminate(v)

        /* Add v to the ordering */
        ordering[i] = v
    }

    return upper_bound
}
```

Listing 3.1: The min-fill heuristic

3.3.2 The minor-min-width heuristic

Contracting an edge is the replacement of both vertices of the edge with a single vertex, such that the neighbors of the original vertices are neighbors of the new one. H is a *minor* of G if H can be formed from G via repeated edge deletion and/or edge contraction.

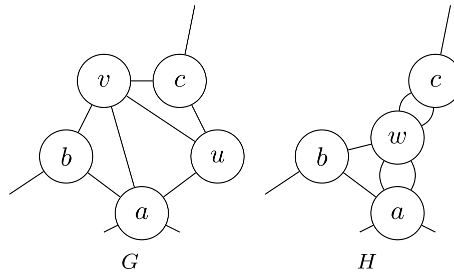


Figure 3.6: Contracting the edges u, v to w in G results in a new graph H that is a minor of G.

Due to the edge coverage property of the tree decomposition, stating that both endpoints of an edge have to exist in at least one bag, the minimum degree of a vertex in the graph is a lower bound for its treewidth. An improvement to this bound can be found by using Lemma 2, constructing a perfect elimination ordering for the graph and finding the minimum degree of the triangulation. This property can also be stated as the *width* of the graph, finding an ordering such that each vertex v_i is joined by an edge to at most w preceding vertices. We then know that the treewidth is at least equal to w [6].

Gogate & Dechter names this the *min-width* ordering. In a min-width ordering, if some vertex v_i has en edge with lb vertices ordered below it, then the treewidth of the graph is a least lb. They further improve on this by using the minor theorem, which states that the treewidth of a graph is never less than the treewidth of its minor. This means that

we can contract edges for each vertex we find to create smaller and smaller minors of the original graph. This approach, named the *minor-min-width* heuristic, has been found empirically to create a better bound than the min-width heuristic [17]. Pseudocode for our implementation of this heuristic is shown in Listing 3.2.

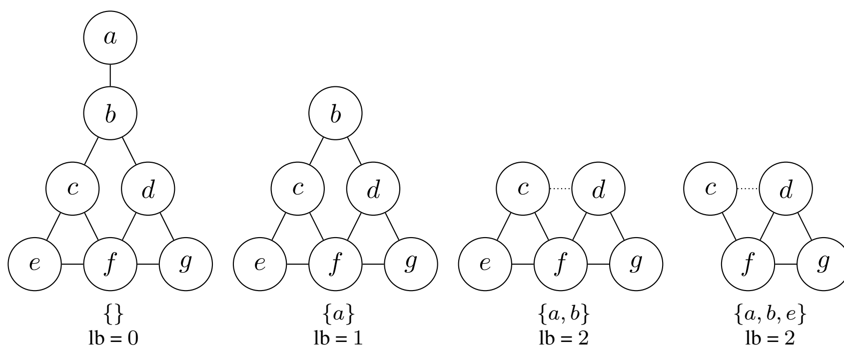


Figure 3.7: First steps of the minor-min-width heuristic. Vertices are shown below the graphs as they are added to the ordering, along with the current lower bound. The dotted edges denotes edges added by the edge contraction.

```
minor-min-width-heuristic(Graph G) {

    lower_bound = 0

    for i in 0 to G.size - 1 {
        /* Vertex in G with the smallest degree */
        v = minDegree(G)

        /* Get the neighborhood of v */
        neighbors = Neighbors(v)

        /* Find vertex u in N(v) such that
           the degree of u is minimum in N(v) */
        u = min_vertex(neighbors)

        /* Update lower bound according to the min-width ordering */
        lower_bound = max(lower_bound, degree(v))

        /* Contract the edge between u, v by:
           removing v and adding all its neighbors to u.
           The resulting graph from this is a minor of the existing graph */
        removeVertex(v)
        contractEdges(u,v)
    }

    return lower_bound
}
```

Listing 3.2: The minor-min-width heuristic

3.4 Program Features

In addition to using the benchmark programs of the PolyBench suite we introduce in Section 4.3, we write additional programs to induce different program features, and show how these affect the treewidth bounds of the resulting RVSDGs.

This section gives an overview of the program features we investigate in this thesis, separated in the categories *functions* and *liveness analysis*. These features are chosen based on our investigation of the benchmarks in the PolyBench suite. Comparing programs from the benchmark suite with different treewidths, these are the programs features we find to have a measurable *impact on the resulting treewidth*.

First, we clarify function terminology and enumerate three different methods of passing variables to functions. We also identify and define two different orderings of function calls. We then introduce variable liveness with respect to the RVSDG.

3.4.1 Functions

As we base our work on the investigation and compilation of programs in the `C` programming language, we use `C`-style terminology when discussing programs and program features. Subroutines or procedures are referred to as functions with a number of *formal* parameters. This is the number of *actual* parameters passed to the function, which is the number of values or references passed to the function call [9]. The actual parameter is also known as the function *argument*. In `C`, this number of parameters is set for each function, and we refer to this number of values with respect to the function, as the function *accepting* n arguments.

When a set of actual parameters is passed to the function, the function is *called* with this set of parameters. We similarly define the *call of a variable or reference*, as the call of a function accepting the variable or reference as an argument.

Three methods of passing arguments to functions

We identify three different methods of passing arguments to functions in the `C` language, which in 5.1.4 is demonstrated to have an impact on the resulting RVSDG treewidth. These methods are shown in Listing 3.3. Firstly, we can pass variables either as values *i.e.*, call by value. This includes copying each value, and using this local copy inside the function body. Secondly there are two ways of passing the variable as a reference, *i.e.*, call by reference. This is done either by passing the variables as members of a struct, or as located inside a contiguous array.

```
/* 1) Passing arguments as separate values */
int variable_sum(int v0, int v1, ...) {
    return v0 + v1 + ...;
}

/* 2) Passing arguments as members of a struct */
int struct_sum(args_t s) {
    return s.s0 + s.s1 + ...;
}
```

```c
/* 3) Passing arguments as elements in an array */
int array_sum(int a[N]) {
    return a[0] + a[1] + ...;
}
```

Listing 3.3: Three separate ways of passing arguments to a function in the C programming language.

Call orderings

We also find that when calling several functions, or the same function several times, the ordering of these calls also impact the resulting RVSDG treewidth. We identify and define two such orderings.

Firstly, we note the ordering of the function calls in an alternating order, as shown in Listing 3.4. As each function is separated into separate blocks, we denote this ordering of calls *blockwise*. An alternative ordering is calling the functions such that all calls accepting variable n_1 are done before all calls to variable n_2 *etc*. This ordering is shown in Listing 3.5. As the different functions are called in a sequence, we denote this as the *sequential* call order.

```c
int n1 = a;
...
int n7 = g;

f1(n1);
...
f1(n7);

f2(n1);
...
f2(n7);
```

Listing 3.4: Blockwise call order.

```c
int n1 = a;
...
int n7 = g;

f1(n1);
f2(n1);

...

f1(n7);
f2(n7);
```

Listing 3.5: Sequential call order.

3.4.2 Liveness Analysis

Liveness analysis, or live-variable analysis, is a data-flow analysis calculating what variables are *live* at a certain point in the program. Identifying live variables is simple in the data-flow graph, as the edges already represent the flow of data. Thus a variable is live at point p as long as there is some edge from p to the variable node [1].

Johnson [19] shows that that for two values connected by an edge in a data flow graph, this edge may introduce constraints on the liveness of the variable. For the RVSDG specifically, the nodes interact when the values or instructions they represent reference each other, either directly as a data edge or indirectly as a state edge. Since the RVSDG is ordered by these references, variables that only reference and are referenced by a set of neighboring nodes in the graph will have a shorter liveness range, while variables that do not will have a longer liveness range.

Chapter 4

Method

The RVSDG is implemented using the `jive` [26] compiler back end. `jive` implements the intermediate representation and provides interfaces and data structures to the `jlm` [31] framework, which further provides the compiler front end and optimizer for the LLVM IR. From `jlm` we can generate an XML representation of the RVSDG IR from a compiled LLVM bytecode program.

This is used as the source of our investigations into the treewidth of the RVSDG IR. Sections 4.1 and 4.2 describes the provided `rvsdg-treedc` framework which includes a parser of the RVSDG XML output, transforming it into a corresponding graph representation in the `dotfile` format. Further, we show how this is used in our graph framework to measure the graphs upper and lower bound treewidths. Section 4.3 describes the benchmarks used to generate graphs for measuring the treewidth and the metrics used to evaluate our results.

We then describe the methods used to induce the different program features presented in Section 3.4. In Section 4.5 we introduce how different optimizations can be used to change the structures of programs, and how these optimizations are applied. In Section 4.6 we show that different methods of passing function parameters affect the program treewidth, and present how this is measured and tested. Finally, Section 4.7 demonstrates the same with respect to variable liveness and variable allocation.

4.1 Representing the RVSDG as a dotfile

As described in Section 3.1, the RVSDG consists of a set of regions which contain one or more nodes. The relationship X contains Y is represented in the XML by Y being a child of X. These nodes may be simple, containing a tuple of inputs and outputs, or structured in which case they also contain sub-regions. Nodes and regions contain corresponding concepts: inputs/arguments and outputs/results, modeled as children of their corresponding node or region. Finally, each region also contains a set of edges between the different nodes and regions. Edges are modeled as unique XML-elements, containing an attribute for the source regions argument and result, or source nodes input and output.

To be able to model this directly as a graph, we represent each region as a subgraph. The XML parser is implemented in `C++` using the `pugixml` XML processing library [2]. The parser reads one XML file produced by the `jlm-print` tool, loads it into an in memory representation of the RVSDG[1], and produces a corresponding graph in the dotfile format [15].

To perform this translation, we map nodes and edges from the XML to the graph by considering inputs to be edges into, and outputs edges out of their corresponding nodes. Arguments and results of a region are modeled as the entry and exit nodes of the graph respectively. An example graph representation for a very simple LLVM bytecode input program is shown in Figure 4.1.

4.2 rvsdg-treedc framework

In this section we describe the provided `rvsdg-treedc` framework developed with this thesis. The pipeline to generate a graph and determine its treewidth is outlined in Figure 4.2, where our framework consists of the last two steps.

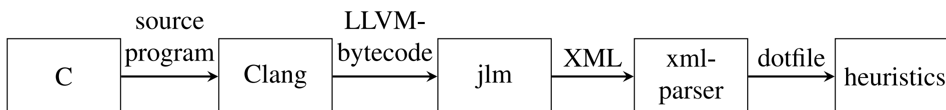


Figure 4.2: Compilation pipeline of the `rvsdg-treedc` framework.

4.2.1 Parser and data structures

The provided `rvsdg-treedc` framework contains a simple dotfile parser which loads graphs into a `C++` graph representation. The framework is made to be modular and extendible, supporting basic graph operations like Depth First Search and simplifying nodes.

The parsed graph is represented in memory as an adjacency list. This is the most space efficient representation for a sparse graph, *i.e.,* when the number of edges is less than n^2 for n nodes. It is also an efficient representation for exploring graphs since looking up a single vertex takes time proportional to its number of neighbors, but looking up subsequent neighbors can be done in constant time [20]. This is a common operation in the heuristic algorithms we implement.

The rows in the adjacency list are represented by a singly linked list, with each root node of the list stored in a `STL` vector. Each vector is contained in a graph object, and each element in the list is represented as a node object. These classes provide abstractions for searching and manipulating the graph. We assume that the graph is undirected and simple *i.e.*, has no parallel edges, to simplify the representation. This is the same generalization as in Diestel [13], where the tree decomposition is defined on such graphs.

[1] Inspired by Asbjørn Djupdals implementation in the `rvsdg-viewer` [3]

```c
int main() {
    int foo = 42;
    return foo;
}
```

```llvm
define i32 @main() {
    %1 = alloca i32
    store i32 42, i32* %1
    %2 = load i32, i32* %1
    ret i32 %2
}
```

```xml
<rvsdg>
    <region id="r93865840026912">
        <node id="n93865840083440" name="" type="lambda">
            <output id="o93865840091600"/>
            <region id="r93865840027200">
                <argument id="o93865840078384"/>
                ...
            </region>
        </node>
        <edge source="o93865840091600" target="i93865840079072"/>
        <result id="i93865840079072"/>
    </region>
</rvsdg>
```

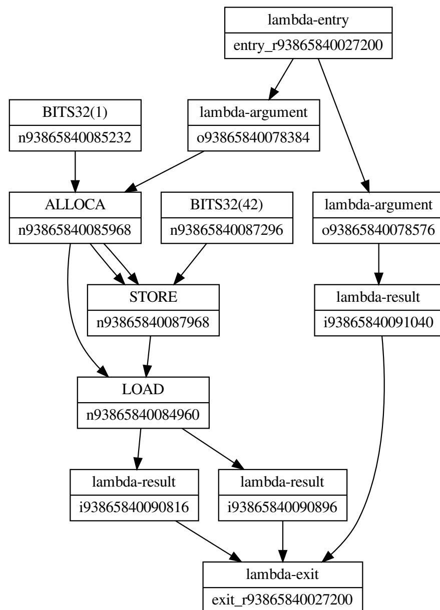


Figure 4.1: Example conversion of a C program, via LLVM bytecode, to an RVSDG XML representation and its corresponding dotfile.

These conditions are ensured by the dotfile-parser which ignores parallel edges and the potential direction of nodes between the edges parsed. The parser is a simple `C++` `stringstream` which handles a small subset the dotfile format. Specifically, it accepts a single graph of n nodes, with one edge per line and nodes numbered $0, 1, \ldots, n$. Otherwise it ignores any line containing the `node`, `label` or `rank` keywords.

Our chosen implementation language for these tasks is `C++`. It allows us to provide a level of abstraction above the simple adjacency list representation, while still maintaining the efficiency needed to represent and analyse graphs with large amounts of nodes.

4.2.2 Heuristics

The heuristics implemented in the framework is described in Section 3.3. Implementation is based on versions presented in Googate & Dechter [17], and a reference python implementation in the `D-Wave-NetworkX`-library [12].

4.3 Benchmarks

To generate a set of RVSDGs to measure the treewidth of, we use a set of programs found in the PolyBench benchmark suite [29] as input. The suite consists of 30 numerical computations from various domains such as linear algebra, statistics, physics simulations *etc*. The computations performed include matrix multiplications, covariance computation and LU decomposition. We chose PolyBench because of the small size and simple structure of the benchmark programs it contains. Since each program in the benchmark is structurally small, generation of XML representations of these programs are simplified. We use an existing fork of PolyBench [27], which contains support for compiling the benchmarks with the `jlm` compiler, and extend it to create the XML files required. The invocation of the compilation pipeline is shown in Listing 4.1

```
# LLVM bytecode is generated using clang 7.0.1
clang -S -emit-llvm -Xclang -disable-llvm-passes source.ll

# generate RVSDG-XML of the program using jlm-print
jlm-print --j2rx --file source.ll > generated.xml

# this output can then be parsed to a set of corresponding dotfiles
rvsdg-treedc/bin/xml_parser generated.xml

# and finally the heuristics for the treewidth can be run
rvsdg-treedc/bin/rvsdg-treedc
```

Listing 4.1: Invocation of the compilation pipeline.

4.4 Metrics

The heuristics we implement provide an upper and lower bound for the treewidth of the graphs they are applied to. We measure both, as the upper bound give a worst case indication of the actual treewidth, and the lower bound is used to measure the tightness of the

bound. Since we know that a lower bound will not be higher than the actual treewidth and that the upper bound will not be lower than the actual treewidth, the gap between these bounds measures how well the heuristic performs. The lower the gap is, the better the performance.

To make sure that our implementation is correct and that this assumption holds, we test the heuristics against a range of different graphs retrieved from the *ToTo* treewidth database. ToTo is an open graph database from Maastricht University that computes and stores tree computations of graphs. Graphs are represented in the *graph6* format [23] and stored along with the current best computation on its upper and lower bounds on treewidth, where the treewidths are found using a range of heuristics. Reported bounds are aggregated in the database such that users have access to the best reported bounds on graphs that are already generated. Users can also send in reports on these bounds, and submit better tree decompositions that improve on the heuristics results [39].

To test our heuristics with these graphs, we import the graph6 strings as a CSV file and generate tests using a set of scripts. The tests themselves are run using the `Unittest++` framework [24]. This process is pipelined, such that new graphs can easily be imported and tested. Candidates were chosen of varying size and gap, based on the selection of graphs with the most submitted results. All expected upper bounds were found, and no lower bounds were reported too high. From a total of 150 graphs tested, 9 of the lower bounds undershot the tightness bound by 1 for graphs with a small gap, and by 1–4 for graphs with larger gaps.

4.5 Optimizations

Running a source program with a different set of optimizations is a simple way to generate different versions of the program that are structurally different. These resulting targets have potentially varying treewidths, while preserving the same semantics. Using the programs in the PolyBench benchmark suite as a source, we generate versions of the programs for a set of different optimizations, and measure the treewidth of each result. In this section we will look at which optimizations are suitable to apply, and the limitations in analyzing the optimizations actual impact on the source code.

Deciding on which optimizations to apply, we first note the difference in the optimizations levels of the LLVM backend and its corresponding `C`-language family frontend, Clang. Clang also refers to the compiler driver that drives the phases of the compiler invocation, and sets the appropriate flags for the current build and system. The Clang driver thus invokes both the Clang frontend, also referred to as `cc1`, and the LLVM backend including the optimizer and assembler [40].

An advantage of invoking optimizations through the driver is that we can enable optimization diagnostics though LLVM. This enables storing and analyzing optimization remarks of the compilation process which can be used to further investigate which optimizations are enabled, their ordering, and other statistics. However, we identify three limitations of driver. Firstly, the analysis tool is limited, as not all LLVM passes emit such diagnostics [25]. The manual states:

"[...] do not expect a report from every transformation made by the compiler. Optimization remarks do not really make sense outside of the major transformations (*e.g.*,

inlining, vectorization, loop optimizations) and not every optimization pass supports this feature.” [36]

Testing this on the programs of the PolyBench suite, we found that this feature is only able to report on inlining, vectorization and global value numbering.

Secondly, invoking the optimizations through the driver adds a set of Clang specific options in addition to the optimization level flags [28]. Lastly, the Clang driver interacts with the front-end through an unstable developer only frontend [37], which does not allow setting individual optimization flags.

As we use the optimizations to create controlled changes in a program to investigate these changes effect on the RVSDG treewidth, we find it sufficient to invoke the optimizations though the modular LLVM optimizer and analyzer `clang-opt`, which does support invoking separate optimization passes.

We have chosen the optimizations used by the LLVM compiler running at optimization level `O2`. Level `O2` generates a list of optimizations that will have an effect on the code, while not including passes that might increase code size. This is the case of the optimizations added at level `O3` [35], which would make comparisons between the default and optimized programs more difficult.

This list of optimization passes can be found by invoking the LLVM optimizer with the `Arguments` specific debug pass. This is shown in Listing 4.2, inputting a random program generated with the LLVM assembler. We verify that the optimization passes printed by this command are the same, and are generated the same order as the optimizations actually applied when running the optimizer at level *O2*, by applying all 263 optimization passes individually. We confirm that this generates a program with the same treewidth and runtime.

```
llvm-as < /dev/null | opt -O2 -disable-output -debug-pass=Arguments
```

Listing 4.2: Finding the individual optimizations performed by LLVM at optimization level `O2`.

These optimizations, or *transform-passes*, are further documented in [30]. We also run the optimization passes supported by the *jlm-opt* RVSDG optimizer, such as common and dead node elimination.

The optimized LLVM bytecode and corresponding RVSDG XML representation is generated via the steps shown in Listing 4.3. This is an implementation of the first four steps shown in the pipeline in Figure 4.2. An optimization is either passed to the RVSDG optimizer, `jlm-opt`, or to the clang optimizer, `opt`.

```
clang -S -emit-llvm -Xclang -disable-llvm-passes -o program.ll ${INPUT_PROGRAM}
jlm-opt --llvm ${JLM_OPT} program.ll > program-jlm.ll
opt -S ${LLVM_OPT} -o program-opt.ll program-jlm.ll
jlm-print --j2rx --file program-opt.ll > rvsdg-program.xml
```

Listing 4.3: Generating the RVSDG XML and optimizing it from an input program.

Then, for each program, the upper and lower bound treewidth of the largest region in the program is calculated, and the program runtime is measured. These results are presented in Section 5.1.2.

4.6 Function arguments

Investigating the unoptimized programs in the PolyBench suite and their corresponding treewidths found in Section 5.1.1, we find that one source of the difference in treewidth between the programs is the number of arguments given to the main computational kernel of the program.

We inspect the lambda region modeling this computational kernel *i.e.*, the function which runs the numerical computation that is benchmarked. *E.g.*, for the `3mm` and `cholesky` benchmarks, we find that these regions have an upper treewidth bound of 5 and 4 respectively. One difference between these two functions is their number of parameters.

The `3mm`-kernel function accepts 13 arguments, while the `cholesky`-kernel function accepts 2 arguments. Changing the contents of these programs, *e.g.*, removing the computational contents of the kernel, results in the same treewidth in the generated program graph. We also find that changing the content of the `3mm`-kernel while maintaining the same number of accepted arguments, setting it to be equal to the `cholesky`-kernel, still results in it having the same upper treewidth bound of 5.

Thus, the number of variables passed to a function directly affects the treewidth of the resulting program graph, and this behavior can be isolated. We will further investigate how function parameters affect the treewidth. This is done by creating a set of test programs, each containing a single function summarizing all arguments passed to it. We then look at the three different ways of passing arguments to this function defined in Section 3.4.1.

This results in three programs with the same semantics, but different program structures and different resulting RVSDGs. The programs we test and measure the treewidth of, consists of a main function setting up the necessary variables and calling one of these three functions respectively.

In Section 5.1.4 we present the treewidth bounds of these programs and overview how these different methods of passing arguments to functions affects the structure of their corresponding RVSDGs. This will be investigated in depth in Section 5.2.2, where we discuss how the resulting dependencies between instructions in the LLVM IR affects the graph structure and treewidth of the RVSDG.

4.7 Variable Liveness

In Section 3.4.2 we introduce variable liveness with respect to the data and state dependencies in the RVSDG. Looking at how the allocation and liveness of variables can impact the treewidth, we use the `3mm` and `cholesky` benchmarks as examples. These benchmarks are at opposite ends of the treewidth bounds found for the PolyBench benchmark programs, `Cholesky` being comparatively high and `3mm` being comparatively low.

Matrices in the PolyBench benchmark suite are represented as two dimensional C arrays, *i.e.*, an array containing pointers to arrays for each row in the matrix. An example allocation is shown in Listing 4.4.

```
float (*mat)[N][M] = (float(*)[N][M])polybench_malloc(N*M, sizeof(float));
```

Listing 4.4: Allocation of a PolyBench style matrix of dimension $N \times M$.

The 3mm benchmark consists of three matrix-matrix multiplications, $(A \times B) \times (C \times D)$, which uses two temporary matrices for the calculation of $(A \times B)$ and $(C \times D)$ respectively, and one matrix for storing the final result. This requires a total of 7 matrices that needs to be allocated and deallocated, with 4 matrices being initialized in a separate function, and 5 integer variables to keep track of the dimensions. Comparably, the `cholesky` benchmark performs a *Cholesky Decomposition* which involves the decomposition of a single square matrix with computations happening in-place. This benchmark thus only requires the allocation of a single matrix and a single integer variable to keep track of the dimensions.

Inspired by the PolyBench benchmark programs, we write a set of custom programs to induce the features that affect the RVSDG treewidth. These effects are then analyzed with respect to liveness, specifically in regards to what kinds and how many variables are allocated, and how they are referenced in the program. We present the results of these investigations into the effects of the liveness, allocation, and the referencing of variables in Section 5.1.5. A detailed discussion of these results follow in Section 5.2.3.

Chapter 5

Results & Discussion

In this chapter, we first present the general results of running the implemented heuristics for the lower and upper bound treewidths on the benchmarks of the PolyBench suite. In Section 5.1 we further present the treewidth results of the program feature investigations, including optimization passes and their relation to program runtime, function parameters and call orders, and finally variable liveness and allocation.

In Section 5.2 we discuss these results. First we look at the general treewidth results and relate them to other findings in the field. We then present an analysis of the program feature impact on the resulting treewidths. This is done by analyzing the function and liveness analysis results impact on the generated graphs.

5.1 Results

In Section 5.1.1 we present the result of the maximum upper bound and treewidth gap of the different benchmarks and compare this to the number of nodes in each region.

Next, we look at the results from our program feature investigations. Section 5.1.2 presents treewidth and timing results for different optimizations. We show all results for two select benchmarks, and aggregated results for all. These results show that while there is no one-to-one correlation between the lower runtime of an optimization in a benchmark and its resulting treewidth, the collections of optimizations have a lower runtime *and* lower treewidth for a majority of benchmarks. Section 5.1.3 further investigates this relationship between runtime and treewidth, further demonstrating the lack of a clear-cut correlation between them.

Finally we show the results from our custom programs, inducing different program features. Section 5.1.4 presents the function analysis results and Section 5.1.5 presents the variable liveness and allocation results.

5.1.1 General treewidth results

We present results from running our heuristic algorithms on the programs in the PolyBench benchmark suite. Each program generates on average 54 RVSDG regions, for a total of 1620 regions. For each graph corresponding to these regions, we calculate the upper and lower heuristic bound on the graphs treewidth.

In Figure 5.1, we show the maximum upper bound treewidth found using the min-fill heuristic for all regions in the benchmark. We find that all benchmarks have an upper bound treewidth between 6 and 15 with an average upper bound treewidth of 9. The largest treewidth gap found, using the minor-min-width heuristic for lower bound, for all regions in the benchmarks is shown in Figure 5.2. The numbers above each bar denote the upper and lower treewidth bound respectively. From this figure we can see that the largest gap is 9 while 86% of the benchmarks, have a gap between 1 and 4. The tightness of these gaps means that our upper bound heuristic is a close approximation of the actual treewidth of the graphs measured.

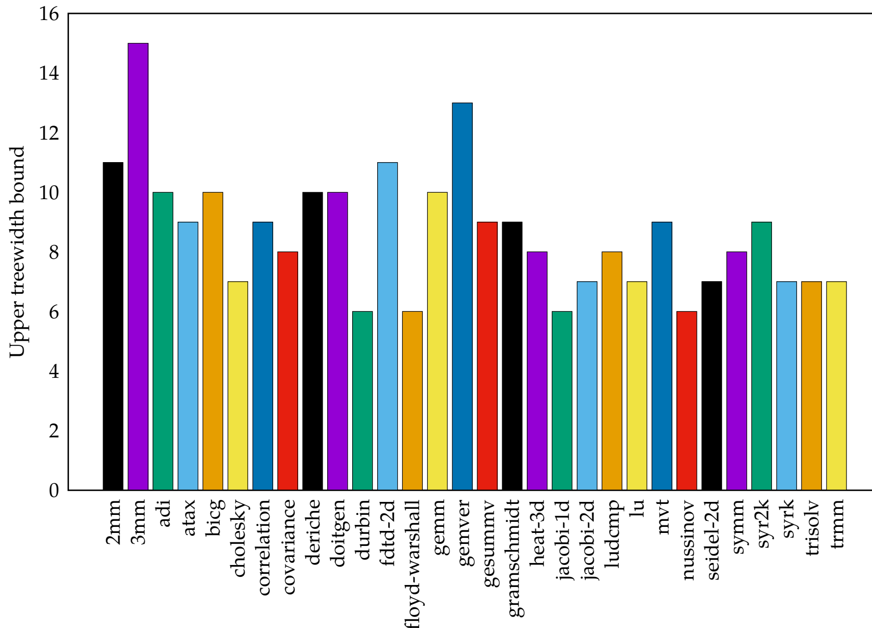


Figure 5.1: Upper bound treewidth per benchmark.

Figure 5.3 show the relationship between the number of nodes and heuristic results for graphs generated from all regions in the benchmark programs. Figure 5.3a shows that as the upper treewidth bound grows, the amount of nodes in the graph increases on average. We see the same trend for the treewidth gap in Figure 5.3b.

In Figure 5.4 we show how the average number of nodes relate to the upper treewidth bound and treewidth gap respectively. We see that as the upper treewidth bound and treewidth gap increases, the average number of nodes in the corresponding regions increases approximately polynomially.

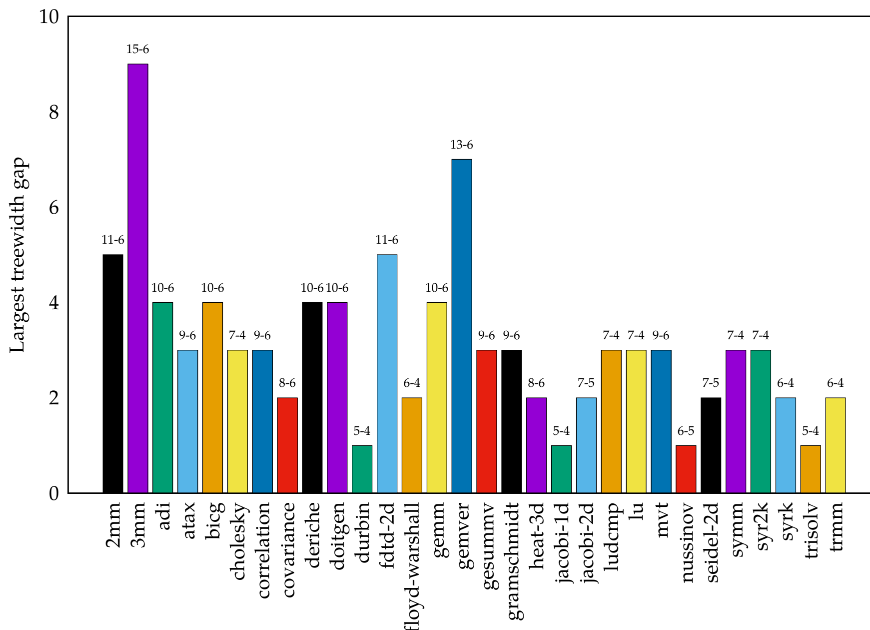


Figure 5.2: Treewidth bound gap per benchmark.

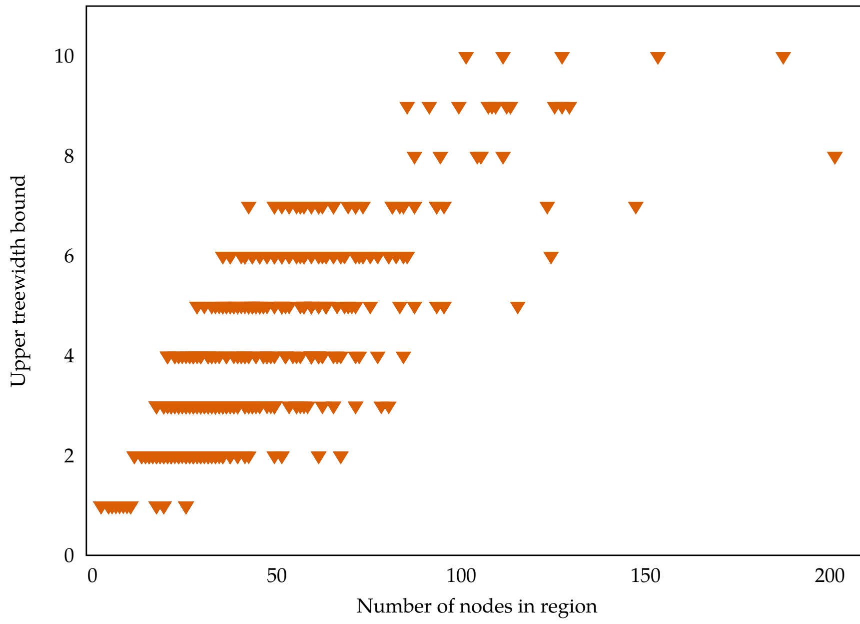
This is demonstrated by fitting a polynomial curve to each figure. Figure 5.4a shows that the number of nodes in the graph increases polynomially as a function of the upper treewidth of the graph. This relationship is approximated by the function $n = 20 + 1.3tw^2$ for the number of nodes n, and treewidth tw. Similarly, Figure 5.4b shows that the number of nodes also increases polynomially as a function of the largest treewidth gap g, given by $n = 50 + g^3$.

We note that of all 1620 regions generated only 4 regions have a treewidth above 10, marked as $11+$ in Figure 5.4a. Similarly, 4 regions have a treewidth gap above 4, marked as $5+$ in Figure 5.4b.

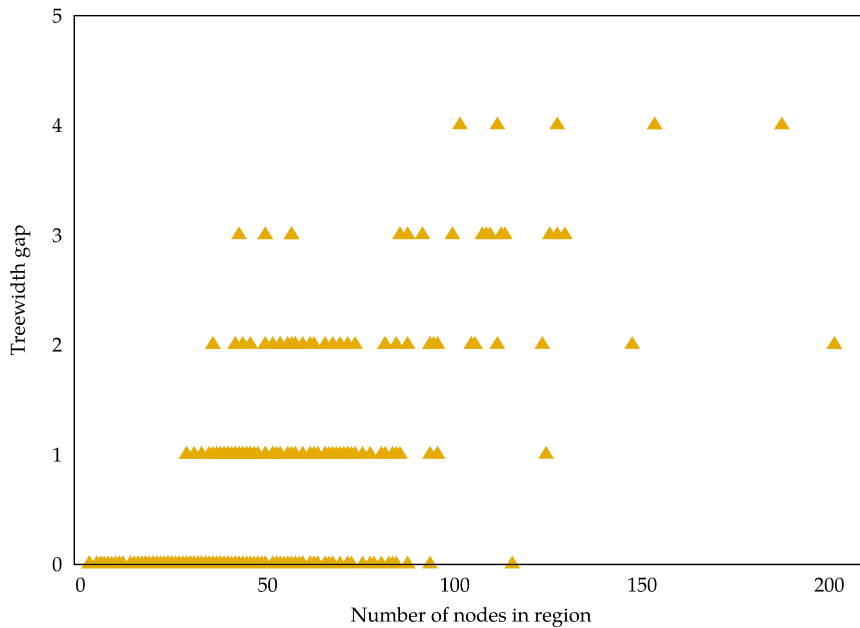
5.1.2 Optimizations

In this section we present the results of the individual optimizations applied to the programs of the PolyBench benchmark suite as described in Section 4.5. We first show the upper bound heuristic results, presenting plots from select benchmarks, demonstrating the effects from the full suite of optimizations. We also include the `O1` and `O2` optimization levels for all benchmarks, and compare the upper treewidth bound generated by these collections of optimizations to the treewidths generated by the optimizations individually.

We then present findings for the lower bound treewidth heuristics in the same fashion, and finally an in depth analysis of the `O2` optimization level process with respect to its effects on the treewidth.

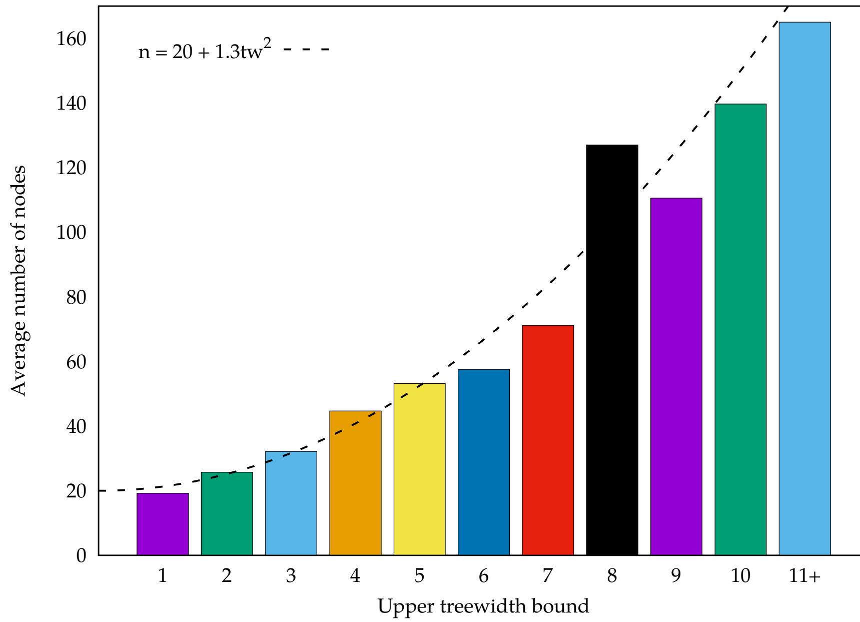


(a)

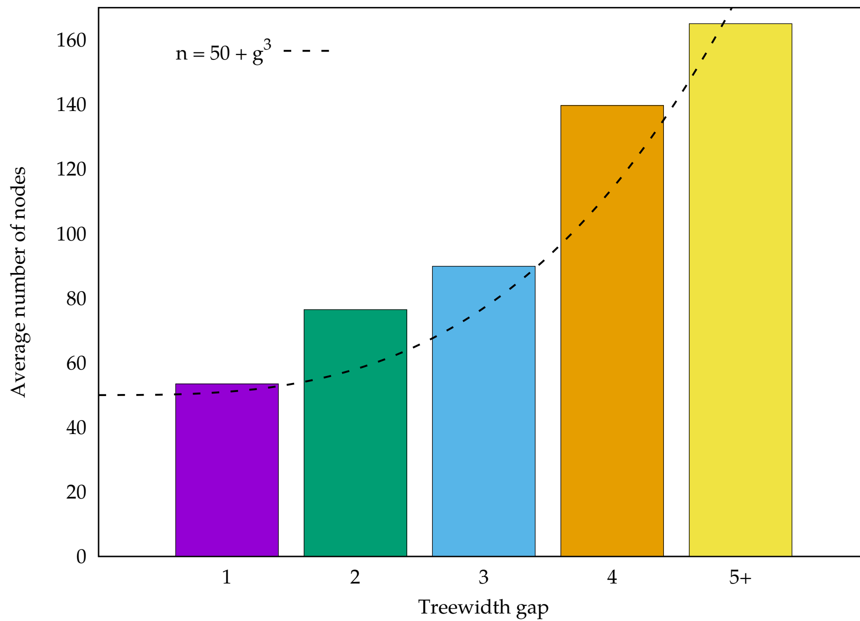


(b)

Figure 5.3: Upper treewidth bound and treewidth gap per number of nodes in region.



(a)



(b)

Figure 5.4: Average number of nodes per upper treewidth bound and treewidth gap.

Upper bound treewidth results

Results for the upper bound treewidth for the `atax` and `seidel-2d` benchmarks are shown in figures 5.5 and 5.6 respectively. From these plots we see that firstly, there is no clear correlation between the treewidth and runtime of the programs. Secondly, we see that a small subset of the different optimizations have an effect on the treewidth of the program itself. Although there is some relationship between which optimization passes lead to a higher and/or lower treewidth, this correlation not one-to-one.

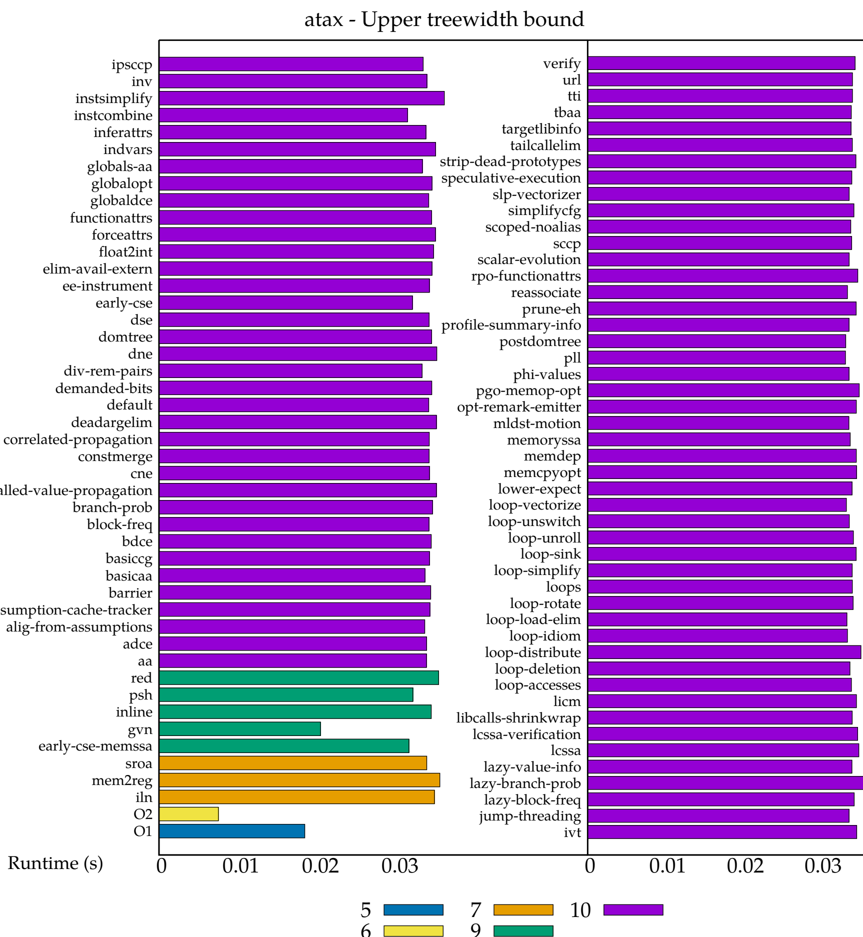


Figure 5.5: Relationship between the upper bound treewidth, as different colored bars, and runtime for the `atax` benchmark for a set of optimizations.

We *e.g.*, see that the `early-cse` optimization leads to a higher treewidth in both benchmarks, but while the `early-cse-memssa` generates a higher treewidth relative to the other optimizations for the `seidel-2d` benchmark, it generates a treewidth in the intermediate range for the `atax` benchmark.

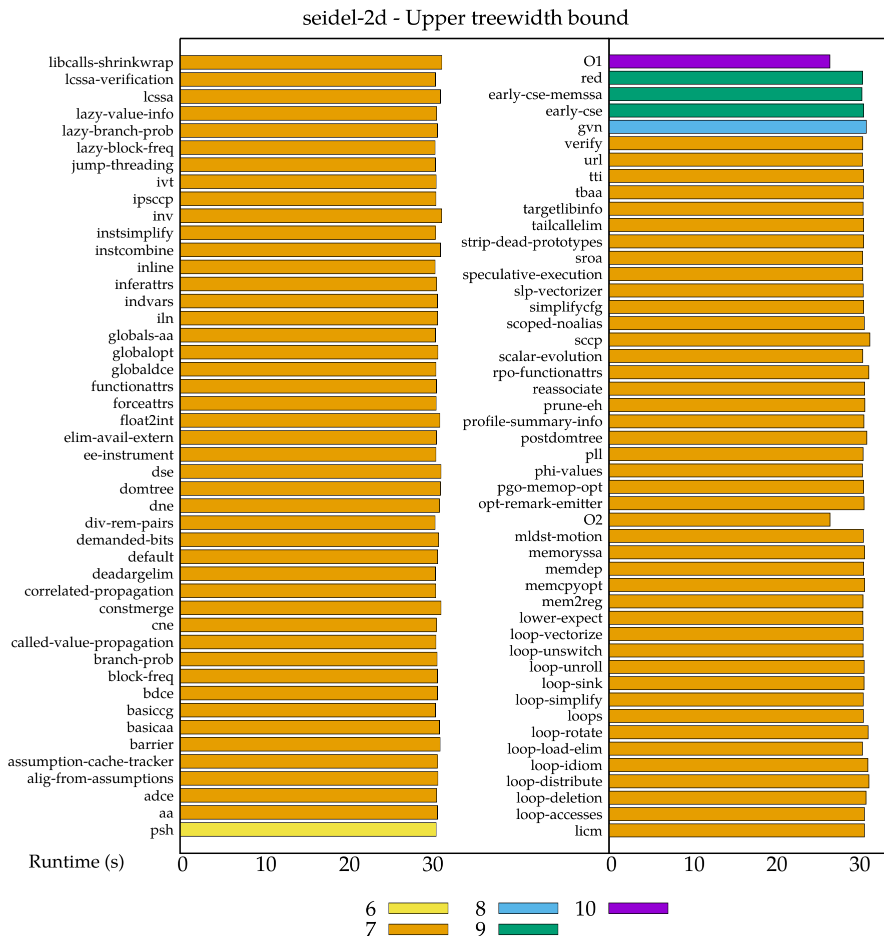


Figure 5.6: Relationship between the upper bound treewidth, as different colored bars, and runtime for the `seidel-2d` benchmark for a set of optimizations.

In addition to the individual optimizations, we also measure the treewidth and timing results of the programs compiled with the collection of optimizations found in level `O1` and `O2` respectively. As expected these programs have lower runtimes than the programs compiled with only individual optimizations.

In the generated plots we see that for a large number of benchmarks, the optimization levels also generates programs with smaller upper bound treewidths relative to the individual `jlm` or `llvm` optimizations. We see this in the `atax` benchmark in Figure 5.5 where the `O1` and `O2` optimizations generate the lowest treewidths. The `seidel-2d` benchmark does however show that this is not true for all programs, with level `O2` generating a treewidth of average size relative to the individual optimizations, and level `O1` generating a relatively higher treewidth.

We find the upper bound treewidth for the benchmarks when optimized at level `O1` or `O2` and compare these to the treewidths generated from the optimizations individually. These results are summarized in Table 5.1, showing that level `O1` generates a minimum upper bound treewidth for 73% of the programs, and level `O2` generates a minimum upper bound treewidth for 59% of the programs. When neither a relative upper or lower treewidth bound is generated, the result is listed as intermediate. The full table of results is found in Appendix A.1.

	total	lower	upper	intermediate
O1	30	22 (73%)	3 (10%)	5 (17%)
O2	29	17 (59%)	5 (17%)	7 (24%)

Table 5.1: Aggregated upper treewidth bound results for the `O1` and `O2` optimization levels. Shows a summary of the number of benchmarks where the programs generate a minimum, maximum, or intermediate treewidth compared to the other treewidths found when using individual optimizations. Full table found in Appendix A.1.

Lower bound treewidth results

Measuring the lower bound treewidth compared to the benchmark runtime, we again show these results for the `atax` and `seidel-2d` benchmarks. These results are presented in figures 5.7 and 5.8 respectively.

The lower bound treewidth measurements give similar results to the upper bound treewidth case. Some optimizations affect the resulting treewidth, but this is not necessarily the same optimizations between benchmarks. *E.g.*, `early-cse-memssa` does not affect the treewidth of the `atax` benchmark, but it does generate lower treewidth bound for the `seidel-2d` benchmark. We again note the lack of a relationship between the runtime of the individual benchmarks runs and their corresponding lower treewidth bound.

We do however find a relationship between the lower treewidth bound and a lower runtime of the program when optimized at levels `O1` and `O2`, as with the upper bound treewidth.

We compare the lower treewidth bound generated by the optimization levels to the maximum and minimum lower treewidth bounds found for the program when compiled with individual optimizations. In Table 5.2 we summarize these results, again splitting the results into three categories for generating the relative lower, upper, or intermediate treewidth bound.

	total	lower	upper	intermediate
O1	30	23 (76%)	3 (10%)	4 (13%)
O2	29	23 (80%)	3 (10%)	5 (17%)

Table 5.2: Aggregated lower treewidth bound results for the `O1` and `O2` optimization levels. Summary of Table A.2, in similar fashion to Table 5.1.

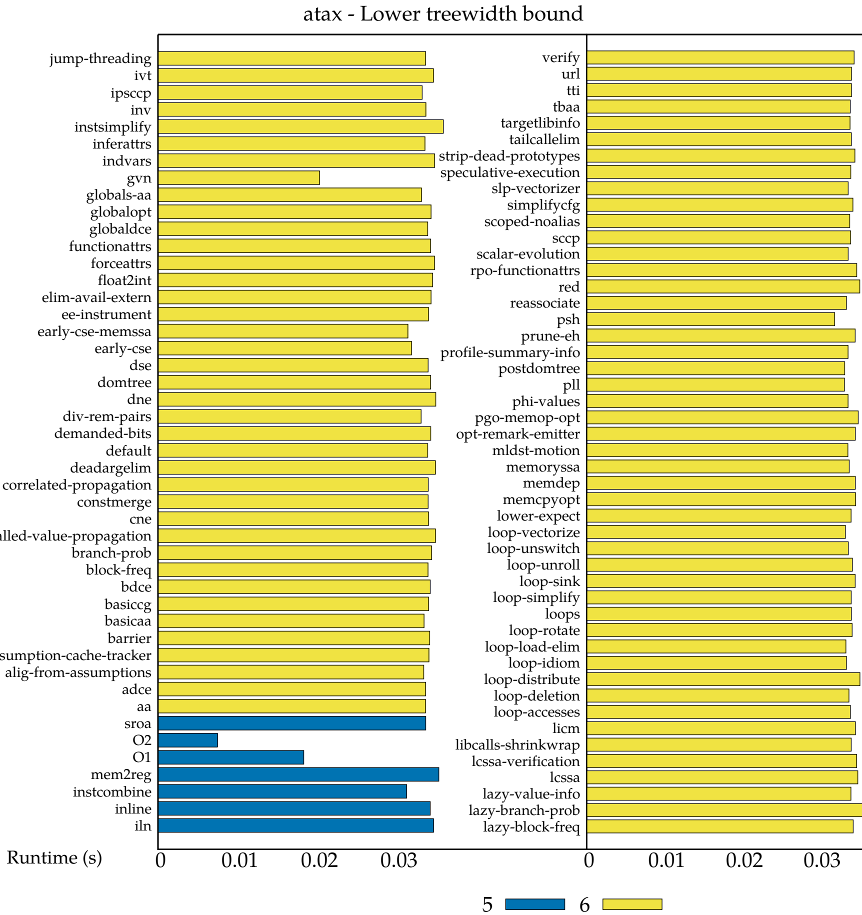


Figure 5.7: Relationship between the lower bound treewidth, as different colored bars, and runtime for the `atax` benchmark for a set of optimizations.

We note that there is a smaller variety in the lower treewidth bound than in the upper treewidth bound, with some benchmarks having the same lower bound for all optimizations. In these cases we have reported the optimization to have found a lower relative treewidth. We find that a similar number of `O1` optimized benchmarks, 76%, and that a greater number of `O2` optimized benchmarks, 80%, generate a lower treewidth bound. The full table of results can be found in Appendix A.2.

Aggregating the treewidth results

Plots from other benchmark programs than *atax* and *seidel-2d*, as well as similar investigations measuring average treewidth of these programs also show the lack of correlation between the runtime of the program and the treewidth of the individual optimizations. Thus

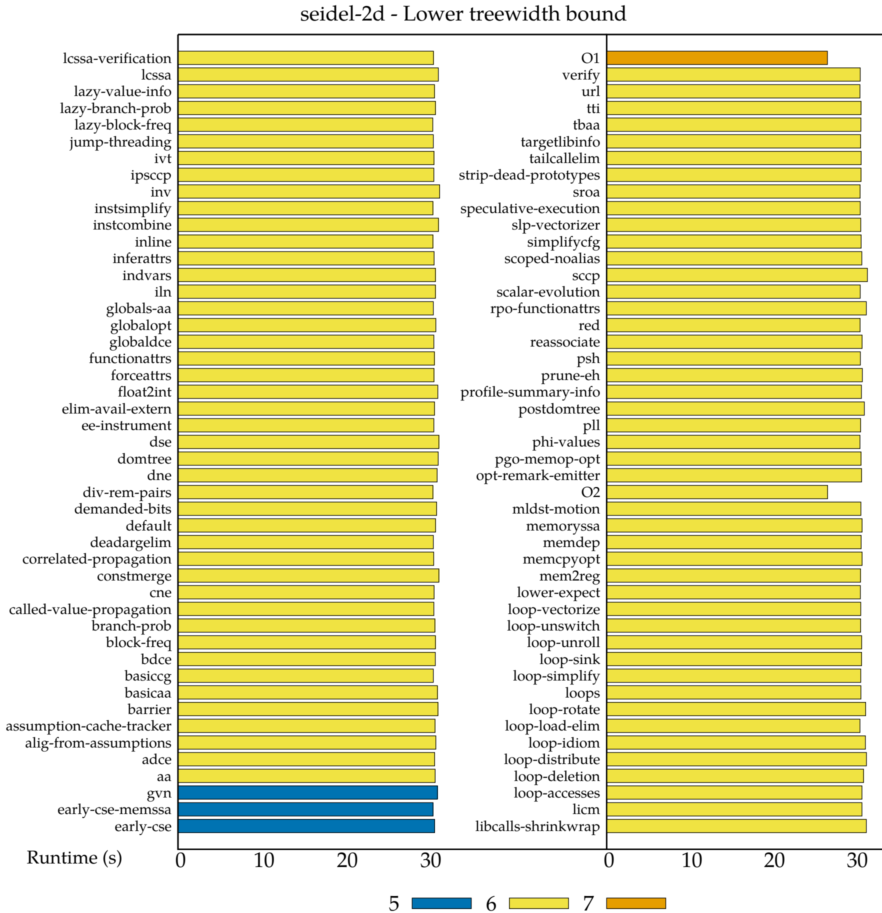


Figure 5.8: Relationship between the lower bound treewidth, as different colored bars, and runtime for the `seidel-2d` benchmark for a set of optimizations.

there is no general observable relationship between the RVSDG upper treewidth bound and the efficiency of the programs itself. This is further investigated in Section 5.1.3. Here where we create two semantically equivalent programs with different treewidths, still resulting in a similar runtime of the programs.

In Figure 5.9 we compare the results of the lower and upper treewidth measurements. For each benchmark we show the relative results of both optimization levels. We see that for most benchmarks, the same or a similar treewidth is found for both upper and lower bounds, and both optimization levels. Only for the `durbin` and `jacobi-2d` benchmarks, is the lower treewidth bound relatively high while the upper treewidth bound is relatively low for the same optimization level. We also find that the `nussinov` benchmark is the only program where the `O1` and `O2` levels generate the lowest and highest

relative treewidths respectively.

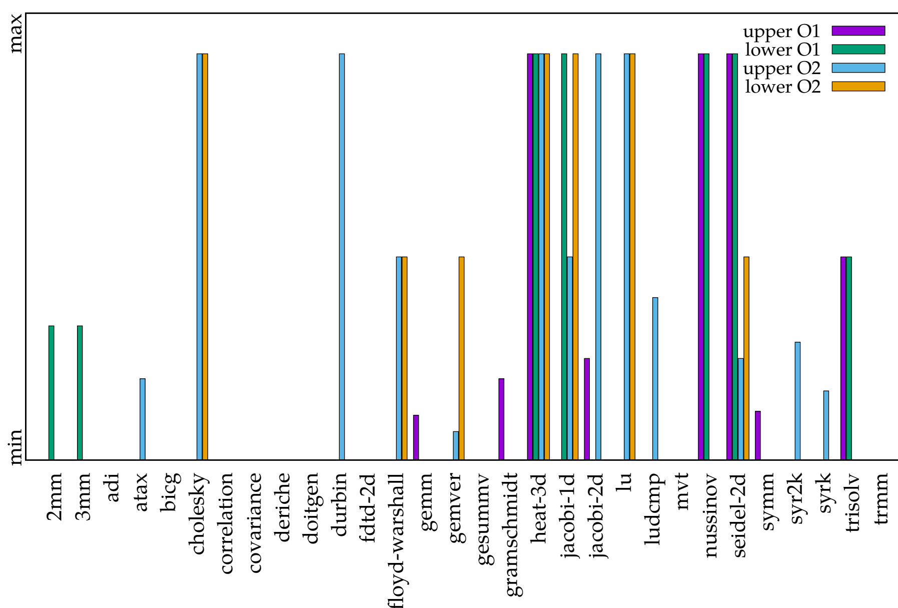


Figure 5.9: Comparison of relative treewidths generated for the upper and lower treewidth bounds at the `O1` and `O2` optimization levels. Treewidth results are listed as minimum, maximum, or intermediate.

Optimization level *O2*

To investigate which optimizations affect the treewidth when running at level *O2* we apply each optimization pass individually, as discussed in 4.5, and measure the treewidth of the program at each intermediate step. We find that most of the passes do *not* change the treewidth of the program. In Figure 5.10 we summarize these results, showing only the passes that have an effect on the treewidth. As the transform passes run in a particular order, and some optimizations are applied several times, we also list at which step of the total 263 passes the change occurs.

From this figure we see that of all passes, only 22 actually affect the treewidth of the resulting program. These optimizations are run on 30 different programs, meaning that only 10 of the passes changes the treewidth for approximately 50% or more of the programs. We note that the *Scalar Replacement of Aggregates* (SRA), `-sroa` optimization pass changes the treewidth of 17 programs in the first pass and 28 in the second pass. SRA introduces new variables for parts of non-aliased aggregates, potentially replacing memory load dependencies [18]. We note that for almost all of these changes, the treewidth *decreases*.

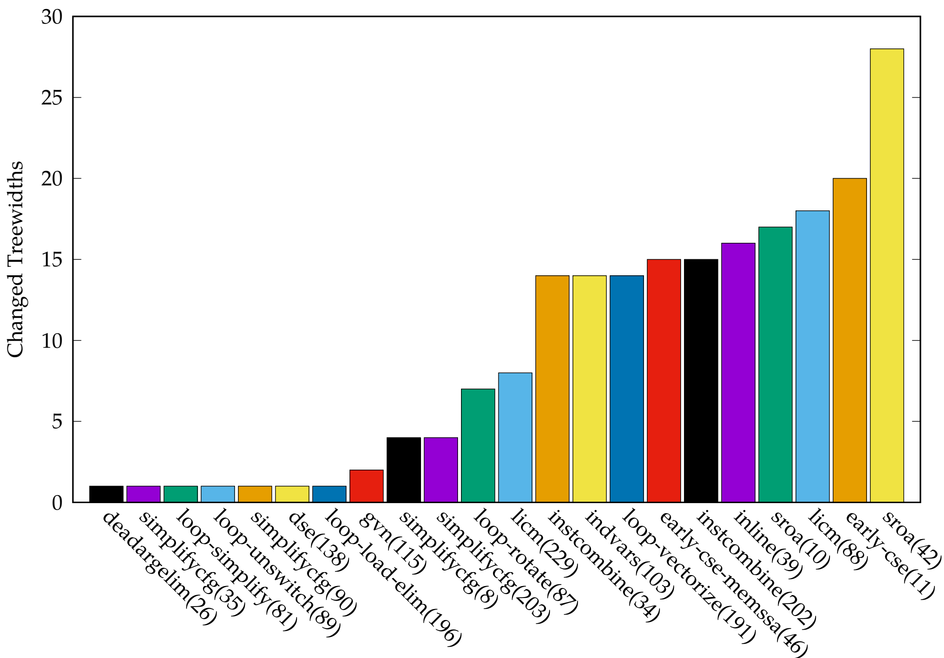


Figure 5.10: Optimization passes that change the treewidth for the programs in the PolyBench benchmark suite. For a total of 30 programs from the suite and all optimization passes applied in the level `O2` optimization process, the figure shown the number of optimization passes that results in a changed treewidth (at step x).

5.1.3 Impact on program runtime

To investigate the treewidth impact on runtime, we create a custom benchmark performing some arbitrary calculations found in the benchmarks of the PolyBench suite, such as matrix transposition and vector addition on large matrices. We create two programs performing the same calculations, only differing in the method that arguments are passed between the functions of the programs such that the treewidth differs as expected between them.

These programs are benchmarked, and the results indicate no change in the runtime between the program that passes the matrices as values to the program that passes the matrices as members of a struct. This indicates the lack of a relationship between the treewidth itself and the runtime efficiency of the generated program, further supporting our findings about the lack of such a relation as discussed above.

5.1.4 Functions

This section presents the different treewidth bounds generated by running three semantically equivalent programs, each loading values using the different methods presented in Section 3.4.1. These results are summarized in Table 5.3, run for functions with 10 parameters. We also give an overview of how these methods affects the structure of the

corresponding RVSDG.

type	tw	Graph structure	Loading of values
variable	4–7	Sequential, dependent on loading of values from memory	Sequential, each argument must be allocated on the stack. Each such allocation is dependent on the allocation of the previous argument.
struct	4–4	Parallel	Parallel, each argument is retrieved via a pointer to the struct, which happens independently of each other.
array	4–4	Partially parallel	Similar to loading of the struct, except that each pointer is dependent on the previous being loaded.

Table 5.3: Summary of the treewidths correlating to the separate methods of passing arguments to the functions in Listing 3.3, with notes about the structure of these graphs and how values are loaded. The *tw*-column shows the lower and upper bound treewidth of the resulting lambda region generated for the program.

A function accepting a single variable, performing the same summary ten times has a treewidth of 4. Therefore, the struct and array methods of passing arguments does *not* increase the treewidth. However, for a variable amount of arguments, increasing amount of of arguments the function accepts results in a higher upper treewidth bound for the corresponding lambda region. We note that this relationship is *not* one-to-one between the number of arguments accepted by the function and the upper bound treewidth.

In Figure 5.11 we show the results of testing a function accepting n number of arguments and returning their sum.

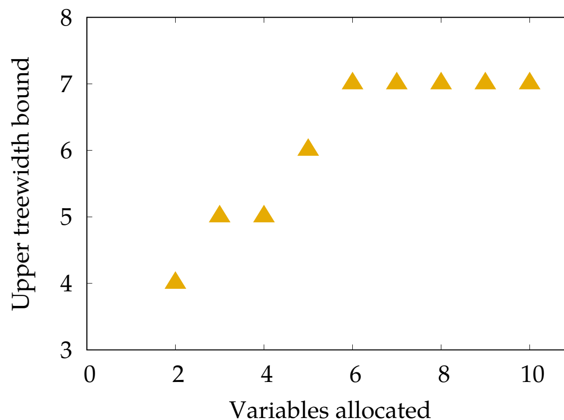


Figure 5.11: Relationship between number of arguments passed and the upper bound treewidth of region representing the `variable_sum()` function.

We see a gradual increase in the treewidth, although not exactly linear, with the increase in the number of arguments accepted. We also note that this increase stops after a certain number of arguments accepted is reached, in this case the upper bound is 7 variables allocated.

From these experiments we also observe two other factors that affect the treewidth of the function. Firstly, allocating other variables inside the function *may* increase the upper bound treewidth. Secondly, changing the addition expression inside the function itself also affects the upper bound treewidth. The results of investigating these factors are presented in Section 5.1.5.

5.1.5 Variable liveness

Running the experiments for the type and number of arguments of the functions discussed above, while we notice some changes in the treewidth of the lambda region representing the functions themselves, we find a comparably larger variation in the treewidth of the main function calling the various summary functions. This section presents our findings for how the variable liveness, as introduced in Section 4.7, affects the treewidths of these main functions.

We first present the treewidth results for a varying amount of variable allocations, then the results for a varying number of references via function calls to each variable. We further present results for different call orderings, and finally variable use in expressions and its effect on the treewidth.

Single variable allocations

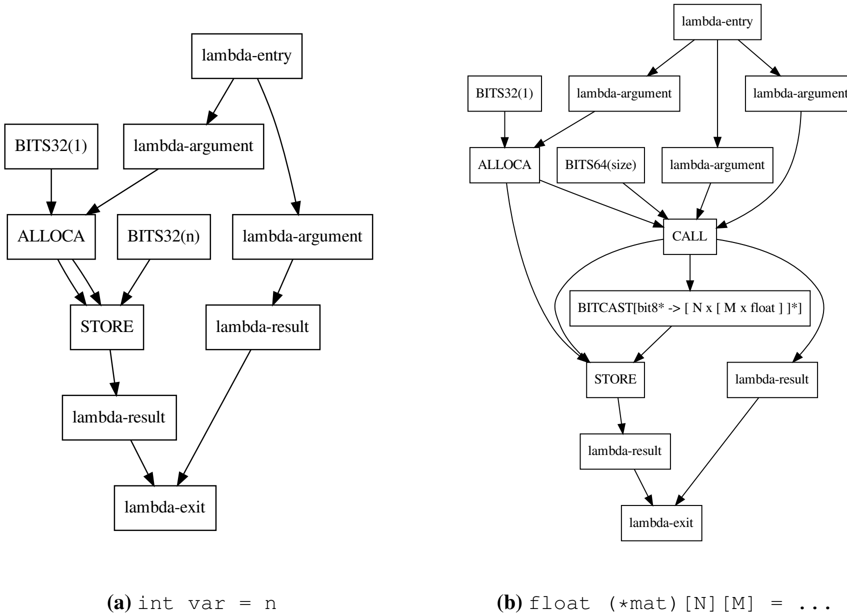
Again using the `3mm` and `cholesky` benchmarks as examples, the main function in the program calling the computational kernel of these two benchmarks has an upper bound treewidth of 13 and 6 respectively. The only difference between these main functions is the number of integer variables being allocated and matrices being allocated and deallocated.

We use the specifications of these programs as a basis for investigating the effect of variable liveness on the RVSDG treewidth, inspecting the graphs generated by two programs created to induce these allocation differences discussed above. These programs allocate the same number of variables with the same live range, one using standard integer allocations and the other using PolyBench style matrix allocations as shown Listing 4.4.

In Figure 5.12 we show the RVSDGs generated from allocating a single integer variable and a single matrix respectively. We notice that the matrix allocation in 5.12b results in a similar graph structure to the integer variable-case allocation in 5.12a, except that an additional call to the PolyBench allocation function is required, resulting in an additional state edge from the lambda entry node to the function call.

Multiple variable allocation

We find that if the variable is not referenced later in the program, allocating it has no effect on the treewidth of the program after the treewidth reaches certain limit. For the two programs in Figure 5.12, this upper treewidth bound is 4 and the lower treewidth bound is



(a) `int var = n` **(b)** `float (*mat)[N][M] = ...`

Figure 5.12: Allocating and storing a single integer and matrix respectively.

3 for 5.12a and 4 for 5.12b. The number of allocations needed to reach this upper limit is 4 integer allocations or 3 matrix allocations.

Variable allocations with single call

Analyzing the allocated variables that are used or referenced *once* in the same function that they are allocated, as shown in Listing 5.1, we find that this use of the variable increases the treewidth beyond the maximum value discussed above. This growth in the treewidth can be seen in Figure 5.13.

We note that the *upper* treewidth growth increases constantly as we add and use more integer variables. The *lower* treewidth bound however, grows slower. As discussed, for the case of only allocating the variables, allocating *and* calling the variables once also tends towards an upper bound. This can be seen by the reduced growth of the treewidth in Figure 5.13 from 7 and 6 integer and matrix variables allocated respectively. As above, the end of this growth is verified using up to 100 allocations and calls.

Variable allocations with multiple calls

When calling the allocated matrices more than once, as shown in Listing 5.2 we find that the treewidth increases further above the results shown in Figure 5.13. *E.g.*, if the integer variables are called twice by a function, the upper bound treewidth increases by 2 and the

```c
int var_1 = 1;
int var_2 = 2;
...
int var_n = n;

f(var_1);
f(var_2);
...
f(var_n);
```

Listing 5.1: Called once.

```c
int var_1 = 1;
int var_2 = 2;
...
int var_n = n;

f(var_1);
f(var_2);
...
f(var_n);

...

f(var_1);
f(var_2);
...
f(var_n);
```

Listing 5.2: Called several times.

```c
int var_1 = 1;
int var_2 = 2;
...
int var_n = n;

f_1(var_1);
f_2(var_2);
...
f_n(var_n);

...

f_1(var_1);
f_2(var_2);
...
f_n(var_n);
```

Listing 5.3: Called several times by different functions.

n integer variables allocated and called in a blockwise order.

lower bound increases by 1. We also find that passing the variables to *different* functions, as shown in Listing 5.3, increases the treewidth further. Passing the integer variables once to two different functions, the upper bound further increases by 1. This new upper limit is also verified on programs for allocating up to 100 variables in the same manner.

Figure 5.14 shows this continued growth in treewidth for a program allocating 7 variables for an increasing number of function calls referencing each variable. This figure also shows the difference in the treewidth dependent on whether these function calls are made to the same function, or to 7 different functions. If the variables are called by the same function 3 times, the lower bound further increases by one, continuing almost correspondingly until a new upper limit for the bounds is reached.

We see a slightly different growth in the treewidth if we instead call a different function each time the variable is referenced. In this case, the lower treewidth bound grows at a similar rate while the upper treewidth bound increases in bigger increments. Calling different functions, the treewidth also continues to grow for an increasing amount of function calls, resulting in a larger upper and lower treewidth bound.

We also note that when calling several different functions, the upper treewidth bound in some cases *decreases* when adding more calls *e.g.*, from 6 to 7 calls in 5.14a, or from 5 to 6 calls in 5.14b.

Variable allocations with multiple calls with different orderings

Further looking into the relationship between the treewidth and the number of times the variables are called, we find that the ordering of these calls impact the resulting treewidth. In Section 3.4.1 we define two different orderings, blockwise and sequential. The results

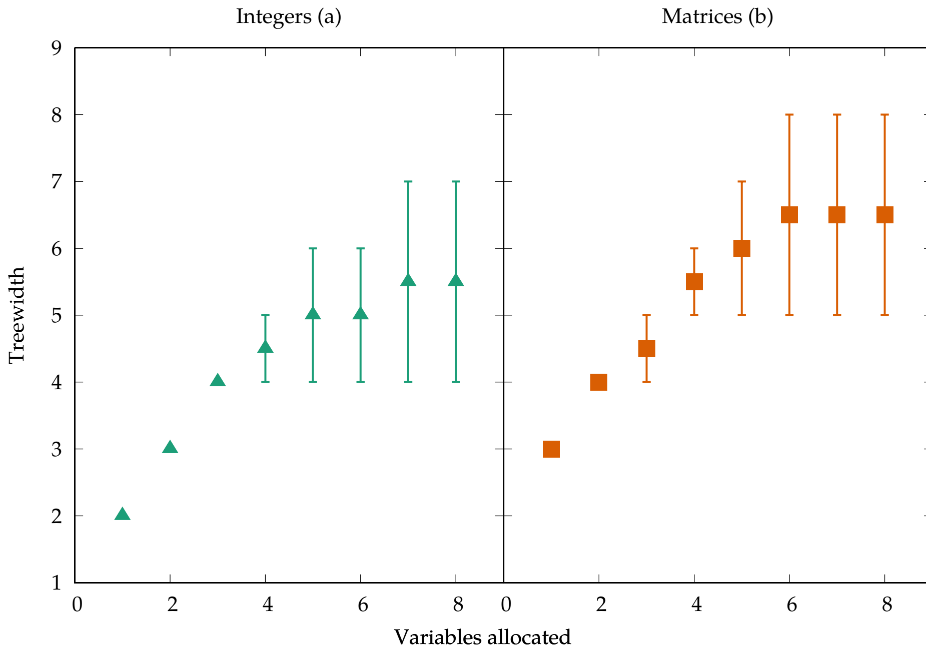


Figure 5.13: Relationship between the number of variables allocated and referenced once, and the upper and lower treewidth bounds of their corresponding RVSDGs.

presented in the previous section call functions in a blockwise order, and in this section we similarly present results for sequentially ordered function calls.

As above, we allocate 7 variables calling an increasing number of functions, testing both the same and different function calls. We find that unlike the blockwise call order, the sequential ordering results in *no increase in treewidth as the number of function calls increases*. This holds true for both integer and matrix variables, and for calling both the same and different functions.

In all cases, the resulting treewidth is the same as for one call in the blockwise call order. These results are also verified for up to at least 100 calls.

Multiple variable allocations and multiple references and calls

Trying to combine the two approaches presented in this section, increasing both the amount of variables allocated *and* the number of times they are called at the same time, does not tend toward an upper bound as when increasing one of the parameters individually.

An excerpt from these results are shown in Figure 5.15a, where we increase both the amount of variables called, and the number of times they are called. The complete table of results can be found in Appendix A.3. We find that increasing either parameter, the treewidth will grow to a certain limit, but when increasing both the treewidth grows indefinitely.

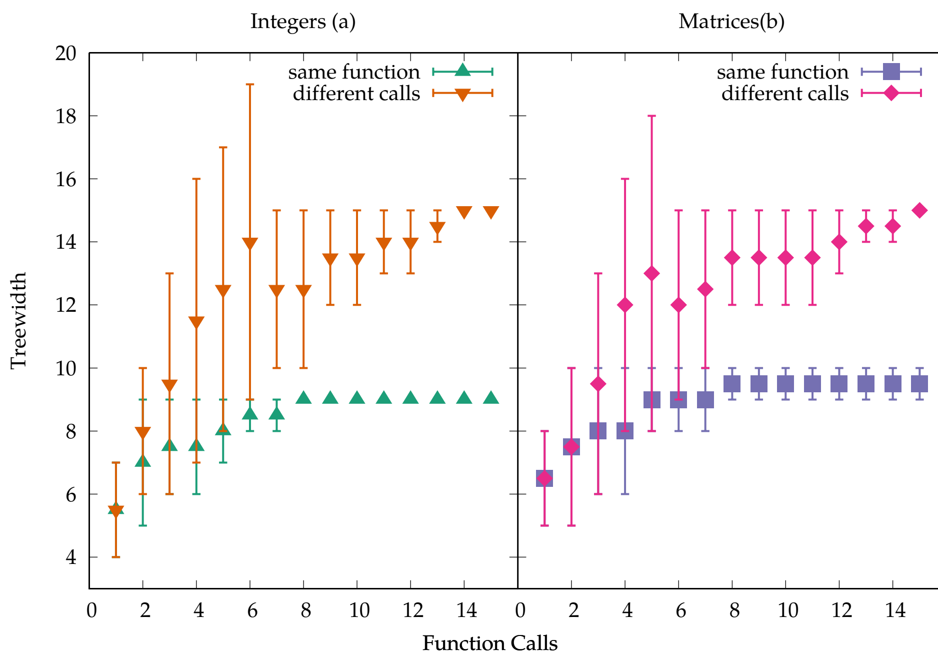


Figure 5.14: Relationship between the number of variables allocated in a blockwise call order referenced 7 times and the upper and lower treewidth bounds of their corresponding RVSDGs.

Variable allocations and use in expressions

Further investigating how the number of variables allocated and how many times they are used in expressions affect the treewidth, we run a set of tests in the same manner as above. These tests vary both the number of variables allocated and the number of times they are using in an expression. The expression is an addition of all the allocated variables. For several variables this is done in a blockwise call order. These results are presented in Figure 5.15b, showing the lower and upper treewidth bounds generated for these programs. Figure 5.15b shows the results when increasing both parameters, the complete table of results can be found in Appendix A.4.

We find similar results to increasing the number function calls. When increasing both parameters, the treewidth of the resulting graphs increases without approaching an upper bound. Table 5.15b also shows that variables referenced in expressions results in smaller treewidths than passing them to function calls.

5.2 Discussion

In this section we discuss and analyse the results of our experiments presented above. Section 5.2.1 discusses the general results we find from calculating the upper and lower treewidth bounds for the RVSDG regions generated by the programs in the PolyBench

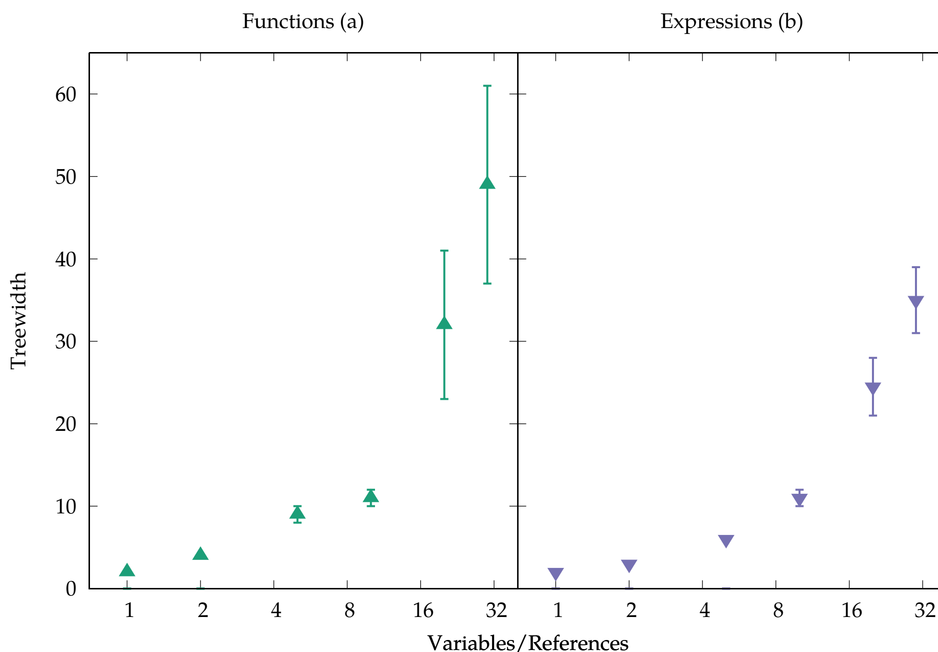


Figure 5.15: Lower and upper treewidth bounds for both an increasing amount of variables allocated, and number of calls to each variable.

benchmark suite. Section 5.2.2 analyses how dependencies in functions affect the RVSDGs generated and their corresponding treewidths. Finally, Section 5.2.3 analyses the liveness results in the same manner, discussing how the dependencies introduced by liveness restrictions affects the generated graphs.

5.2.1 General treewidth results

In Section 5.1.1 we present the results from running the heuristic algorithms on the programs of the PolyBench benchmark suite. This includes the upper bound treewidth and treewidth bound gap per benchmark, along the with average number of nodes for the graphs of these measurements.

From these results we find that the average upper bound treewidth is 9 for the graphs corresponding to regions of the RVSDG IR for the programs tested. To claim that this is a low treewidth, and to relate our findings to another work, we refer to Krausse *et al.* [22]. They find, in a correction of Thorup [38], that the treewidth of the CFGs of `C` programs is $7+g$ for g `goto` statements. Tree decompositions of this kind are successfully used in the SDCC compiler, which shows that treewidth of this magnitude can be used to implement practical and useful optimizations in a compiler. Examples of optimization algorithms that have been solved in linear time with the help of tree decompositions are also from Krausse [21]. Here it is proven that for programs with a CFG representation of bounded

treewidth, the register allocation problem can be solved in polynomial time, and that CSE can be solved in linear time.

For the benchmarked programs, we show that the average number of nodes in the RVSDG regions increase polynomially as a function of the treewidth of the graph. We also see a similar relationship between the gap of the upper and lower treewidth bounds, and the number of nodes in the region. As described in Section 4.4, a small gap between these bounds means the heuristics perform well, closely approximating the actual treewidth. Since both the upper bound treewidth, and the treewidth bound gap grows slower as the number of nodes in the graph increases, this gives us reason to belive that both the treewidth will be bounded, and that the performance of the heuristics will hold for even larger graphs.

We conclude that the results from our benchmarking programs are promising, showing low and bounded treewidths for a large set of RVSDG representations. We are able to find these treewidths in polynomial time using the implemented heuristics, which we are also able to show that perform well, closely modeling the accrual treewidth of the graphs. These results indicate the tree decomposition as a viable path for finding better optimizations for the RVSDG IR.

5.2.2 Analysis of dependencies in functions

In this section we analyse the results presented in Section 5.1.4. Figures 5.16, 5.17, and 5.18 show the generated graphs for the three summary-functions presented in Section 3.4.1. For all functions we see that while having more arguments in the function results in more nodes being created, this will not necessarily increase the treewidth, as in the struct and array cases.

To further understand how these graphs are generated, we have to consider the LLVM bytecode that is generated from the input programs. For all three graphs, the relationship between the allocation of the variable, and the necessary dependencies between this and the load and store operations will be used to reason about the resulting graph structure and its impact on the treewidth.

To create a program in SSA form directly from an AST, LLVM allocates function arguments on the stack for further use. This is achieved in the LLVM bytecode purposing the `ALLOCA` instruction. Since the RVSDG construction algorithm uses a single memory state edge to sequentialise memory operations, each such allocation node in the RVSDG is dependent on the previous allocation. This can be seen in Figure 5.16, generated from arguments passed as variables function, in the section marked "Allocation of stack space for variables". The variables are then stored at this location before they can finally be loaded and added together. This is expressed in the LLVM bytecode with the `LOAD`, `STORE`, and `BITADD32` instructions respectively. Since the RVSDGs current implementation serializes I/O operations, loads and stores in the graph are also interdependent of each other. In addition, each storage is dependent on its corresponding allocation, and each load is dependent on both its corresponding allocation and the preceding stores.

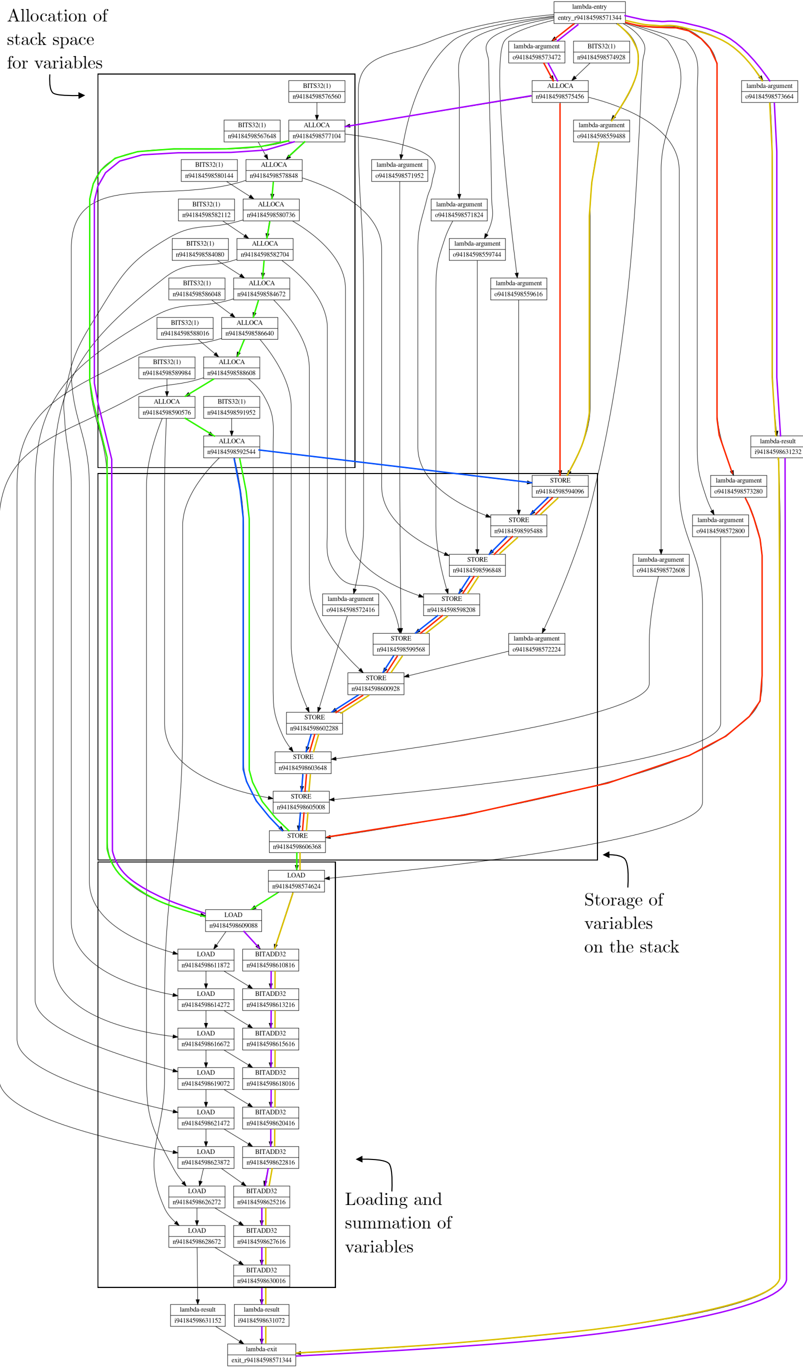


Figure 5.16: Lambda region for a function summarizing 10 arguments passed individually.

These three phases of the summary-function can be seen in the highlighted areas in Figure 5.16. We also highlight three different types of cycles generated for each argument summarized in the function. The green cycle is generated by the dependencies between allocation calls, the dependency between the allocation and the corresponding load, and the dependency between the set of stores and loads. Red cycles are generated by the dependency between the first store and allocation nodes, and each lambda-argument *i.e.*, the function argument to its corresponding store node. We also find two kinds of cycles generated due to the dependencies the entry node has on the result node of the function. This state edge at the right side of the figure forms cycles either with the chain of dependencies through the allocation and load nodes, or though the load and store nodes. These are represented as the magenta and yellow colored cycles respectively. The blue cycle is generated by the interdependence between stores and the dependency between the allocations and stores. As opposed to the other cycles, this cycle is only generated between the last allocation node and the first and last interdependent store nodes. This causes the blue kind of cycle *not* to scale with the number of arguments received and used by the function.

Looking at Figure 5.17 generated from the struct summary function, we see that the graph differs due to the alternative method of loading the values from memory. Instead of allocating the variables locally on the stack, a pointer to the struct is passed to the function, and the LLVM bytecode instruction `GETELEMENTPTR` is issued to get a reference to the individual elements in the struct. These elements are then loaded and added in the same way as in Figure 5.16. Unlike `ALLOCA` calls, `GETELEMENTPTR` calls are not interdependent and can be issued in parallel. The highlighted paths in the figure denotes the same kind of cycles as in figure above. The yellow and magenta paths denote the cycles generated by the state edge between the entry and exit nodes at the left side of the figure, through the chain of loads, or through the lambda-entry node respectively.

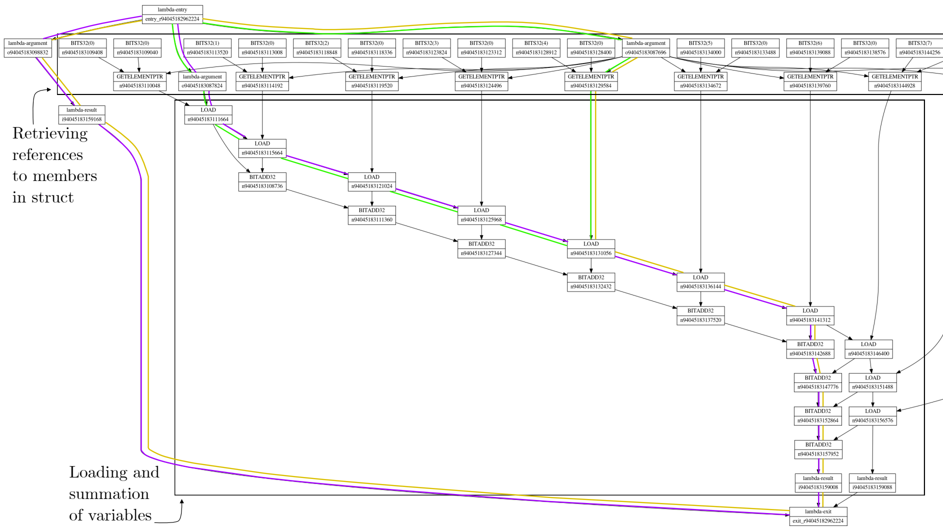


Figure 5.17: Lambda region for a function summarizing 10 arguments passed as a struct.

In the graph generated from the struct parameter, there is no chain of allocations and stores before the loads. In the parameters as values case, this magenta type cycle scales with the number of function parameters. This is not the case for the functions with the struct parameter. The green cycles are generated by the dependencies between loads and their respective `GETELEMENTPTR` call. We see that the blue and red type of cycles do not occur in Figure 5.17, since they are generated by dependencies including the store and allocation call chains, which does not occur in the struct case.

Figure 5.18 shows the program generated from the function with the array parameter. We notice some differences between loading values from a `C` style array and how structs are handled in the LLVM bytecode. We see that the program is able to predict that it needs to store n elements, and issues *one* allocation call for all members of the argument array passed to the function. The pointer to the array itself is also stored on the stack and has to be loaded each time it is referenced. The chain of dependencies appearing on the left side of the figure is therefore the sequence of loading the pointer, getting the element reference, loading the element and finally adding it together with the next element.

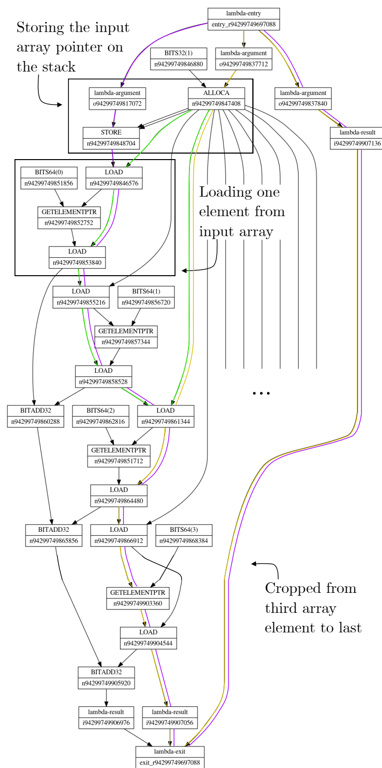


Figure 5.18: Lambda region for a function summarizing 10 arguments passed as an array.

Even though this graph structure has some differences from the struct case, we detect the same kinds of cycles in both graphs. Again, we see the yellow and magenta paths going through the entry and exit nodes on the right of the figure and either the chains of

loads and allocations or through the store node respectively. As in the struct case, the magenta cycles does not scale with the number of variables since this store only occurs once. We also observe similar green cycles from the dependencies between the loads and their allocations, as well as the lack of the red and blue types of cycles.

5.2.3 Analysis of dependencies with respect to variable liveness

This section presents an analysis of the results found in Section 5.1.5. As in the previous section, we will use the LLVM bytecode compiled from the various programs presented in the section, along with the graph structure of its corresponding RVSDGs to reason about the resulting treewidths.

Multiple variable allocation

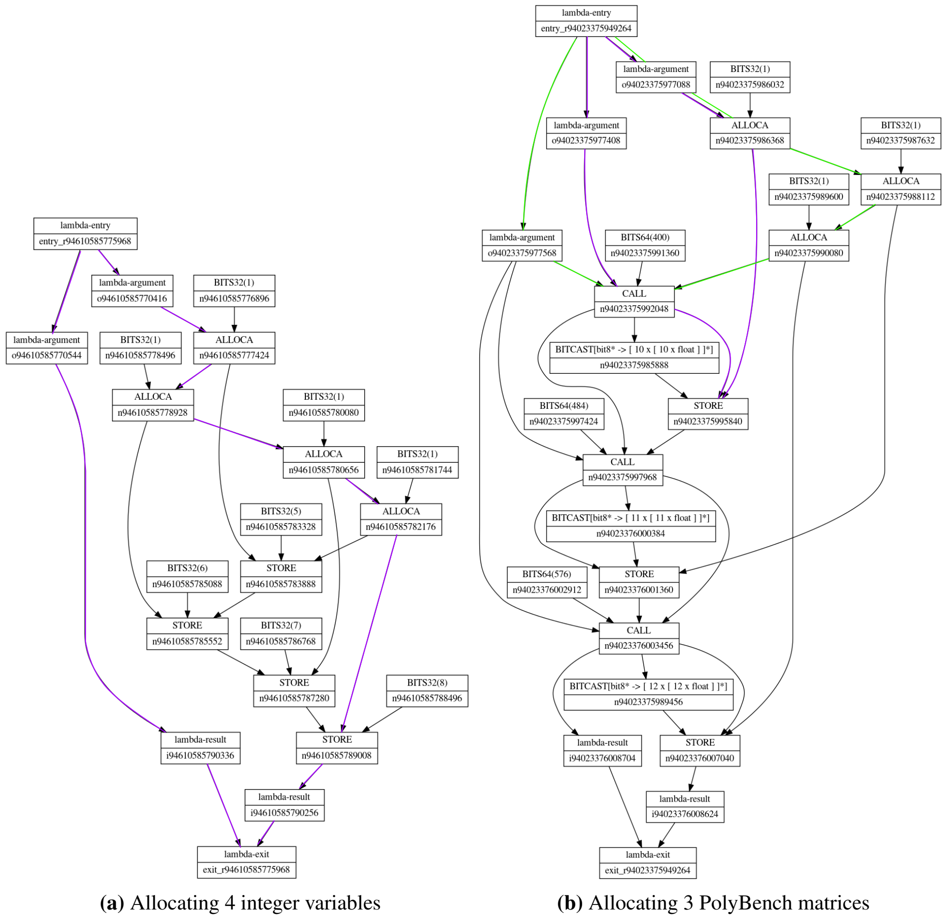
In Section 5.1.5, we identify the upper treewidth bounds for two programs allocating variables and matrices to be 3 and 4 respectively. In Figure 5.19 we show graphs representing these cases. Marked in both figures, are the type of cycles that appear in the graphs which increase as the number of variables allocated increases. For both types of allocations we find the paths colored magenta, representing the dependencies between the allocation and storage of the variables. In Figure 5.19b we also see cycles generated by the dependencies between the set of allocations and the respective function calls, denoted in green.

Variable allocations

In Figure 5.20a we show the generated graph for an allocation of 4 integer variables where each variable is passed once to the same function. Compared to the graph in Figure 5.19, the function called is itself passed as an argument to the node where the call of the functions occurs. This additional dependency to the lambda entry node, introduces a set of cycles in the graph marked as green in the figure. Also, each allocation node introduces an additional state edge to the load node that occurs before each call to the function, further adding another set of cycles in the graph marked in blue.

Verifying that this extra load dependency has a observable effect on the treewidth, we run a similar experiment with 4 integer variable allocations, where we instead of calling a function, add the 4 variables together. We find that the graph generated by this program is similar to the function call case discussed above, and that the treewidth is similar. When the variables are not called or added together, both have a higher upper treewidth bound of one. The graph using additions is shown in Figure 5.20b, where the colored paths mark the same kind of cycles as in Figure 5.20a. We find two differences between these functions, which therefore does not affect the treewidth of the graph in this case. Firstly, the dependency edge from the entry node to the exit nodes goes through the chain of function calls in Figure 5.20a, while it goes directly to the exit node in Figure 5.20b. This causes the blue path in 5.20b to contain fewer edges than in 5.20a. We also see that the green cycle through the function call dependencies does not appear in 5.20b.

In Figure 5.21 we show how the graphs differ when the functions calls are made to the same function, compared against the case where each variable is called by several different functions. This graph is generated from a program allocating 3 variables. In the Fig-



(a) Allocating 4 integer variables

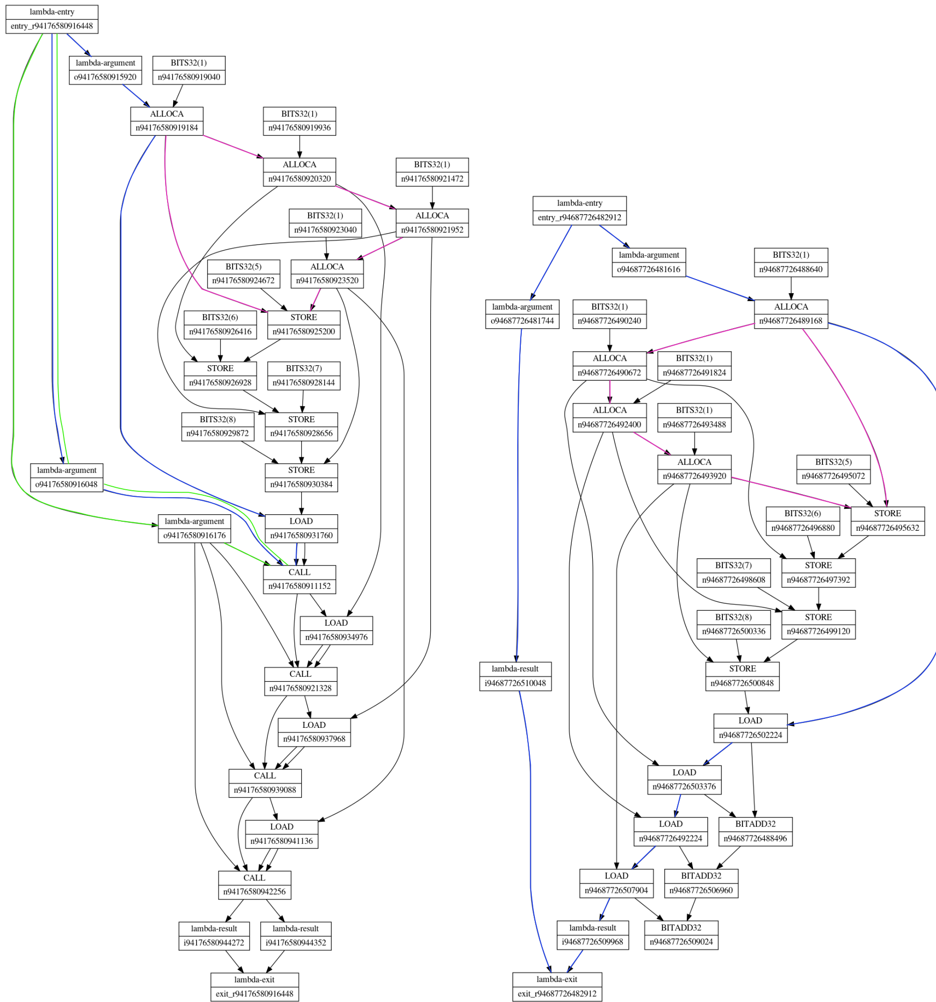
(b) Allocating 3 PolyBench matrices

Figure 5.19: Minimum number of variables needed to be allocated to generate a graph of maximum treewidth for the corresponding allocation types.

ure 5.21a each variable is called twice by the same function, and in Figure 5.21b each variable is called twice by two different functions, resulting in a higher upper bound treewidth. The structural difference between these graphs is the dependency of the function call to the entry node of the function. For 5.21a we see that each call-node is dependent on the same lambda-argument, shown in magenta, while for 5.21b two such lambda-arguments are passed to the two different functions, shown in magenta and green respectively.

Call orderings

In Figure 5.22 we show the allocation of 3 variables referenced two times by two different functions in a sequential call order. Compared to the graph showing the corresponding blockwise referencing to the variables in Figure 5.21a, we see that the edges from the



(a) Allocating 4 variables and passing each to a function call.

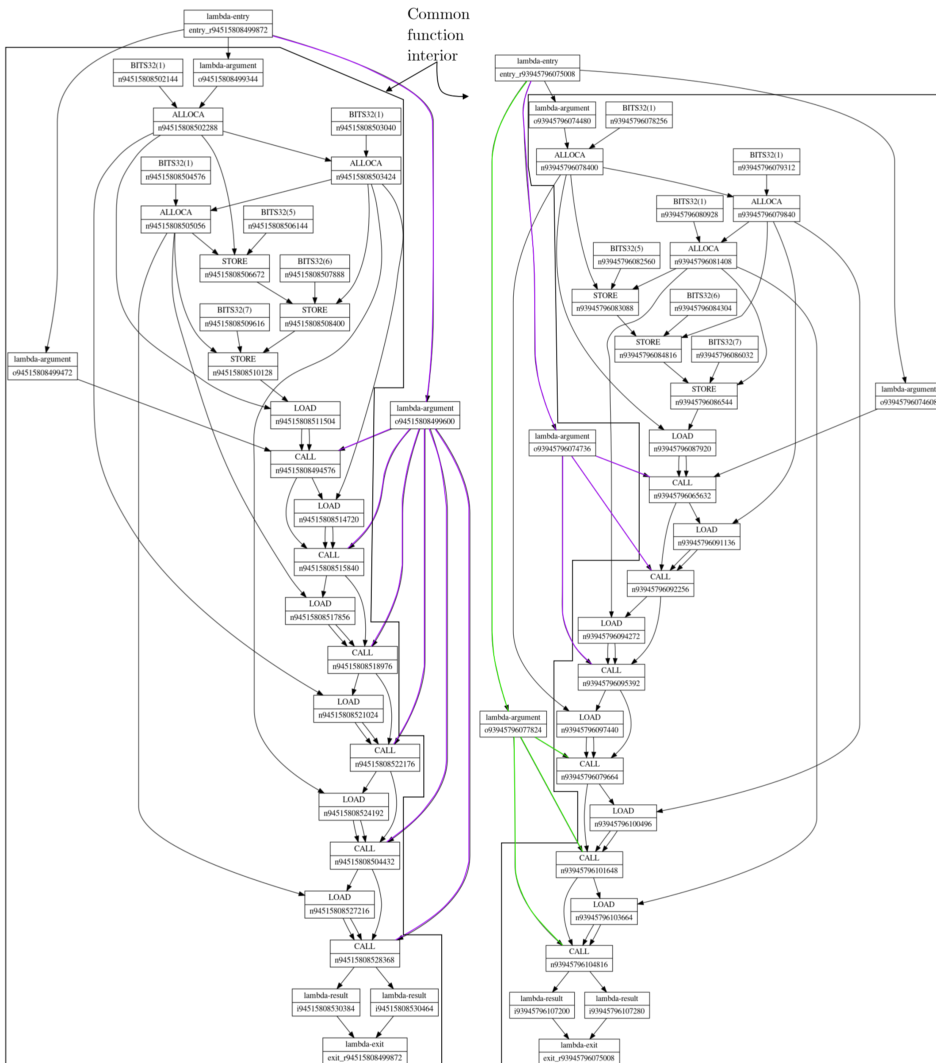
(b) Allocating 4 variables and adding them together.

Figure 5.20

lambda argument nodes to the function call nodes are separated by other calls throughout this chain of function calls, rather than blocked together as they are when referenced sequentially.

We find that further increasing the amount of references to a variable beyond a certain number of times in a blockwise call sequence, will no longer increase the treewidth. Increasing the number of times the variable is called in a sequential call sequence will not increase the treewidth at all.

Again, this indicates an upper bound that can be reached by a certain *number of calls* to the variables allocated, which is the same result we found for the *number of variables*



(a) Allocating 3 variables and passing them twice to the same funciton.

(b) Allocating 3 variables and passing them twice to three different functions.

Figure 5.21

allocated. Both these approaches assume either a constant number of variables allocated, or a constant number of times each variable is called. As seen in Section 5.1.5, when both these parameters are increased, the treewidth increases indefinitely.

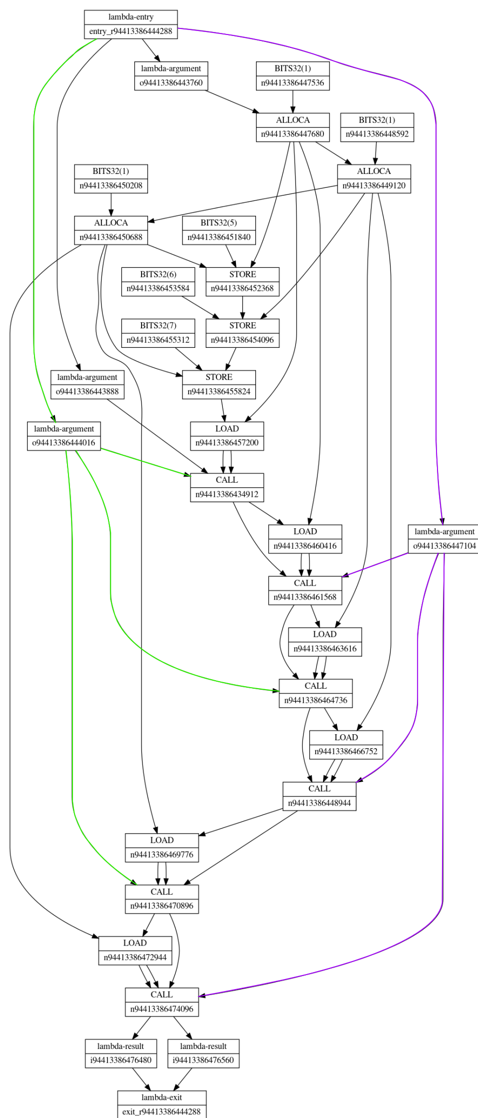


Figure 5.22: Allocating 3 variables and calling them twice in a sequential call order.

Variable allocations and use in expressions

We see that the difference in the resulting programs is how the loading of the variables are dependent on each other. Each value loaded is also dependent on the allocation of its storage space on the stack, and when only loading the variables, this dependency and the state dependency to the previous load are the only paths generating cycles in the graph. When adding all variables together we get a chain of additions that are also dependent on

both the previous load and the previous addition. Splitting the expressions into smaller parts, the number of these dependencies on previous additions gets fewer, and the length of these dependency chains gets smaller.

The effects on the treewidth appear to be minimal however, decreasing the upper treewidth bound by one when splitting the expression and by two when only loading the variables. From Figure 5.15b, we find the upper treewidth bound of 30 variable declarations referenced in 30 expressions. When either splitting the expression into two parts, or when loading all values separately this upper treewidth bound decreases from 39 to 31.

We also find that the ordering of the variables in the expressions affect the treewidth in the same manner as when passing the variables to function calls. While Figure 5.15b show the variables referenced in the expressions in a blockwise call order, when referencing the variables in a sequential call order the treewidth increases more gradually to $4 - 8$ for 14 allocations and 15 references, or 30 allocations and 4 references. The treewidth does then *not increase* above this threshold for the sequential call ordering.

If the variables are referenced once in a single expression, such as adding all the variables together, the upper treewidth is higher than if each such expression is split in two, adding half of the variables together in two separate expressions. We continue reducing the expression until the variables are not used in an expression, but simply stated by themselves, resulting in a single load of the variable in the resulting LLVM bytecode. We find that the upper treewidth bound is continuously lowered as the expression size gets smaller.

In Figure 5.19 we see that the graphs resulting from a program either allocating variables and passing each to a single function call, allocating 4 variables and adding them together are similar and have the same treewidths. We find that the major difference in the resulting graphs is the fact that the function calls have an extra dependency to the lambda-argument, which expressions such as addition does not.

In Figure 5.23 we show the differing sections of three programs issuing 10 variable allocations, referencing them in the three different manners presented above. We see that when only declaring the variables in the top row of the figure, there is a dependency chain between the loading of the values. When also adding the variables together, we introduce addition nodes which are also dependent on their corresponding load and previous addition. This is seen in the bottom two subfigures.

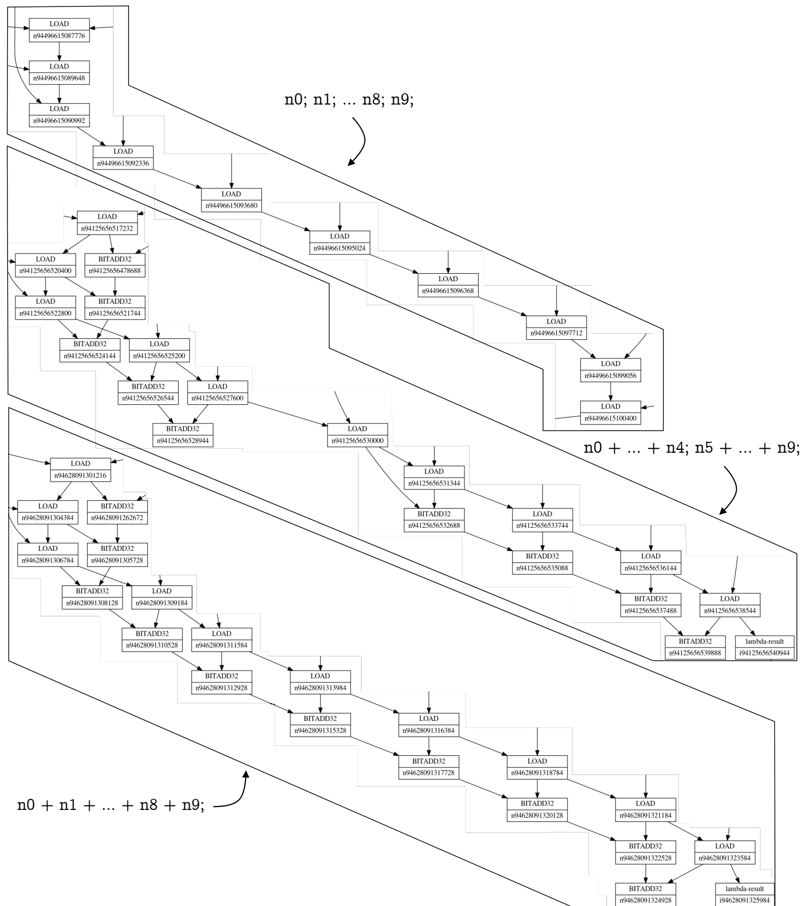


Figure 5.23: Section of the graph concerning the reference to the variables after their allocation for three different expression types.

Chapter 6

Conclusion

6.1 Conclusion

In this thesis we have developed a framework for both parsing the RVSDG IR to a corresponding graph representation, and finding the upper and lower treewidth bounds of these graphs. For a set of benchmark programs we find that the treewidth is low and bounded.

This framework is further used to investigate how different program features impact the treewidth and graph structure of the RVSDG. We have induced structural changes to a set of benchmark programs by applying different optimizations, and analyzed how this affects the treewidth of the resulting programs. We found that while there is no correlation between the runtime of the resulting programs and the optimizations individually, collections of optimizations that increase the efficiency of the program also results in a reduced treewidth of the corresponding graphs.

We have also identified and induced program features that affect the RVSDG treewidth, and presented tailored programs to investigate these effects. We have analyzed two such categories of features. Firstly, identifying different methods of passing arguments to functions and different orderings of function calls, we have seen how these affect the resulting treewidth. Secondly, mapping the effects of variable liveness and allocation, we found that for an increasing amount of variables allocated, or an increasing amount of references to these variables, the treewidth of the resulting program increases. We have shown that this growth is bounded when only one of the parameters increase, and that it grows indefinitely when both increase.

We have also conducted a detailed analysis of the RVSDGs generated by these programs. We have shown how the program feature changes affect the resulting graphs, using the data flow and state dependencies between nodes to reason about the resulting treewidths. We identified different types of generated cycles in the graph, connecting them to the program feature changes induced. These cycles scales proportionally with the number of nodes or operations in the graph.

6.2 Future work

Determining that we have a low and bounded treewidth for the RVSDG opens several avenues of further research.

By implementing a method for finding the upper bound for the treewidth, we have taken the first step towards finding the tree decomposition itself. We also have created a framework that allows parsing of RVSDG programs into corresponding graphs. This framework already includes important features required to find the tree decomposition, and is possible to extend to implement further operations needed. An overview of approaches to find the tree decomposition is given in Section 2.5.2.

To benefit from these results, we also need to decide and design specific optimization algorithms for the RVSDG. This requires formulating the optimization problems on the graph, such that they can be solved using dynamic programming techniques with the help of the tree decomposition. These techniques are discussed in Section 2.5.3. We could also investigate practical properties of running dynamic programming algorithms using the tree decomposition. This can be used to further investigate the feasibility of running such algorithms in a real-world compiler, and determine what treewidth the tree decomposition needs to provide results within a realistic time-frame.

In the experiments we investigate the program feature impact on the resulting treewidth and note that there is only a change in the upper bound treewidth. There is thus a possibility that the actual treewidth does not change. Instead, structures in the graph might change in such a way that the minor-min-width upper bound heuristic does not perform as well, overestimating the upper treewidth bound of the graph. To further investigate this, finding the tree decomposition as delineated above will be beneficial.

Bibliography

[1] Aho, Alfred V. and Lam, Monica S. and Sethi, Ravi and Ullman, Jeffrey D. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[2] Arseny Kapoulkine *et. al.* pugixml, version 1.10. `https://pugixml.org/`, 2019. GitHub; Accessed 12-06-2019.

[3] Asbjørn Djupdal. rvsdg-viewer. `https://github.com/phate/rvsdg-viewer`, 2017. GitHub; Accessed 12-06-2019.

[4] Helge Bahmann, Nico Reissmann, Magnus Jahre, and Jan Meyer. Perfect reconstructability of control flow from demand dependence graphs. *ACM Transactions on Architecture and Code Optimization*, 11:1–25, 01 2015.

[5] Jeremias Berg and Matti Jarvisalo. SAT-based approaches to treewidth computation: An evaluation. In *2014 IEEE 26th International Conference on Tools with Artificial Intelligence*, pages 328–335. IEEE, 2014.

[6] Hans Bodlaender. Discovering treewidth. *Lecture Notes in Computer Science*, 3381:1–16, 01 2005.

[7] Hans L. Bodlaender. Dynamic programming on graphs with bounded treewidth. *Automata, Languages and Programming*, pages 105–118, 1988.

[8] Hans L. Bodlaender, Pål Grønås Drange, Markus S. Dregi, Fedor V. Fomin, Daniel Lokshtanov, and Michal Pilipczuk. A $\mathcal{O}(c^k n)$ 5-approximation algorithm for treewidth. *CoRR*, abs/1304.6321, 2013.

[9] Burgess, Mark and Hale-Evans, Ron. The GNU C Programming Tutorial. *A GNU Manual*, 2020. Edition 4.1; `http://www.crasseux.com/books/ctut.pdf`.

[10] G. Charwat. Dynamic programming on tree decompositions using binary decision diagrams: Research summary. *Technical Communications of ICLP*, 1433, 01 2015.

[11] François Clautiaux, Aziz Moukrim, Stéphane Nègre, and Jacques Carlier. Heuristic and metaheuristic methods for computing graph treewidth. *RAIRO - Operations Research*, 38(1):13–26, 2019.

[12] D-Wave Systems Inc. D-wave networkx, version 0.82. `https://docs.ocean.dwavesys.com/projects/dwave-networkx/en/latest/_modules/dwave_networkx/algorithms/elimination_ordering.html`, 2019. Read the Docs; Accessed 12-06-2019.

[13] R. Diestel. *Graph Theory*. Electronic library of mathematics. Springer, 2006.

[14] R. G Downey and M. R Fellows. *Fundamentals of parameterized complexity*. Springer, 2013. OCLC: 865474474.

[15] E. R. Gansner, E. Koutsofios, S. C. North, and K. . Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, March 1993.

[16] Fănică Gavril. Algorithms for minimum coloring, maximum clique, minimum covering by cliques, and maximum independent set of a chordal graph. *SIAM Journal on Computing*, 1(2):180–187, 1972.

[17] Vibhav Gogate and Rina Dechter. A complete anytime algorithm for treewidth. *CoRR*, abs/1207.4109, 2012.

[18] Martin Jambor. The new intraprocedural Scalar Replacement of Aggregates. *GCC Summit*, 2010. `https://gcc.gnu.org/wiki/summit2010?action=AttachFile&do=view&target=jambor.pdf`.

[19] Neil Johnson and Alan Mycroft. Combined code motion and register allocation using the value state dependence graph. In *Lecture Notes in Computer Science*, pages 1–16. Springer Berlin Heidelberg, 2003.

[20] J. Kleinberg and É. Tardos. *Algorithm Design*. Pearson/Addison-Wesley, 2006.

[21] Philipp Klaus Krause. *Graph decomposition in routing and compilers*. PhD thesis, Frankfurt am Main, 2016.

[22] Philipp Klaus Krause, Lukas Larisch, and Felix Salfelder. The tree-width of c. *Discrete Applied Mathematics*, 2019.

[23] Brendan McKay. graph formats. `http://users.cecs.anu.edu.au/~bdm/data/formats.txt`, `http://users.cecs.anu.edu.au/~bdm/data/formats.html`, 2015. Accessed 12-06-2019.

[24] N. Llopis, C. Nicholson et. al. Unittest++, version 2.0.0. `https://github.com/unittest-cpp/unittest-cpp`. Github.

[25] Adam Nemet. Compiler-assisted Performance Analysis. *2016 US LLVM Developers' Meeting*, 2016. https://llvm.org/devmtg/2016-11/Slides/Nemet-Compiler-assistedPerformanceAnalysis.pdf.

[26] Helge Bahmann Nico Reismann. Jive RVSDG API. https://github.com/phate/jive.git, 2019. GitHub; Accessed 08-10-2019.

[27] Nico Reismann, Magnus Sjalander. polybench-jlm. https://github.com/phate/polybench-jlm, 2019. GitHub; Accessed 12-06-2019.

[28] Geoff Nixon. Clang optimization levels. https://stackoverflow.com/a/27576831, 2014. Stack Overflow; Accessed 02-06-2020.

[29] Ohio State University. PolyBench/C. http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/, 2015. Accessed 12-06-2019.

[30] LLVM Project. LLVM's Analysis and Transform Passes. https://llvm.org/docs/Passes.html, 2020. The LLVM Compiler Infrastructure Documentation; Accessed 15-05-2020.

[31] Nico Reismann. Jlm: An experimental compiler/optimizer for llvm ir. https://github.com/phate/jlm.git, 2019 - checked out at commit 3ae45dfe406f2d4ec6005ff093eb5b929d3de8ff.

[32] Nico Reissmann. *Principles, Techniques, and Tools for Explicit and Automatic Parallelization*. PhD thesis, NTNU, 2019.

[33] Nico Reissmann, Jan Meyer, Helge Bahmann, and Magnus Själander. RVSDG: An Intermediate Representation for Optimizing Compilers. *arXiv:1912.05036*, 2019.

[34] Donald J. Rose. A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations. In RONALD C. READ, editor, *Graph Theory and Computing*, pages 183 – 217. Academic Press, 1972.

[35] Clang Development Team. clang - the Clang C, C++, and Objective-C compiler. https://clang.llvm.org/docs/CommandGuide/clang.html, 2020. Clang 11 Documentation; Accessed 01-06-2020.

[36] Clang Development Team. Clang Compiler User's Manual. https://clang.llvm.org/docs/UsersManual.html, 2020. Clang 11 Documentation; Accessed 20-05-2020.

[37] Clang Development Team. Frequently Asked Questions (FAQ). https://clang.llvm.org/docs/FAQ.html, 2020. Clang 11 Documentation; Accessed 06-05-2020.

[38] Mikkel Thorup. All structured programs have small tree width and good register allocation. *Information and Computation*, 142(2):159–181, 1998.

[39] Rim van Wersch and Steven Kelk. Toto: An open database for computation, storage and retrieval of tree decompositions. *Discrete Applied Mathematics*, 217:389 – 393, 2017.

[40] Vince Bridgers, Felipe de Azevedo Piovezan. LLVM IR Tutorial. *LLVM Developers Conference Brussels*, 2019. `https://www.llvm.org/devmtg/2019-04/slides/Tutorial-Bridgers-LLVM_IR_tutorial.pdf`.

[41] Daniel Weise, Roger F. Crew, Michael D. Ernst, and Bjarne Steensgaard. Value dependence graphs: Representation without taxation. In *POPL '94: Proceedings of the 21st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 297–310, Portland, OR, January 1994.

Appendix A

Result Tables

Benchmarks	Upper bound		Limits		Relative	
	O1	O2	max	min	O1	O2
2mm	5	5	16	5	0.00	0.00
3mm	5	5	14	5	0.00	0.00
adi	7	7	10	7	0.00	0.00
bicg	5	5	10	5	0.00	0.00
correlation	5	5	10	5	0.00	0.00
covariance	5	5	9	5	0.00	0.00
deriche	7	7	12	7	0.00	0.00
doitgen	5	5	10	5	0.00	0.00
fdtd-2d	7	7	13	7	0.00	0.00
gesummv	5	5	12	5	0.00	0.00
mvt	5	5	10	5	0.00	0.00
trmm	5	5	10	5	0.00	0.00
gemm	6	5	14	5	0.11	0.00
symm	6	5	13	5	0.12	0.00
gramschmidt	6	5	10	5	0.20	0.00
trisolv	6	4	8	4	0.50	0.00
nussinov	8	6	8	6	1.00	0.00
gemver	5	6	19	5	0.00	0.07
syrk	5	6	11	5	0.00	0.17
atax	5	6	10	5	0.00	0.20
seidel-2d	10	7	10	6	1.00	0.25
syr2k	6	8	13	6	0.00	0.29
ludcmp	5	7	10	5	0.00	0.40
jacobi-1d	5	6	7	5	0.00	0.50
floyd-warshall	5	6	7	5	0.00	0.50
cholesky	4	7	7	4	0.00	1.00
durbin	5	7	7	5	0.00	1.00
lu	4	7	7	4	0.00	1.00

jacobi-2d	7	10	10	6	0.25	1.00
heat-3d	11	11	11	8	1.00	1.00

Table A.1: Comparison between the upper bound treewidth for a program optimized at level `O1` or `O2` compared to programs optimized with individual optimizations. Included is the generated upper bound treewidth for the optimization level, largest and smallest treewidths generated by the program for any optimization, and relative treewidth sizes compared to these.

Benchmarks	**Lower bound**		**Limits**		**Relative**	
	`O1`	`O2`	max	min	`O1`	`O2`
adi	6	6	7	6	0.00	0.00
atax	5	5	6	5	0.00	0.00
bicg	5	5	6	5	0.00	0.00
correlation	5	5	7	5	0.00	0.00
covariance	4	4	6	4	0.00	0.00
deriche	5	5	7	5	0.00	0.00
doitgen	4	4	6	4	0.00	0.00
durbin	5	5	5	5	0.00	0.00
fdtd-2d	6	6	6	6	0.00	0.00
gemm	5	5	7	5	0.00	0.00
gesummv	5	5	7	5	0.00	0.00
gramschmidt	5	5	6	5	0.00	0.00
jacobi-2d	5	5	6	5	0.00	0.00
ludcmp	5	5	6	5	0.00	0.00
mvt	5	5	6	5	0.00	0.00
symm	5	5	6	5	0.00	0.00
syr2k	5	5	6	5	0.00	0.00
syrk	5	5	6	5	0.00	0.00
trmm	5	5	6	5	0.00	0.00
2mm	5	4	7	4	0.33	0.00
3mm	5	4	7	4	0.33	0.00
trisolv	5	4	6	4	0.50	0.00
nussinov	6	5	6	5	1.00	0.00
floyd-warshall	4	5	6	4	0.00	0.50
gemver	5	6	7	5	0.00	0.50
seidel-2d	7	6	7	5	1.00	0.50
cholesky	4	5	5	4	0.00	1.00
lu	4	5	5	4	0.00	1.00
heat-3d	7	7	7	6	1.00	1.00
jacobi-1d	5	5	5	4	1.00	1.00

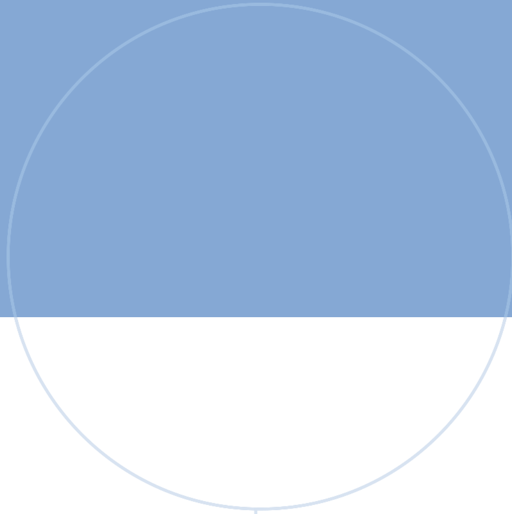
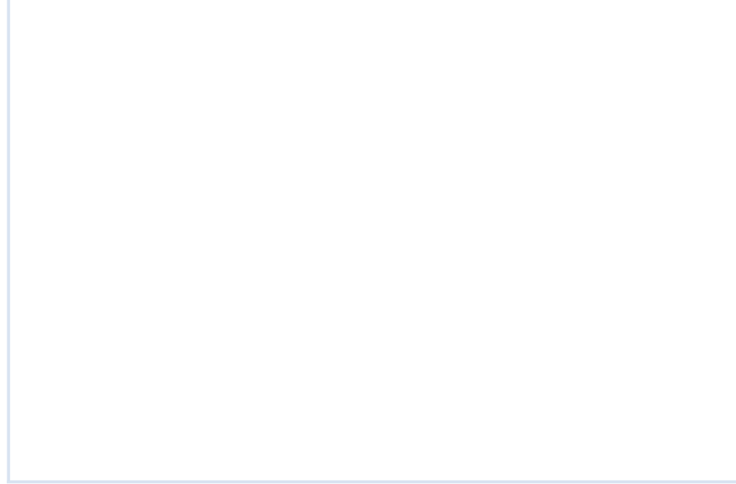
Table A.2: Comparison between the lower bound treewidth for a program optimized at level `O1` or `O2` compared to programs optimized with individual optimizations.

Variables	References in function calls									
	1	2	3	4	5	6	7	9	20	30
1	2–2	3–3	3–3	3–3	3–3	3–3	3–3	3–3	3–3	3–3
2	3–3	4–4	4–4	4–4	4–4	4–4	4–4	4–4	5–5	5–5
3	4–4	5–6	5–6	6–6	6–6	6–6	6–6	6–6	7–7	7–7
4	4–5	5–8	5–9	6–10	7–8	8–8	8–8	8–8	9–9	9–9
5	4–6	5–9	6–11	7–12	8–10	8–10	9–10	9–10	11–11	11–11
6	4–6	5–10	6–12	7–14	8–15	9–12	9–12	11–12	13–13	13–13
7	4–7	5–9	6–13	8–16	8–17	8–14	9–14	11–14	15–15	15–15
8	4–7	5–9	6–13	7–17	8–19	8–16	10–16	12–16	17–17	17–17
9	4–7	5–9	6–12	7–17	8–21	9–22	11–23	11–18	18–19	19–19
10	4–7	5–10	6–15	7–18	8–22	10–23	10–25	13–20	19–21	21–21
20	4–7	5–11	7–15	7–18	8–22	9–24	10–27	13–32	23–41	34–41
30	5–7	5–12	6–15	7–18	8–22	10–25	11–28	14–36	30–70	37–61

Table A.3: Lower and upper treewidth bounds for an increasing amount of variables allocated downwards, and number of calls to each variable rightwards.

Variables	References in expressions					
	1	2	5	10	20	30
1	2–2	2–2	2–2	2–2	2–2	2–2
2	3–3	3–3	3–3	3–3	3–3	3–3
3	4–4	4–4	4–4	4–4	4–4	4–4
4	4–5	4–5	5–5	5–5	5–5	5–5
5	4–6	5–6	6–6	6–6	6–6	6–6
6	4–6	5–7	6–7	7–7	7–7	7–7
7	4–7	5–9	7–8	8–9	8–9	8–9
8	4–7	5–10	7–11	9–11	9–11	9–11
9	4–8	5–11	7–12	10–12	10–12	10–12
10	4–7	6–11	7–12	10–12	11–12	11–12
20	4–10	5–15	8–31	13–26	21–28	21–26
30	4–10	5–15	8–35	13–47	22–39	31–39

Table A.4: The generated lower and upper bound treewidth bounds for programs of an increasing number of variables allocated downwards, and increasing number of times these are used in an expression rightwards.

NTNU
Norwegian University of
Science and Technology