Feasibility of Optimizations Requiring Bounded Treewidth in a Data Flow Centric Intermediate Representation

Sigve Sjømæling Nordgaard and Jan Christian Meyer
Department of Computer Science, NTNU
Trondheim, Norway

Abstract

Data flow analyses are instrumental to effective compiler optimizations, and are typically implemented by extracting implicit data flow information from traversals of a control flow graph intermediate representation. The Regionalized Value State Dependence Graph is an alternative intermediate representation, which represents a program in terms of its data flow dependencies, leaving control flow implicit. Several analyses that enable compiler optimizations reduce to NP-Complete graph problems in general, but admit linear time solutions if the graph's *treewidth* is limited. In this paper, we investigate the treewidth of application benchmarks and synthetic programs, in order to identify program features which cause the treewidth of its data flow graph to increase, and assess how they may appear in practical software. We find that increasing numbers of live variables cause unbounded growth in data flow graph treewidth, but this can ordinarily be remedied by modular program design, and monolithic programs that exceed a given bound can be efficiently detected using an approximate treewidth heuristic.

1 Introduction

Effective program optimizations depend on an intermediate representation that permits a compiler to analyze the data and control flow semantics of the source program, so as to enable transformations without altering the result. The information from analyses such as live variables, available expressions, and reaching definitions pertains to data flow. The classical method to obtain it is to iteratively traverse a control flow graph, where program execution paths are explicitly encoded and data movement is implicit. Alternatively, the program can be represented as a data flow graph, where data movement and dependencies are explicitly encoded, and control flow information is implicit for the compiler to infer.

Explicit data flow encoding facilitates a number of desirable properties in an intermediate representation [16]. It also reflects modern processor architectures more accurately, given that performance optimizations are increasingly driven by mitigating the cost of data movement, and relaxing constraints on the order of execution to exploit potential concurrency [2]. Data flow centric intermediate representations [18] have seen limited practical application, primarily because most variants require control flow information to be encoded separately for code generation purposes. The Regionalized

This paper was presented at the NIK-2020 conference; see http://www.nik.no/.

Value State Dependence Graph *(RVSDG)* mitigates this issue, and implicitly retains sufficient information to recreate the control flow of the source program [3]. This permits entire programs to be treated entirely as demand dependence graphs. Several analyses that enable compiler optimizations reduce to NP-complete graph problems in general, but admit linear time solutions if the *treewidth* of the graph is bounded.

In this study, we empirically examine treewidths of RVSDG representations derived from a set of benchmark programs. These encompass both realistic computational kernels from applications, and synthetic benchmarks of our own design. Our objective is to identify whether or not we can isolate programming constructs that can produce RVSDGs of very large treewidth, and to evaluate whether or not these features are likely to occur in computationally demanding application software. Due to the complexity of calculating treewidth precisely, we implement two known heuristics that provide upper and lower bounds, and use them to reveal tendencies in its growth. We find that our application benchmarks exhibit low treewidth values, that the constructs which increase RVSDG treewidth are unlikely to be applied in a practical programming scenario, and that the heuristics can be used to efficiently evaluate whether or not to disable an optimization that becomes intractable in corner cases.

The rest of the paper is structured as follows. Section 2 describes the treewidth metric of a graph, and justifies its significance to compiler optimizations. Section 3 summarizes the RVSDG intermediate representation, and illustrates its use. Section 4 describes our experimental setup, heuristics to evaluate approximate treewidth, choice of application benchmarks, and the design of our synthetic benchmarks. Section 5 presents the results of our experiments, and discusses tendencies in treewidth among the application benchmarks, as well as the consequences of varying the parameters of the synthetic benchmarks. Section 6 concludes our study.

2 The Treewidth Graph Metric

A *tree decomposition* of a graph $G(V,E)$ is a tree $T(I,F)$ where the nodes $X_i \in I$ correspond to subsets of V, and satisfy three criteria:

1. The union of all X_i equals V.
2. If a vertex in V is included in two subsets X_i, X_j, it is also included in each subset on the path between them.
3. For each edge $\{u,v\} \in E$, there is an X_i which contains both u and v.

The *width* of T is the size of its largest subset minus one, *i.e.*

$$\max_i |X_i| - 1 \tag{1}$$

As tree decompositions are not unique, the *treewidth* of G is defined as the minimal width among all its tree decompositions.

The treewidth is a measure of the tree decomposition of the graph, and thus captures the graphs property of being "tree-like". Specifically, it represents the number of nodes that must be deleted in order to separate the graph. Many NP-complete problems are tractable on graphs of bounded treewidth [9]. One application is in the implementation of efficient optimization algorithms. Significant compiler optimization improvements have been found using tree decompositions for CFGs, including polynomial-time algorithms for band selection, redundancy elimination, and register allocation.

Use of tree decompositions admits efficient and provably optimal algorithms [10]. It is important that decompositions have small treewidths to achieve an efficient run-time and result quality of these algorithms. Several graph problems can be solved with time complexities that are single-exponential in terms of treewidth, and linear in number of nodes [5]. Thorup found that `goto`-free programs have bounded treewidth, and Throup's heuristic is used to obtain tree decompositions of CFGs in the SDCC `C` compiler [17].

3 Regionalized Value State Dependence Graphs

The RVSDG is an acyclic demand-dependence graph where nodes represent operations and control flow constructs, while edges represent dependencies between them. Each node encompasses an (optional) ordered set of operand inputs, and a set of resulting outputs. Each edge connects an output to exactly one input of another node, thus modeling data flow in the program, and implicitly, a constraint on admissible sequences that the operations can be executed in. *Simple* nodes represent primitive operations, while *structural* nodes represent *regions* that encapsulate subgraphs of arbitrary complexity, and assign particular semantics to them.

There are six types of structural nodes:

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 additional *apply* node invokes a function, passes arguments, and computes the function body.

δ-nodes model global variables. Inputs are external variables the node is dependent on, and output is 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.

Figure 1 demonstrates RVSDG representation using a recursive Fibonacci computation as an example. The nested structure of the graph is shown with the hierarchy of regions drawn as a graph of structural nodes, and also with region contents expanded, to clarify the correspondence with the source program. This example illustrates how RVSDG explicitly represents data flow. It is beyond the scope of this paper to prove that RVSDG suffices to model arbitrary program logic, and we refer to Bahmann *et al.*[3] for a rigorous treatment of the topic.

4 Method

In this section, we describe the configuration of our experiments, and the construction of the programs we experiment with. We begin by summarizing the software that is used throughout the experiments, including the design of the components that were developed

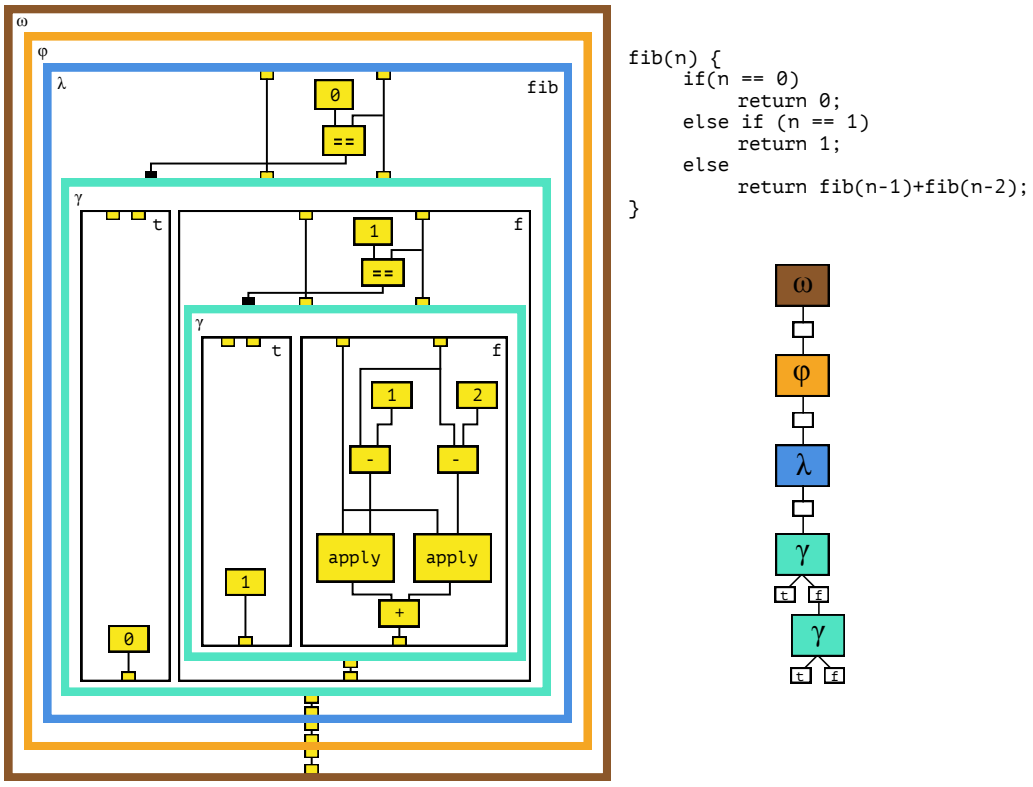

Figure 1: Recursive Fibonacci computation in RVSDG representation, shown with region contents expanded (left), and as a hierarchy of the corresponding structural nodes (right).

specifically for this work. Subsequently, we explain our approach to find approximate graph treewidths. Next, we describe a set of realistic, performance-sensitive application benchmark programs that are examined for comparison with the results from the synthetic benchmarks, and finally, we cover our synthetic benchmark programs designed to experiment with the impact that particular constructs have on RVSDG treewidth.

Experimental Setup

The principal components of our method are the `jive` compiler back end [11], the `LLVM` compiler infrastructure [14], and the `jlm` framework interface [15]. `LLVM` provides a complete compilation toolchain from C source code through to `x86_64` machine code. We utilize it to study RVSDG representations by injecting analyses at the intermediate representation stage: an RVSDG representation is constructed from LLVM IR, our bespoke programs manipulate and examine the RVSDG, before the RVSDG is destructed to LLVM IR and passed on to its low-IR optimizations and code generation stages.

We implement the `rvsdg-treedc` framework which includes a parser of the RVSDG XML output from *jlm*, transforming it into a corresponding graph representation in the *dotfile* format [6]. The pipeline to generate a graph and determine its treewidth from a source program is shown in Figure 2, where the `rvsdg-treedc` framework consists of the last two steps.

Treewidth Approximation

Due to the intractable computational expense of determining the treewidth of an arbitrary graph precisely [4], we estimate it by applying simpler, computationally feasible heuristics that establish upper and lower bounds on its magnitude. Gogate and Dechter [7]

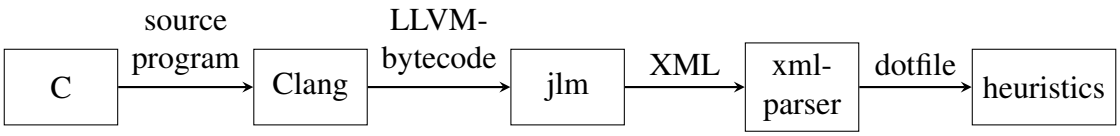

Figure 2: Compilation pipeline of the `rvsdg-treedc` framework.

present a branch-and-bound algorithm for evaluating treewidth, including three heuristics to obtain upper bounds, and a novel *minor-min-width* heuristic for lower bounds.

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. We know that for every graph G there exists a triangulation H such that the treewidth of G equals the treewidth of H.

Thus, the treewidth of G is at most as large as the treewidth found for H. The *min-fill* heuristic finds this triangulation using an ordering of vertexes that adds the least number of edges when eliminated from the graph. Our experimental results align with their observation that the *min-fill* approach consistently produces tighter bounds than the alternative upper bound heuristics, and this is used as the upper bound in our experiments.

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. 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. This is the basis of the minor-min-width heuristic.

We show graph treewidths as an interval between *minor-min-width* and *min-fill* bounds, as this suffices to show trends in its variation without determining an exact value.

Application Benchmarks

We generate a set of RVSDGs using a subset of the PolyBench benchmark suite [13] as input, and measure their treewidths. The suite consists of 30 numerical computations from various domains such as linear algebra, statistics, physics simulations *etc*. The small size and simple structure of the PolyBench benchmark programs simplify their analysis, and we use an existing fork of PolyBench [12], which contains support for compiling the benchmarks with the `jlm` compiler. This is extended to create the required XML files.

Synthetic Benchmark Construction

Functions and Arguments

By examining the treewidths of unoptimized programs in the PolyBench suite, we discover that the number of arguments passed to their main computational kernels is a significant cause of treewidth differences. We investigate the effect of function parameters by creating a set of synthetic benchmarks, each containing a single function that calculates the sum of its arguments, testing three different methods of passing arguments to a `C` function, as shown in Listing 1. They encompass call-by-value using individual variables, call-by-reference to a structured type, and call-by-reference to a contiguous array.

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

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

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

Listing 1: Three separate ways of passing arguments to a function in the C programming language.

This results in three semantically equivalent functions with different program structures, and different resulting RVSDGs. The programs we experiment with consist of a main function that initializes the variables, and calls one of these three functions.

Another influential factor is the order of function calls when calling several functions, or one function several times. We identify two orders shown in Listings 2 and 3, and refer to these as *blockwise* and *sequential* order, respectively. Blockwise order groups invocations of one function with differing arguments together, while sequential order groups invocations where the same arguments are passed to different functions.

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

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

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

Listing 2: Blockwise call order.

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

f1(n1);
f2(n1);

...

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

Listing 3: Sequential call order.

Variable liveness

We create a similar set of synthetic benchmark programs to investigate the effect of variable liveness with respect to data and state dependencies in the RVSDG. Live variable analysis is a data-flow analysis to determine which variables contain values that will still be used after a given program point. Live variables are easily found in a data-flow graph, as the edges already represent the flow of data. Thus, a variable is live at point p when there is an edge from p to the variable node [1].

Johnson [8] shows that that for two values connected by an edge in a data flow graph, the edge may introduce constraints on the liveness of the variable. RVSDG 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 short live-range, while variables that do not will have a longer live-range.

Inspired by the PolyBench benchmark programs, we write a set of custom programs to induce these features that also affect the RVSDG treewidth. These are analyzed with

respect to liveness, specifically with regard to how many variables are allocated, their types, and how they are referenced in the program.

5 Results and Discussion

In this section, we present the results of our experiments with application and synthetic benchmarks. We begin by examining the application benchmarks, and find that the upper bounds on treewidths of their RVSDG representations are low, typically single-digit numbers. Next, we investigate effects that are induced by various sequences of function calls and argument passing methods, and by increasing the number of live variables.

Application Benchmark 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.

Figure 3 shows 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.

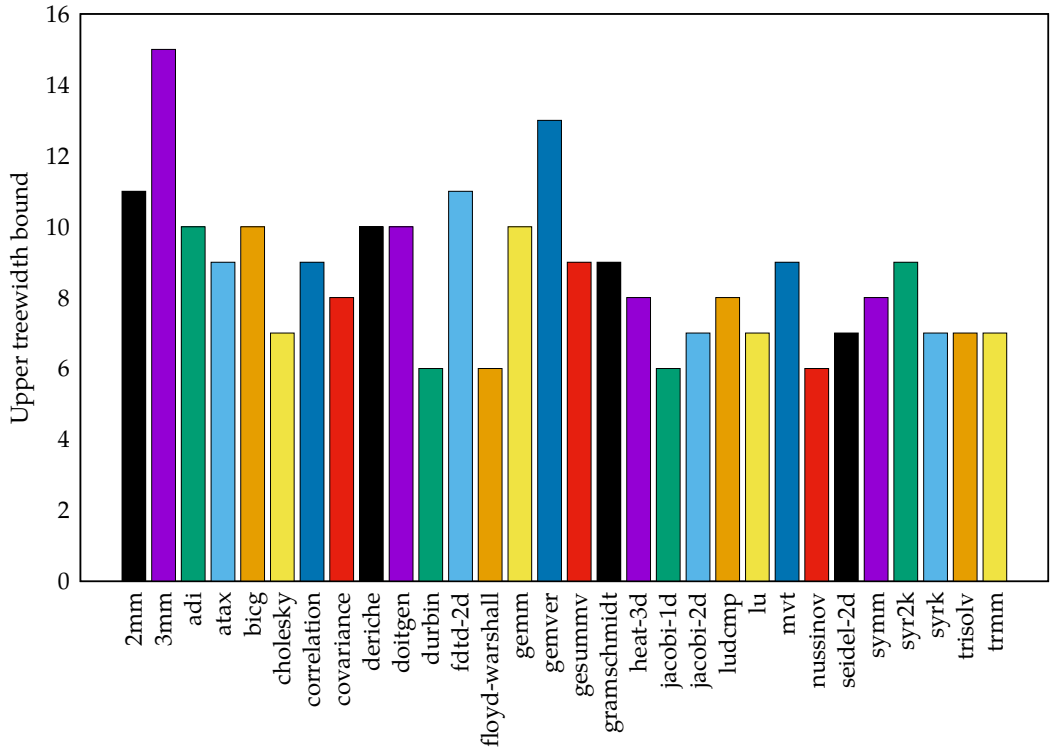

Figure 3: Upper bound treewidth per benchmark.

Figure 4 shows how the average number of nodes relates to the upper treewidth bound. We see that as the upper treewidth bound increases, the average number of nodes in the corresponding regions increases approximately polynomially.

This is demonstrated by fitting a polynomial curve to each figure. Figure 4 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 $n = 20 + 1.3tw^2$

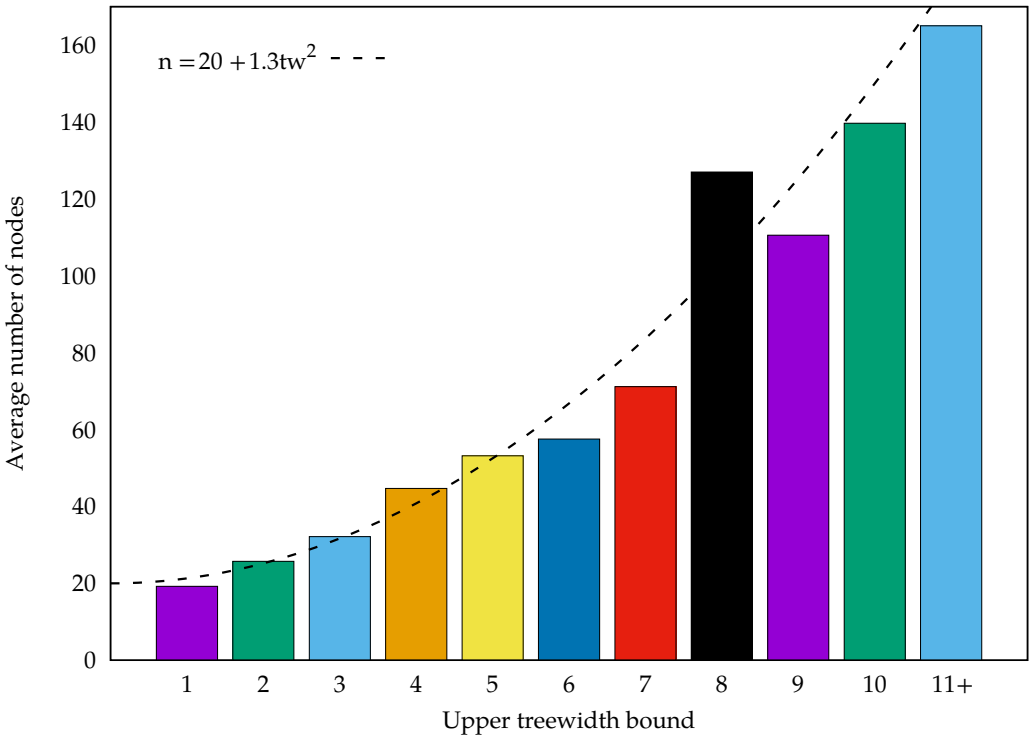

Figure 4: Average number of nodes per upper treewidth bound.

for the number of nodes n, and treewidth tw. We note that of all 1620 regions generated only 4 regions have a treewidth above 10, marked as 11+.

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

Synthetic Benchmark Results

Functions and Arguments

We first look at different treewidth bounds generated by running three semantically equivalent programs, each loading values using the *variable*, *struct* and *array* methods presented in Listing 1 of Section 4. These results are summarized in Table 1, run for functions with 10 parameters. We also give an overview of how these methods affects the structure of the corresponding RVSDG.

Testing a function accepting n number of arguments and returning their sum, we find a gradual increase in treewidth following increases in its number of arguments. We also note that this increase stops after a certain number of arguments is reached. From these experiments we also observe two other factors that affect the treewidth of the function. Firstly, allocating local variables inside the function may increase the upper treewidth bound. Secondly, changing the addition expression inside the function itself also affects the upper treewidth bound.

After treewidth reaches a certain limit, allocating a variable has no effect on the treewidth of the program if the variable is not referenced later in the program. Analyzing variables that are used or referenced in the same function that they are allocated, we find

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 1: Summary of the treewidths correlating to the separate methods of passing arguments to the functions in Listing 1, 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.

that uses of the variable increases the treewidth beyond the maximum value discussed above. Figure 5 shows this continued growth in treewidth for a program allocating 7 integer or matrix 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.

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, but the upper treewidth bound increases in bigger increments. Calling different functions, the treewidth also continues to grow for an increasing number of function calls, resulting in a larger upper and lower treewidth bound.

The presented results are obtained with blockwise ordering of function calls. Analyzing results for sequentially ordered function calls, we found that this ordering produces no increase in treewidth as the number of function calls increases.

Variable liveness

Combining the two approaches presented in this section, simultaneously increasing both the number of allocated variables *and* the number of times they are referenced does not tend toward an upper bound. An excerpt from these results are shown in Figure 6a, where we increase both the number of variables, and the number of times they are referenced. We find that when increasing either parameter in isolation, the treewidth approaches a limit, but when increasing both the treewidth grows indefinitely.

We further investigate how the number of variables allocated and how many times they are used in *expressions* affect the treewidth. The evaluated expression is an addition of all allocated variables. For several variables, this is done in a blockwise order. Figure 6b shows the results, which are similar to the results of increasing the number function calls. When increasing both parameters, the treewidth of the resulting graphs increases without approaching an upper bound.

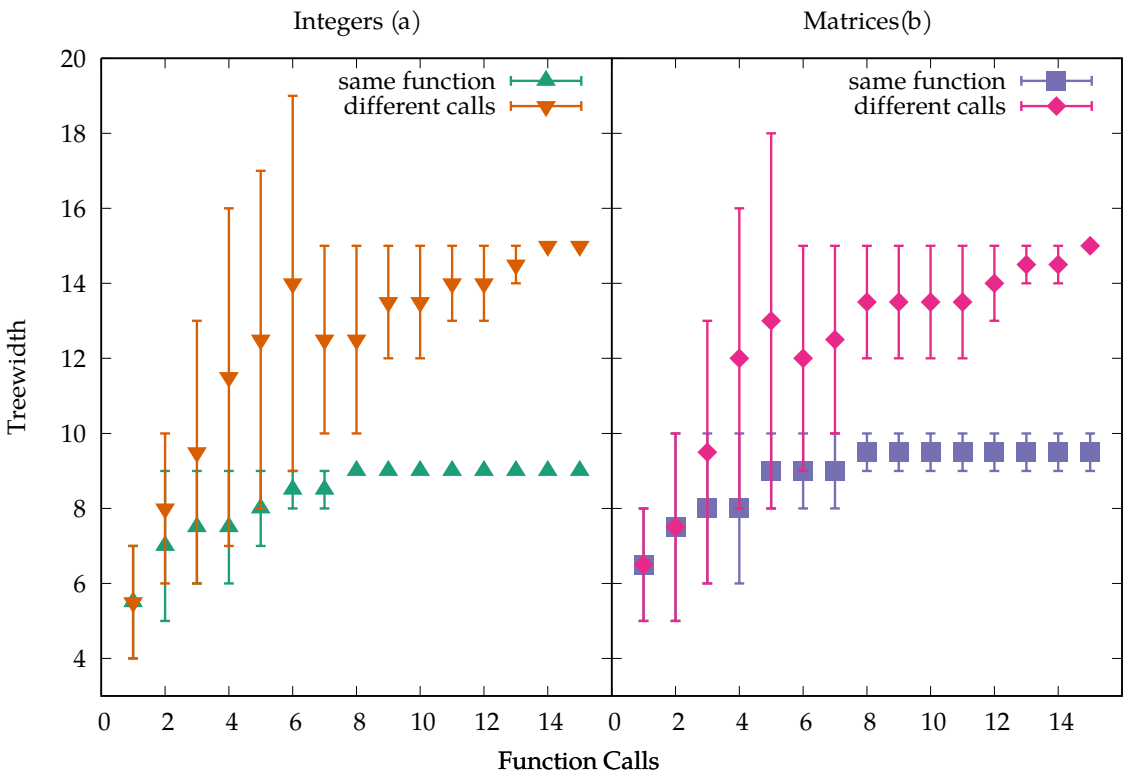

Figure 5: Relationship between the number of references to set of 7 variables allocated in a blockwise call order and the upper and lower treewidth bounds of their corresponding RVSDGs.

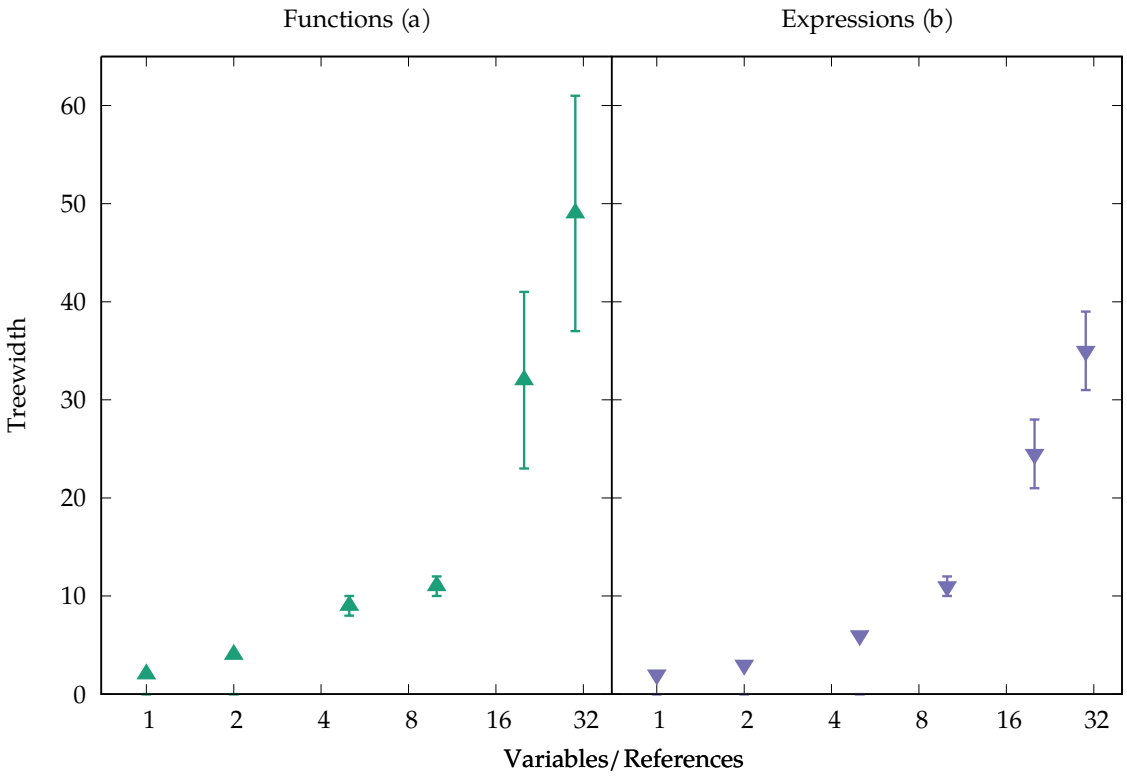

Figure 6: Lower and upper treewidth bounds for both an increasing amount of variables allocated, and number of calls to each variable.

6 Conclusions

In this paper, we have reviewed the Regionalized Value State Dependence graph intermediate representation, implemented efficient heuristics for bounds on its treewidth, and empirically examined its magnitude using a set of application benchmarks, as well as synthetic benchmarks specifically designed to influence RVSDG treewidth.

Our application benchmark results indicate that RVSDG treewidth tends to be low and limited for the practical purposes of application software, which suggests that optimizations that are only computationally feasible within bounded treewidth will still be broadly applicable in an RVSDG based compiler.

Our synthetic benchmark results reveal that certain program features can produce RVSDG representations of unbounded treewidth. Specifically, simultaneous growth in the number of local variables and their number of uses uniformly increases the treewidth.

While scopes with very large local namespaces have limited application in practical software, a compiler must make the conservative assumption that they may occur as valid input, without causing an exponential inflation in compile time. The use of heuristics to establish upper and lower treewidth bounds can, however, serve as a computationally inexpensive detection mechanism, and identify cases when specific bounds are required.

Thefore it is feasible to find the tree decomposition of the RVSDG. This makes possible the implementation of polynomial-time solutions for several NP-complete optimization algorithms using the IR. We conclude this study with the recommendation that future RVSDG-based compiler optimizations requiring bounded treewidth should be implemented along with the *minor-min-width* and *min-fill* heuristics, dynamically using their values to guide whether or not the optimization pass should be enabled.

References

[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] Krste Asanović, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[3] 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.

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

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

[6] 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.

[7] Vibhav Gogate and Rina Dechter. A complete anytime algorithm for treewidth. *Proceedings of the 20th conference on Uncertainty in Artificial Intelligence (UAI'04)*, pages 201–208, 2004.

[8] 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.

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

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

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

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

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

[14] LLVM Project. LLVM. `https://llvm.org`, 2020. The LLVM Compiler Infrastructure.

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

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

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

[18] 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.