RVSDG: An Intermediate Representation for Optimizing Compilers

NICO REISSMANN, Norwegian University of Science and Technology, Norway
JAN CHRISTIAN MEYER, Norwegian University of Science and Technology, Norway
HELGE BAHMANN, Auterion AG, Switzerland
MAGNUS SJÄLANDER, Norwegian University of Science and Technology and Uppsala University, Sweden

Intermediate Representations (IRs) are central to optimizing compilers as the way the program is represented may enhance or limit analyses and transformations. Suitable IRs focus on exposing the most relevant information and establish invariants that different compiler passes can rely on. While control-flow centric IRs appear to be a natural fit for imperative programming languages, analyses required by compilers have increasingly shifted to understand data dependencies and work at multiple abstraction layers at the same time. This is partially evidenced in recent developments such as the MLIR proposed by Google. However, rigorous use of data flow centric IRs in general purpose compilers has not been evaluated for feasibility and usability as previous works provide no practical implementations.

We present the Regionalized Value State Dependence Graph (RVSDG) IR for optimizing compilers. The RVSDG is a data flow centric IR where nodes represent computations, edges represent computational dependencies, and regions capture the hierarchical structure of programs. It represents programs in demand-dependence form, implicitly supports structured control flow, and models entire programs within a single IR. We provide a complete specification of the RVSDG, construction and destruction methods, as well as exemplify its utility by presenting Dead Node and Common Node Elimination optimizations. We implemented a prototype compiler and evaluate it in terms of performance, code size, compilation time, and representational overhead. Our results indicate that the RVSDG can serve as a competitive IR in optimizing compilers while reducing complexity.

CCS Concepts: • **Software and its engineering** → **Compilers**; • **Theory of computation** → Functional constructs; Control primitives.

Additional Key Words and Phrases: Regionalized Value State Dependence Graph, RVSDG, LLVM, Intermediate Representation

ACM Reference Format:
Nico Reissmann, Jan Christian Meyer, Helge Bahmann, and Magnus Själander. 2020. RVSDG: An Intermediate Representation for Optimizing Compilers. *ACM Trans. Embedd. Comput. Syst.* 1, 1, Article 1 (January 2020), 25 pages. https://doi.org/10.1145/3391902

1 INTRODUCTION

Intermediate representations (IRs) are at the heart of every modern compiler. These data structures represent programs throughout compilation, connect individual compiler stages, and provide abstractions to facilitate the implementation of analyses, optimizations, and program transformations. A suitable IR highlights and exposes

Authors' addresses: Nico Reissmann, nico.reissmann@ntnu.no, Norwegian University of Science and Technology, Trondheim, 7491, Norway; Jan Christian Meyer, jan.christian.meyer@ntnu.no, Norwegian University of Science and Technology, Trondheim, 7491, Norway; Helge Bahmann, hcb@chaoticmind.net, Auterion AG, Zürich, 8820, Switzerland; Magnus Själander, magnus.sjalander@ntnu.no, Norwegian University of Science and Technology, Trondheim, 7491, and Uppsala University, Uppsala, 75236, Sweden.

1539-9087/2020/1-ART1 $15.00
https://doi.org/10.1145/3391902

program properties that are important to the transformations in a specific compiler stage. This reduces the complexity of optimizations and simplifies their implementation.

Modern embedded systems have become increasingly parallel as system designers strive to improve their computational power and energy efficiency. Increasing the number of cores in a system enables each core to be operated at a lower clock frequency and supply voltage, improving overall energy efficiency while providing sufficient system performance. Multi-core systems also reduce the total system cost by enabling the consolidation of multiple functionalities onto a single chip. In order to take full advantage of these systems, optimizing compilers need to expose a program's available parallelism. This has led to an interest in developing more efficient program representations [11, 23] and methodologies and frameworks [7] for exposing the necessary information.

Data flow centric IRs, such as the Value (State) Dependence Graph (V(S)DG) [19, 20, 24, 42, 43, 48], show promises as a new class of IRs for optimizing compilers. These IRs are based on the observation that many optimizations require data flow rather than control flow information, and shift the focus to explicitly expose data instead of control flow. They represent programs in demand-dependence form, encode structured control flow, and explicitly model data flow between operations. This raises the IR's abstraction level, permits simple and powerful implementations of data flow optimizations, and helps to expose the inherent parallelism in programs [20, 24, 43]. However, the shift in focus from explicit control flow to only structured and implicit control flow requires more sophisticated construction and destruction methods [24, 42, 48]. In this context, Bahmann et al. [3] presents the *Regionalized Value State Dependence Graph* (RVSDG) and conclusively addresses the problem of intra-procedural control flow recovery for demand-dependence graphs. They show that the RVSDG's restricted control flow constructs do not limit the complexity of the recoverable control flow.

In this work, we are concerned with the aspects of unified program representation in the RVSDG. We present the required RVSDG constructs, consider construction and destruction at the program level, and show feasibility and practicality of this IR for optimizations by providing a practical compiler implementation. Specifically, we make the following contributions: *i)* A complete RVSDG specification, including intra- and inter-procedural constructs. *ii)* A complete description of RVSDG construction and destruction, augmenting the previously proposed algorithms with the construction and destruction of inter-procedural constructs, as well as the handling of intra-procedural dependencies during construction. *iii)* A presentation of Common Node Elimination (CNE) and Dead Node Elimination (DNE) optimizations to demonstrate the RVSDG's utility. CNE permits the removal of redundant computations by detecting congruent operations. DNE combines dead and unreachable code elimination, as well as dead function removal. *iv)* A publicly available prototype compiler [35] that implements the discussed concepts. It consumes and produces LLVM IR, and is to our knowledge the first optimizing compiler that uses a demand dependence graph as IR. *v)* An evaluation of the RVSDG in terms of performance and size of the produced code, as well as compile time and representational overhead.

Our results show that the RVSDG can serve as the IR in a compiler's optimization stage, producing competitive code even with a conservative modeling of programs using a single memory state. Even though this leaves significant parallelization potential unused, it already yields satisfactory results compared to control-flow based approaches. This work paves the way for further exploration of the RVSDG's properties and their effect on optimizations and analyses, as well as its usability in code generation for dataflow and parallel architectures.

2 MOTIVATION

Contemporary optimizing compilers are mainly based on variants of the control flow graph as imperative program representations. These representations preserve sequential execution semantics of the input program, such as access order of aliased references. The LLVM representation is based on the instruction set of a virtual CPU with operation semantics resembling real CPUs. This choice of representation is somewhat at odds with the requirements of code optimization analysis, which often focuses on *data dependence* instead. As Table 1 shows,

most executed optimization passes are concerned with data flow analysis in the form of SSA construction and interpretation, or in-memory data structures in the form of alias analysis and/or memory SSA.

We propose the data-dependence centric RVSDG as an alternative. While it requires more effort to construct the RVSDG from imperative programs and recover control flows for code generation, we believe this cost is more than recovered by benefits to analyses and optimizations. The following sections provide illustrative examples.

2.1 Simplified Compilation by Strong Representation Invariants

Table 1. Thirteen most invoked LLVM 7.0.1 passes at 03.

Optimization	# Invocations
1. Alias Analysis (`-aa`)	19
2. Basic Alias Analysis (`-basicaa`)	18
3. Optimization Remark Emitter (`-opt-remark-emitter`)	15
4. **Natural Loop Information (`-loops`)**	14
5. Lazy Branch Probability Analysis (`-lazy-branch-prob`)	14
6. Lazy Block Frequency Analysis (`-lazy-block-freq`)	14
7. **Dominator Tree Construction (`-domtree`)**	13
8. Scalar Evolution Analysis (`-scalar-evolution`)	10
9. **CFG Simplifier (`-simplifycfg`)**	8
10. Redundant Instruction Combinator (`-instcombine`)	8
11. **Natural Loop Canonicalization (`-loop-simplify`)**	8
12. **Loop-Closed SSA Form (`-lcssa`)**	7
13. **Loop-Closed SSA Form Verifier (`-lcssa-verification`)**	7
Total	155
SSA Restoration	14

The Control Flow Graph (CFG) in Static Single Assignment (SSA) form [12] is the dominant IR for optimizations in modern imperative language compilers [44]. Its nodes represent a list of totally ordered operations, and its edges a program's possible control flow paths, permitting efficient control flow optimizations and simple code generation. The CFG's translation to SSA form improves the efficiency of many data flow optimizations [37, 47]. Figure 1a shows a function with a simple loop and a conditional, and Figure 1b shows the corresponding CFG in SSA form.

SSA form is not an intrinsic property of the CFG, but a specialized variant that must be actively maintained. Compiler passes such as jump threading or live-range splitting may perform transformations that cause the CFG to no longer satisfy this form. As shown in Table 1, LLVM requires SSA restoration [8] in 14 different passes.

Moreover, CFG-based compilers must frequently (re-)discover and canonicalize loops, or establish various invariants besides SSA form. Table 1 shows that six of the 13 most invoked passes in LLVM only perform such tasks, and account for 21% of all invocations. This lack of invariants complicates implementation of optimizations and analyses, increases engineering effort, prolongs compilation time, and leads to compiler bugs [25–27].

In contrast, the RVSDG is always in strict SSA form as edges connect each operand input to only one output. It explicitly exposes program structures such as loops in a tree structure (Section 4), similarly to the Program Structure Tree [21]. This makes SSA restoration and the other helper passes from Table 1 redundant. Figure 1c shows the RVSDG corresponding to Figure 1a. It is an acyclic demand-dependence graph where nodes represent simple operations or control flow constructs, and edges represent dependencies between computations (Section 4). In Figure 1c, simple operations are colored yellow, conditionals are green, loops are red, and functions are blue.

2.2 Unified Representation of Different Levels of Program Structures

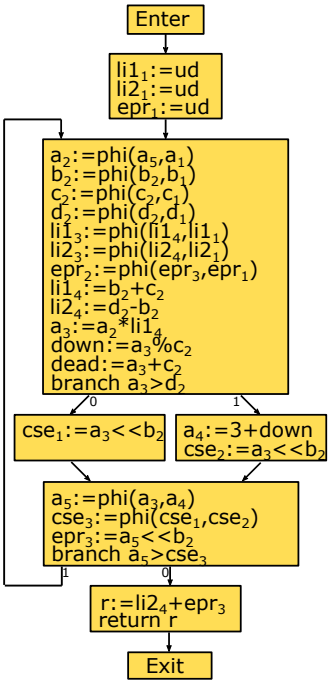
While the CFG can represent a single procedure, representation of programs as a whole requires additional data structures such as call graphs. The RVSDG can represent a program as a unified data structure where a def-use dependency of one function on another is modeled the same way as the def-use dependency of scalar quantities. This makes it possible to apply the same transformation at multiple levels, and considerably reduce the number of transformation passes and algorithms, e.g., by uniform treatment of unreachable code, dead function analysis, and dead variable analysis (Section 6.2).

2.3 Strongly Normalized Representation

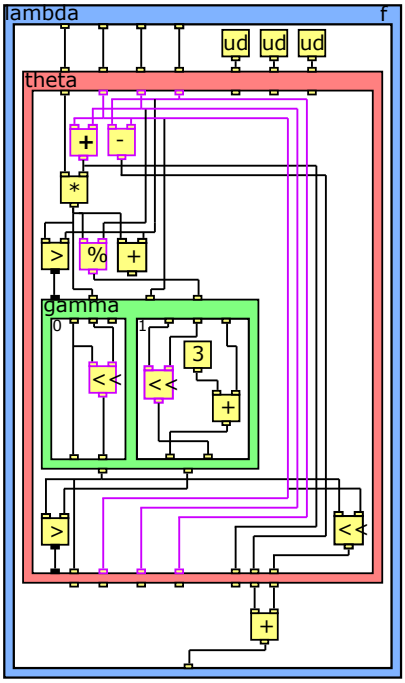
The RVSDG program representation is much more strongly normalized than control flow representations. Programs differing only in the ordering of (independent) operations result in the same RVSDG representation, while state edges ensure the correct evaluation order of stateful computations. Loops and conditionals always take

```
int
f(int a, int b, int c, int d)
{
  int li1, li2;
  int cse, epr;
  do {
    li1 = b+c;
    li2 = d-b;
    a = a*li1;
    int down = a%c;
    int dead = a+d;
    if(a > d) {
      int acopy = a;
      a = 3+down;
      cse = acopy<<b;
    } else {
      cse = a<<b;
    }
    epr = a<<b;
  } while(a > cse);
  return li2+epr;
}
```

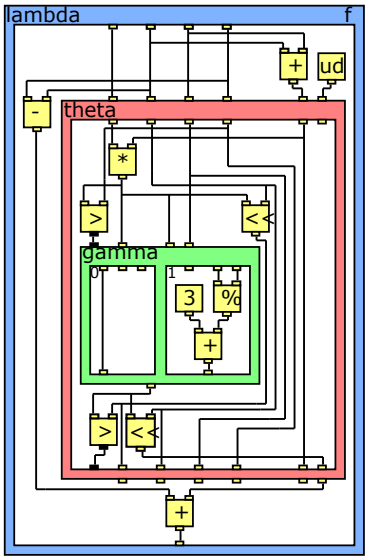
(a) Code



(b) CFG in SSA form



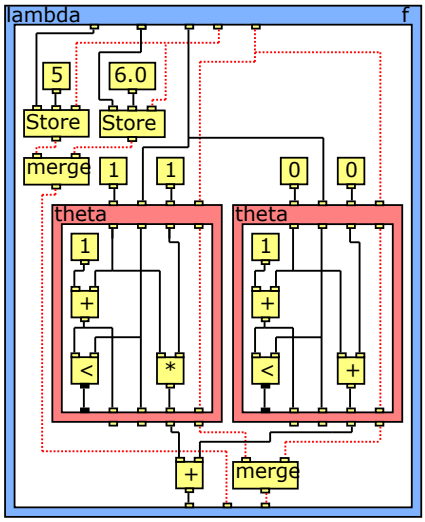
(c) Unoptimized RVSDG



(d) Optimized RVSDG

```
int
f(int* x, float* y, int k)
{
  *x = 5;
  *y = 6.0;
  int i=0;
  int f=1;
  int sum=0;
  int fac=1;
  do {
   sum += i;
   i++;
  } while(i < k);
  do {
    fac *= f;
    f++;
  } while(f < k);
  return fac+sum;
}
```

(e) Code



(f) RVSDG of Code 1e

Fig. 1. RVSDG Examples

a single canonical form. These normalizations already simplify the implementation of transformations [20, 24, 48] and eliminate the need for (repeated) compiler analysis passes such as loop detection.

Some common program optimizing transformations take a particular simple form in the RVSDG representation. For example, Figure 1d shows the optimized RVSDG of Figure 1c, illustrating some of these optimizations: The inputs to the "upper left" plus operation are easily recognized as loop invariant because their "loop entry ports" connect directly to the corresponding "loop exit ports" (operations, ports, and edges highlighted in purple). A simple push strategy allows to recursively identify data dependent operations as invariant and hoist them out of the loop: The addition and subtraction computing `li1` and `li2` are moved out of the loop (theta) as their operands, i.e., `b`, `c`, and `d`, are loop invariant (all three of them connect the entry of the loop to the exit). Similarly, the shift operation common to both conditional branches is hoisted and combined, while the division operation is moved into the conditional as it is only used in one alternative. In contrast to CFG-based compilers, all these optimizations are performed directly on the unoptimized RVSDG of Figure 1c and can be performed in a single regular pass. No additional data structures or helper passes are required. See also Section 6 for further details.

2.4 Exposing Independent Computations

CFGs implicitly represent a single global machine state by sequencing every operation that affects it. While RVSDGs can model the same machine, they are not limited to this interpretation. RVSDGs can also model systems consisting of multiple *independent* states. Figures 1e and 1f illustrate this concept with a function that contains two non-aliasing store operations (targeting memory objects of incompatible types) and two independent loops.

In a CFG, both stores and loops are strictly ordered. Their mutual independence needs to be established by explicit compiler passes (and may need to be re-established multiple times during the compilation process as the number of alias analysis passes in Table 1 illustrate) and represented using auxiliary data structures and/or annotations. In contrast, the RVSDG can encode such information directly in the graph, as shown in Figure 1f. Disjoint memory regions (consisting of `int`-typed and `float`-typed memory objects) are modeled as disjoint states, exposing the independence of affecting operations in the representation. RVSDG can in principle go even further in representing a *memory SSA* form that is not formally any different from *value SSA* form, enabling the same kind of optimizations to be applied to both.

2.5 Multiple Levels of Abstraction

The RVSDG can contain operational nodes at vastly different abstraction levels: operational nodes may closely correspond to "source code" level constructs operating on data structures modeled as state, or may map to individual machine instructions affecting machine and memory state. This allows compilers to be structured so that they preserve considerably more source code semantics and utilize it at any later stage in the translation. The contents of two distinct `std::vector` instances can e.g. never alias by language semantics, but this fact is lost in present-day compilers due to early lowering into a machine-like representation that discards high-level semantics. The RVSDG does not have such limitations (vector contents can be modeled as independent states from the beginning), can optimize at multiple abstraction levels, and can preserve vital invariants across abstraction levels. We expect this effect to become particularly pronounced the more input programs are formulated above the abstraction level of the C language, e.g., functional languages or languages expressing contracts on affected state.

2.6 Summary

The RVSDG raises the IR abstraction level by enforcing desirable properties, such as SSA form, explicitly encoding important structures, such as loops, and relaxing the strict order of the input program. This gives a more normalized program representation, avoids many idiosyncrasies and artifacts of other IRs, and exposes parallelism in programs.

3 RELATED WORK

A cornucopia of IRs has been presented in the literature to better expose desirable program properties for optimizations. For brevity, we restrict our discussion to the most prominent IRs, only highlighting their strengths and weaknesses in comparison to the RVSDG, and refer the reader to Stanier et al. [44] for a greater survey.

3.1 Control (Data) Flow Graph

The Control Flow Graph (CFG) [1] exposes the intra-procedural control flow of a function. Its nodes represent basic blocks, i.e., an ordered list of operations without branches or branch targets, and its edges represent the possible control flow paths between these nodes. This explicit exposure of control flow simplifies certain analyses, such as loop identification or irreducibility detection, and enables simple target code generation. The CFG's translation to SSA form [12], or one of its variants, such as gated SSA [46], thinned gated SSA [16], or future gated SSA [14], additionally improves the efficiency of data flow optimizations [37, 47]. These properties along with its simple construction from a language's abstract syntax tree made the CFG in SSA form the predominant IR for imperative language compilers [44], such as LLVM [22] and GCC [10]. However, the CFG has also been criticized as an IR for optimizing compilers [15, 19, 20, 24, 48–50]:

(1) It is incapable of representing inter-procedural information. It requires additional IRs, e.g., the call graph, to represent such information.
(2) It provides no structural information about a procedure's body. Important structures, such as loops, and their nesting needs to be constantly (re-)discovered for optimizations, as well as normalized to make them amenable for transformations.
(3) It emphasizes control dependencies, even though many optimizations are based on the flow of data. This is somewhat mitigated by translating it to SSA form or one of its variants, but in turn requires SSA restoration passes [8] to ensure SSA invariants.
(4) It is an inherently sequential IR. The operations in basic blocks are listed in a sequential order, even if they are not dependent on each other. Moreover, this sequentialization also exists for structures such as loops, as two independent loops can only be represented in sequential order. Thus, the CFG is by design incapable of explicitly encoding independent operations.
(5) It provides no means to encode additional dependencies other than control and true data dependencies. Other information, such as loop-carried dependencies or alias information, must regularly be recomputed and/or memoized in addition to the CFG.

The Control Data Flow Graph (CDFG) [30] tries to mitigate the sequential nature of the CFG by replacing the sequence of operations in basic blocks with the Data Flow Graph (DFG) [13], an acyclic graph that represents the flow of data between operations. This relaxes the strict ordering within a basic block, but does not expose instruction level parallelism beyond basic block boundaries or between program structures.

3.2 Program Dependence Graph/Web

The Program Dependence Graph (PDG) [15, 17] combines control and data flow within a single representation. It features data and control flow edges, as well as statement, predicate, and region nodes. Statement nodes represent operations, predicate nodes represent conditional choices, and region nodes group nodes with the same control dependency. If a region's control dependencies are fulfilled, then its children can be executed in parallel. Horwitz et al. [18] extended the PDG to model inter-procedural dependencies by incorporating procedures into the graph.

The PDG improves upon the CFG by employing region nodes to relax the overly restrictive sequence of operations. This relaxed sequence combined with the unified representation of data and control dependencies simplifies complex optimizations, such as code vectorization [4] or the extraction of thread-level parallelism [32, 39]. However, the unified data and control flow representation results in a large number of edge types, five in

Ferrante et al. [15] and four in Horwitz et al. [17], which need to be maintained to ensure the graph's invariants. The PDG suffers from aliasing and side-effect problems, as it supports no clear distinction between data held in registers and memory. This complicates or can even preclude its construction altogether [20]. Moreover, program structure and SSA form still need to be discovered and maintained.

The Program Dependence Web (PDW) [31] extends the PDG and gated SSA [46] to provide a unified representation for the interpretation of programs using control-, data-, or demand-driven execution models. This simplifies the mapping of programs written in different paradigms, such as the imperative or functional paradigm, to different architectures, such as Von-Neumann and dataflow architectures. In addition to the elements of the PDG, the PDW adds μ nodes to manage initial and loop-carried values and η nodes to manage loop-exit values. Campbell et al. [5] further refined the definition of the PDW by replacing μ nodes with β nodes and eliminating η nodes. As the PDW is based on the PDG, it suffers from the same aliasing and side-effect problems. The PDW's additional constructs further complicate graph maintenance and its construction is elaborate, requiring three additional passes over a PDG, and is limited to programs with reducible control flow.

3.3 Value (State) Dependence Graph

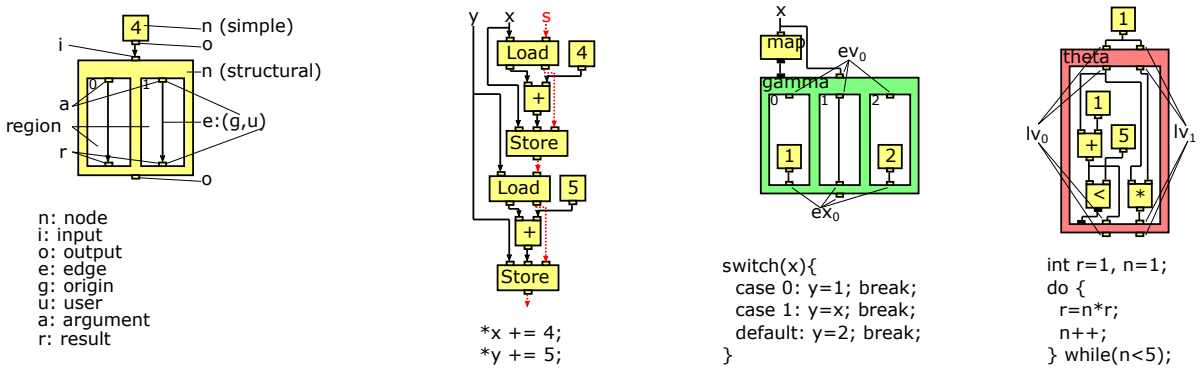
The Value Dependence Graph (VDG) [48] abandons the explicit representation of control flow and only models the flow of values using ports. Its nodes represent simple operations, the selection between values, or functions, using recursive functions to model loops. The VDG is implicitly in SSA form and abandons the sequential order of operations from the CFG, as each node is only dependent on its values. However, modeling only data flow between stateful computations raises a significant problem in terms of preservation of program semantics, as the "evaluation of the VDG may terminate even if the original program would not..." [48].

The Value State Dependence Graph (VSDG) [19, 20] addresses the VDG's termination problem by introducing state edges. These edges are used to model the sequential execution of stateful computations. In addition to nodes for representing simple operations and selection, it introduces nodes to explicitly represent loops. Like the VDG, the VSDG is implicitly in SSA form, and nodes are solely dependent on required operands, avoiding a sequential order of operations. However, the VSDG supports no inter-procedural constructs, and its selection operator is only capable of selecting between two values based on a predicate. This complicates destruction, as selection nodes must be combined to express conditionals. Even worse, the VSDG represents all nodes as a flat graph, which simplifies optimizations [20], but has a severe effect on evaluation semantics. Operations with side-effects are no longer guarded by predicates, and care must be taken to avoid duplicated evaluation of these operations. In fact, for graphs with stateful computations, lazy evaluation is the only safe strategy [24]. The restoration of a program with an eager evaluation semantics complicates destruction immensely, and requires a detour over the PDG to arrive at a unique CFG [24]. Zaidi et al. [49, 50] adapted the VSDG to spatial hardware and sidestepped this problem by introducing a predication-based eager/dataflow semantics. The idea is to effectively enforce correct evaluation of operations with side-effects by using predication. While this seems to circumvent the problem for spatial hardware, it is unclear what the performance implications would be for conventional processors.

The RVSDG solves the VSDG's eager evaluation problem by introducing regions. These regions enable the modeling of control flow constructs as nested nodes, and the guarding of operations with side-effects. This avoids any possibility of duplicated evaluation, and in turn simplifies RVSDG destruction. Moreover, nested nodes permit the explicit encoding of a program's hierarchical structure into the graph, further simplifying optimizations.

4 THE REGIONALIZED VALUE STATE DEPENDENCE GRAPH

A Regionalized Value State Dependence Graph (RVSDG) is an acyclic hierarchical multigraph consisting of nested regions. A region $\mathcal{R} = (A, N, E, R)$ represents a computation with argument tuple A, nodes N, edges E, and result tuple R, as illustrated in Figure 2a. A node can be either *simple*, i.e., it represents a primitive operation, or *structural*,



(a) Notation (b) Simple nodes (c) γ-node (d) θ-node

Fig. 2. Notation and examples of simple, γ- and θ-nodes. Fig. 2c annotates the γ-node's only entry and exit variable. Fig. 2d annotates the θ-node's two loop variables.

i.e., it contains regions. Each node $n \in N$ has a tuple of inputs I and outputs O. For simple nodes they correspond to arguments and results of the represented operation, whereas for structural nodes they map to arguments and results of contained regions. For nodes $n_1, n_2 \in N$, an edge $(g, u) \in E$ connects either output $g \in O_{n_1}$ or argument $g \in A$ to either input $u \in I_{n_2}$ or result $u \in R$ of matching type. We refer to g as the *origin* of an edge, and to u as the *user* of an edge. Every input or result is the user of *exactly one* edge, whereas outputs or arguments can be the origins of multiple edges. All inputs or results of an origin are called its *users*. The corresponding node of an origin is called its *producer*, whereas the corresponding node of a user is called *consumer*. Correspondingly, the set of nodes of all users of an origin are referred to as its *consumers*. The types of inputs and outputs are either *values*, representing arguments or results of computations, or *states*, used to impose an order on operations with side-effects. A node's *signature* is the types of its inputs and outputs, whereas a region's signature is the types of its arguments and results. Throughout this paper, we use n, e, i, o, a, and r with sub- and superscripts to denote individual **n**odes, **e**dges, **i**nputs, **o**utputs, **a**rguments, and **r**esults, respectively. We use g and u to denote an edge's ori**g**in and **u**ser, respectively. An edge e from origin g to user u is also denoted as $e : (g, u)$, or short (g, u).

The RVSDG can model programs at different abstraction levels. It can represent simple data-flow graphs such as those used in machine learning frameworks, as well as machine level programs used for code generation in compiler back-ends. This flexibility makes it possible to use the RVSDG for the entire compilation pipeline. In this paper, we target an abstraction level similar to that of LLVM IR. This permits us to illustrate all of the RVSDG's features without involving architecture-specific details. The rest of this section defines the necessary constructs.

4.1 Intra-Procedural Nodes

This section defines the nodes for representing the intra-procedural aspects of programs. It explains simple nodes and discusses the two structural nodes required for modeling intra-procedural program behavior:

(1) *Gamma-Nodes* model conditionals with symmetric split and joins, such as `if-then-else` statements.
(2) *Theta-Nodes* represent tail-controlled loops, i.e., `do-while` loops.

4.1.1 Simple nodes. Simple nodes model primitive operations such as addition, subtraction, load, and store. They have an operator associated with them, and a node's signature must correspond to the signature of its operator. Simple nodes map their input value tuple to their output value tuple by evaluating their operator with the inputs

as arguments, and associating the results with their outputs. Figure 2b illustrates the use of simple nodes as well as value and state edges. Solid lines represent value edges, whereas dashed red lines represent state edges. Nodes have as many value inputs and outputs as their corresponding operations demand. The ordering of the load and store nodes is preserved by sequentializing them with the help of state edges.

4.1.2 Gamma-Nodes. A γ-node models a decision point and contains regions $\mathcal{R}_0, ..., \mathcal{R}_k \mid k > 0$ of matching signature. Its first input is a *predicate*, which determines the region under evaluation. It evaluates to an integer v with $0 \leq v \leq k$. The values of all other inputs are mapped to the corresponding arguments of region $\mathcal{R}_v$, $\mathcal{R}_v$ is evaluated, and the values of its results are mapped to the outputs of the γ-node.

γ-nodes represent conditionals with symmetric control flow splits and joins, such as `if-then-else` or `switch` statements without fall-throughs. Figure 2c shows a γ-node. It contains three regions: one for each case, and a default region. The map node takes the value of x as input and maps it to zero, one, or two, determining the region under evaluation. This region is evaluated and its result is mapped to the γ-node's output.

We define the *entry variable* of a γ-node as a pair of an input and the arguments the input maps to during evaluation, and the *exit variable* as a pair of an output and the results the output could receive its value from:

Definition 1. *The pair $ev_l = (i_{l+1}, A_l)$ is the l-th* entry variable *of a γ-node with $k+1$ regions. It consists of the $l+1$-th input and tuple $A_l = \{a_l^{\mathcal{R}_0}, ..., a_l^{\mathcal{R}_k}\}$ with the l-th argument from each region. We refer to the set of all entry variables as EV.*

Definition 2. *The pair $ex_l = (R_l, o_l)$ is the l-th* exit variable *of a γ-node with $k+1$ regions. It consists of a tuple $R_l = \{r_l^{\mathcal{R}_0}, ..., r_l^{\mathcal{R}_k}\}$ of the l-th result from each region and the l-th output they would map to. We refer to the set of all exit variables as EX.*

4.1.3 Theta-Nodes. A θ-node models a tail-controlled loop. It contains one region that represents the loop body. The length and signature of its input tuple equals that of its output, or the region's argument tuple. The first region result is a *predicate*. Its value determines the continuation of the loop. When a θ-node is evaluated, the values of all its inputs are mapped to the corresponding region arguments and the body is evaluated. When the predicate is true, all other results are mapped to the corresponding arguments for the next iteration. Otherwise, the result values are mapped to the corresponding outputs. The loop body of an iteration is always fully evaluated before the evaluation of the next iteration. This avoids "deadlock" problems between computations of the loop body and the predicate, and results in well-defined behavior for non-terminating loops that update external state.

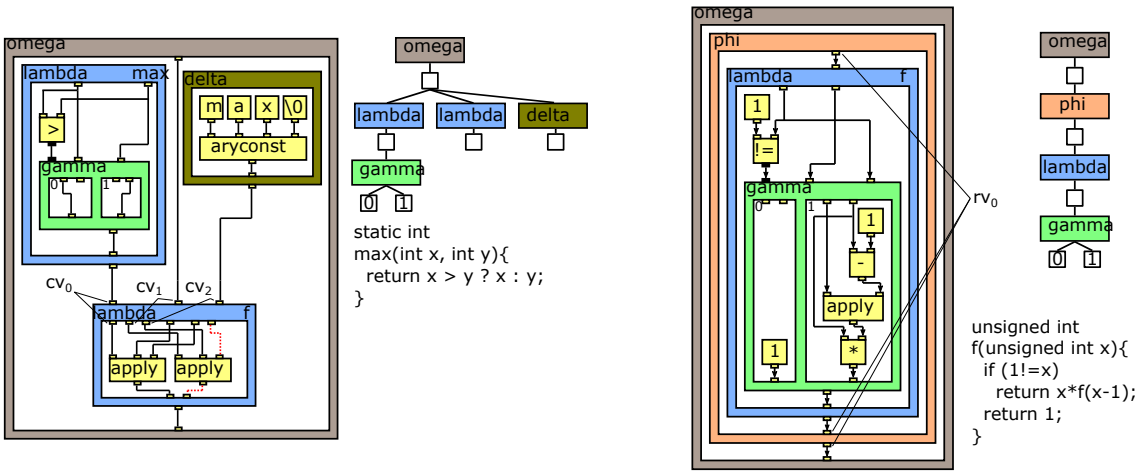
θ-nodes permit the representation of `do-while` loops. In combination with γ-nodes, it is possible to model head-controlled loops, i.e., `for` and `while` loops. Thus, employing tail-controlled loops as basic loop construct enables us to express more complex loops as a combination of basic constructs. This normalizes the representation and reduces the complexity of optimizations as there exists only one construct for loops. Another benefit of tail-controlled loops is that their body is guaranteed to execute at least once, enabling the unconditional hoisting of invariant code with side-effects.

Figure 2d shows a θ-node with two loop variables, n and r, and an additional result for the predicate. When the predicate evaluates to true, the results for n and r of the current iteration are mapped to the region arguments to continue with the next iteration. When the predicate evaluates to false, the loop exits and the results are mapped to the node's outputs. We define a *loop variable* as a quadruple that represents a value routed through a θ-node:

Definition 3. *The quadruple $lv_l = (i_l, a_l, r_{l+1}, o_l)$ is the l-th* loop variable *of a θ-node. It consists of the l-th input i_l, argument a_l, and output o_l, and the $l+1$-th result of a θ-node. We refer to the set of all loop variables as LV.*

4.2 Inter-Procedural Nodes

This section defines the four structural nodes used for modeling the inter-procedural aspects of programs:



(a) RVSDG with λ- and δ-nodes

(b) RVSDG with a ϕ-node

Fig. 3. Usage of λ-, δ-, and ϕ-nodes, and corresponding region trees. Fig. 3b annotates the ϕ-node's recursion variable.

(1) *Lambda-Nodes* are used for modeling procedures and functions.
(2) *Delta-Nodes* model global variables.
(3) *Phi-Nodes* represent mutually recursive environments, such as (mutually) recursive functions.
(4) *Omega-Nodes* represent translation units.

4.2.1 Lambda-Nodes. A λ-node models a function and contains a single region representing a function's body. It features a tuple of inputs and a single output. The inputs refer to external variables the λ-node depends on, and the output represents the λ-node itself. The region has a tuple of arguments comprised of a function's external dependencies and its arguments, and a tuple of results corresponding to a function's results.

An *apply*-node represents a function invocation. Its first input takes a λ-node's output as origin, and all other inputs represent the function arguments. In the rest of the paper, we refer to an *apply*-node's first input as its *function input*, and to all its other inputs as its *argument inputs*. Invocation maps the values of a λ-node's input k-tuple to the first k arguments of the λ-region, and the values of the function arguments of the *apply*-node to the rest of the arguments of the λ-region. The function body is evaluated and the values of the λ-region's results are mapped to the outputs of the *apply*-node.

Figure 3a shows an RVSDG with two λ-nodes. Function f calls functions *puts* and *max* with the help of *apply*-nodes. The function *max* is part of the translation unit, while *puts* is external and must be imported (see the paragraph about ω-nodes for more details). We further define the *context variable* of a λ-node. A context variable provides the corresponding input and argument for a variable that a λ-node depends on.

Definition 4. *The pair $cv_l = (i_l, a_l)$ is a λ-node's l-th* context variable. *It consists of the l-th input and argument. We refer to the set of all context variables as CV.*

Figure 3a shows the three context variables of fuction f annotated: one for function *max*, one for function *puts*, and one for the global variable representing the string argument to *puts*.

Definition 5. *The λ-node connected to a function input is the* callee *of an apply-node, and an apply-node is the* caller *of a λ-node. We refer to the set of all callers of a λ-node as CLL.*

4.2.2 Delta-Nodes. A δ-node models a global variable and contains a single region representing the constants' value. It features a tuple of inputs and a single output. The inputs refer to the external variables the δ-node depends on, and the output represents the δ-node itself. The region has a tuple of arguments representing a global variable's external dependencies and a single result corresponding to its right-hand side value.

Figure 3a shows an RVSGD with a δ-node. Function *puts* takes a string as argument that is the right-hand side of a global variable. Similarly to λ-nodes, we define the *context variable* of a δ-node. It provides the corresponding input and argument for a variable a δ-node depends on.

Definition 6. *The pair $cv_l = (i_l, a_l)$ is a δ-node's l-th* context variable. *It consists of the l-th input and argument. We refer to the set of all context variables as CV.*

4.2.3 Phi-Nodes. A ϕ-node models an environment with mutually recursive functions, and contains a single region with λ-nodes. Each single output of these λ-nodes serves as origin to a single result in the ϕ-region. A ϕ-node's outputs expose the individual functions to callers outside the ϕ-region, and must therefore have the same arity and signature as the results of the ϕ-region. The first input of an *apply*-node from *outside* the ϕ-region takes these outputs as origin to invoke one of the functions.

The inputs of a ϕ-node refer to variables that the contained functions depend on and are mapped to corresponding arguments in the ϕ-region when a function is invoked. In addition, a ϕ-region has arguments for each contained function. An *apply*-node from inside a ϕ-region takes these as origin to its function input.

ϕ-nodes permit a program's mutually recursive functions to be expressed in the RVSDG without the introduction of cycles. Figure 3b shows an RVSDG with a ϕ-node. The function f calls itself, and therefore needs to be in a ϕ-node to preserve the RVSDG's acyclicity. The region in the ϕ-node has one input, representing the declaration of f, and one output, representing the definition of f. The ϕ-node has one output so that f can be called from outside the recursive environment.

We define *context variables* and *recursion variables*. Context variables provide corresponding inputs and arguments for variables the λ-nodes from within a ϕ-region depend on. Recursion variables provide the argument and output an *apply*-node's function input connects to.

Definition 7. *The pair $cv_l = (i_l, a_l)$ is the l-th* context variable *of a ϕ-node. It consists of the l-th input and argument. We call the set of all context variables CV.*

Definition 8. *For a ϕ-node with n context variables, the triple $rv_l = (r_l, a_{l+n}, o_l)$ is the l-th* recursion variable. *It consists of the l-th result and $l + n$-th argument of the ϕ-region as well as the l-th output of the ϕ-node. We refer to the set of all recursion variables as RV.*

4.2.4 Omega-Nodes. An ω-node models a translation unit. It is the top-level node of an RVSDG and has no inputs or outputs. It contains exactly one region. This region's arguments represent entities that are external to the translation unit and therefore need to be imported. Its results mark all exported entities in the translation unit. Figure 3a and 3b illustrate the usage of ω-nodes. The ω-region in Figure 3a has one argument, representing the import of function `puts`, and one result, representing the export of function `f`. The ω-region in Figure 3b has only one export for function `f`.

4.3 Edges

Edges connect node outputs or region arguments to a node input or region result, and are either value typed, i.e., represent the flow of data between computations, or state typed, i.e., impose an ordering on operations with side-effects. State edges are used to preserve the observational semantics of the input program by ordering its side-effecting operations. Such operations include memory read and writes, as well as exceptions.

In practice, a richer type system permits further distinction between different kind of values or states. For example, different types for fixed- and floating-point values helps to distinguish between these arithmetics, and a type for functions permits to correctly specify the output types of λ-nodes and the function input of *apply*-nodes.

5 CONSTRUCTION & DESTRUCTION

RVSDG construction and destruction generate an RVSDG from an input program and reestablish control flow for code generation, respectively. We present both stages with an *Inter-Procedure Graph* (IPG) and a CFG as input and output. The IPG is an extension of a call graph and captures all static dependencies between functions and global variables, incorporating not only those originating from (direct) calls, but also those from other references within a function. In the IPG, an edge from node $n1$ to node $n2$ exists, if the body of a function/global variable corresponding to $n1$ references a function/global variable represented by $n2$. The utilization of an IPG and a CFG permits a language-independent presentation of RVSDG construction and destruction.

5.1 Construction

RVSDG construction maps all constructs, concepts, and abstractions of an input language to the RVSDG. The mapping is language-specific, and depends on the language's features. Languages that permit unstructured control flow, such as C or C++, cannot be mapped directly to the RVSDG and require a CFG as a stepping stone, while languages such as Haskell permit direct construction [34]. In this section, we present RVSDG construction for the former case as it encompasses the latter. Conceptually, RVSDG construction can be split in two phases:

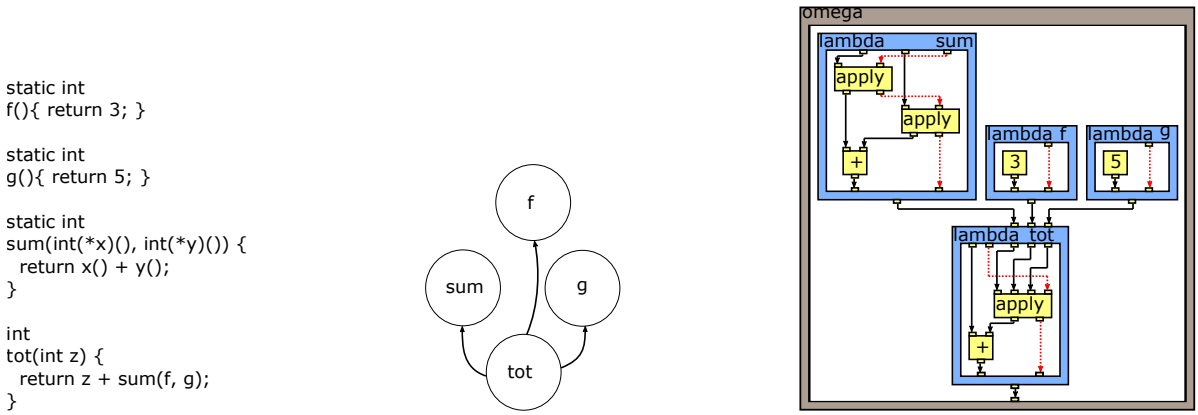
(1) *Inter-Procedural Translation* (Inter-PT) translates functions, global variables, and inter-procedural dependencies, creating λ-, δ-, and ϕ-nodes.
(2) *Intra-Procedural Translation* (Intra-PT) translates intra-procedural control and data flow, creating a δ-/λ-region from a function's/global variables' body.

Inter-PT invokes Intra-PT for each function's or global variables' body, and both phases interact through a common symbol table. The table maps function and CFG variables to RVSDG arguments or outputs, and is updated with every creation of a node or region. We omit the updates from our algorithm descriptions for brevity.

5.1.1 Inter-Procedural Translation. Inter-PT converts all functions and global variables from the *Inter-Procedure Graph* (IPG) of a translation unit to λ-nodes and δ-nodes, respectively. Figure 4b shows the IPG for the code in Figure 4a. The code consists of four functions, with function `sum` performing two indirect calls. The corresponding IPG consists of four nodes and three edges. All edges originate from node *tot*, as it is the only function that explicitly references other functions, i.e. `sum` for a direct call, and `f` and `g` to pass as argument. No edge originates from node `sum`, as the corresponding function does not explicitly reference any other functions, and the functions for the indirect calls are provided as arguments.

The RVSDG puts two constraints on the translation from an IPG. Firstly, mutually recursive entities need to be created within ϕ-nodes to preserve the RVSDG's acyclicity. Secondly, Inter-PT must respect the calling dependencies of functions to ensure that λ-nodes are created before *apply*-nodes. In order to embed mutually recursive entities into ϕ-nodes, we need to identify the strongly connected components (SCCs). We consider an SCC *trivial*, if it consists of a single node with no self-referencing edges. Otherwise, it is *non-trivial*. Moreover, a trivial SCC might not have a CFG associated with it, and is therefore defined in another translation unit.

Algorithm I outlines the RVSDG construction from an IPG. It finds all SCCs and converts trivial SCCs to individual δ-/λ-nodes, while the δ-/λ-nodes created from non-trivial SCCs are embedded in ϕ-nodes. This satisfies the first constraint. The second constraint is satisfied by processing SCCs in topological order, creating λ-nodes before their *apply*-nodes. Identification and ordering of SCCs can be done in a single step with Tarjan's algorithm [45], which returns identified SCCs in reverse topological order. Figure 4c shows the RVSDG after application of Algorithm I to the IPG in Figure 4b. In addition to a function's arguments, Algorithm I adds a state



(a) Code (b) IPG (c) RVSDG

Fig. 4. Inter-Procedural Translation

Algorithm I: Inter-Procedural Translation

Compute all SCCs in an IPG and process them in topological order of the directed acyclic graph formed by the SCCs as follows:

(1) ***Trivial SCC****:*

(a) ***Function with CFG****: Begin a λ-node by adding all context variables, function arguments, and an additional state argument to the λ-region. Translate the CFG with Intra-PT as explained in Section 5.1.2, and finish the λ-node by adding the function results and the state result to the λ-region. If a function is exported, add a result to the ω-region and connect the λ-node's output to it.*

(b) ***Global variable with CFG****: Begin a δ-node by adding all context variables to the δ-region. Translate the CFG with Intra-PT as eplained in Section 5.1.2, and finish the δ-node by adding the result to the δ-region. If a global variable is exported, add a result to the ω-region and connect the δ-node's output to it.*

(c) ***Without CFG****: Add a ω-region argument for the external entity.*

(2) ***Non-trivial SCC****: Begin a ϕ-node by adding all functions/global variables as well as context variables to the ϕ-region. Translate each entity in the SCC according to Trivial SCC without exporting them. Finish the ϕ-node by adding all outputs as results to the ϕ-region. If an entity is exported, add a result to the ω-region and connect the ϕ-node's output to it.*

argument and result to λ-regions (the red dashed line in Figure 4c), to sequence stateful computations. Nodes representing operations with side-effects consume this state and produce a new state for the next node.

5.1.2 Intra-Procedural Translation. The RVSDG puts several constraints on the translation of intra-procedural control and data flow. Firstly, it requires that the control flow only consists of constructs that can be translated to γ- and θ-nodes, i.e. it can only consist of tail-controlled loops and conditionals with symmetric control flow splits and joins. Secondly, the nesting and relation of these constructs to each other is required as the RVSDG is a hierarchical representation. Thirdly, it is necessary to know the data dependencies of these structures in order to construct γ- and θ-nodes. While these constraints are beneficial for optimizations by substantially simplifying their implementation, they render RVSDG construction non-trivial.

This section's construction algorithm enables the translation of any data and control flow, irregardless of its complexity, to the RVSDG. It creates a δ-/λ-region from a global variables' or function's body in four stages:

(1) *Control Flow Restructuring* (CFR) restructures a CFG to make it amenable to RVSDG construction.
(2) *Structural Analysis* constructs a control tree [29], discovering the CFG's individual control flow regions.
(3) *Demand Annotation* annotates the discovered control flow regions with the variables that are demanded by the instructions within these regions.

(4) *Control Tree Translation* converts the annotated control tree into a δ-/λ-region.

CFR ensures the first requirement by translating a function's control flow to a form that is amenable to RVSDG construction. It restructures control flow to a form that enables the direct mapping of a CFG's control flow regions to the RVSDG's γ- and θ-nodes. CFR can be omitted for languages with limited control flow structures, such as Haskell or Scheme. Structural analysis ensures the second requirement by constructing a control tree from the CFG, exposing the control regions nesting and the relation to each other. Demand annotation fulfills the third requirement by annotating the control tree's nodes with their data dependencies. Finally, the annotated control tree can be translated to a δ-/λ-region. The rest of this section covers the four stages in detail.

Control Flow Restructuring: CFR converts a CFG to a form that only contains tail-controlled loops and conditionals with properly nested splits and joins. This stage is only necessary for languages that support more complex control flow constructs, such as `goto` statements or short-circuit operators, but can be omitted for languages with more limited control flow. CFR consists of two interlocked phases: loop restructuring and branch restructuring. Loop restructuring transforms all loops to tail-controlled loops, while branch restructuring ensures conditionals with symmetric control flow splits and joins. We omit an extensive discussion of CFR as it is detailed in Bahmann et al. [3]. In contrast to node splitting approaches [51], CFR avoids the possibility of exponential code blowup [6] by inserting additional predicates and branches instead of cloning nodes. Moreover, it does not require a CFG in SSA form as this form is automatically established throughout construction.

Structural Analysis: After CFR, a restructured CFG consists of 3 single-entry/single-exit control flow regions:

- *Linear Region*: A linear subgraph where the entry node and all intermediate nodes have only one outgoing edge, and the exit node as well as all intermediate nodes have only one incoming edge.
- *Branch Region*: An subgraph with the entry and exit node representing the control flow split and join, respectively, and each branch alternative consisting of a single node.
- *Loop Region*: A single node where an edge originates and targets this node.

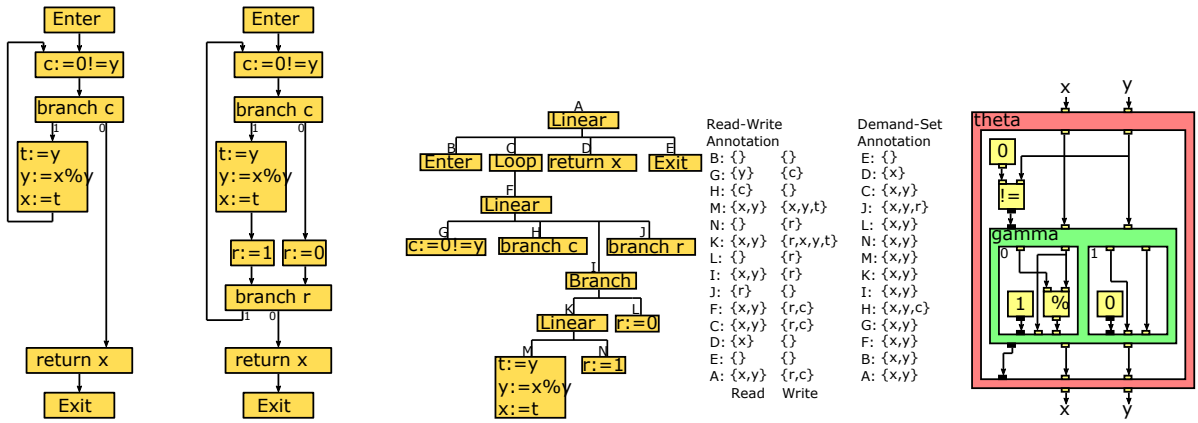
These control flow regions and their corresponding nesting structure can be exposed by performing an interval [29] or structural [40] analysis. The analysis result is a control tree [29] with basic blocks as leaves and abstract nodes representing the control flow regions as branches.

A linear region maps to a *linear node* in the control tree with the linear subgraph's entry and exit node as the node's left and right most child, respectively. A branch region maps to two control tree nodes: a *branch node* and a linear node. The branch node represents the region's alternatives with the corresponding nodes as its children. A linear node with three children can then be used to capture the rest of the branch region. Its first child is the region's entry node, the second child the branch node representing the alternatives, and the third child the region's exit node. Finally, a loop region maps to a *loop node* with the region's single node as its child.

Figure 5a shows Euclid's algorithm as a CFG, and Figure 5b shows the same CFG after CFR, which restructured the head-controlled loop to a tail-controlled loop. The left of Figure 5c shows the corresponding control tree.

Demand Annotation: Structural analysis exposes necessary control flow regions for a direct translation to RVSDG. A control flow tree's branch and loop nodes map directly to γ- and θ-nodes, and individual instructions to simple nodes, but it is further necessary to expose the data dependencies of these nodes for efficient generation.

This is the task of demand annotation. It exposes these data dependencies by annotating control tree nodes with the variables that are demanded by the instructions within control flow regions. It accomplishes this using a *read-write* and *demand-set* annotation pass. The read-write pass annotates each control tree node with the set of read and written variables of the corresponding control flow region, while the demand-set pass uses these variables to annotate each control tree node with the set of demanded variables, i.e. variables that are necessary to fulfill the dependencies of the instructions within a control flow region.



(a) CFG (b) Restructured CFG (c) Annotated control tree (d) RVSDG

Fig. 5. Intra-Procedural Translation

Algorithm II: Demand Annotation

(1) ***Read-Write Annotation****: Process the control tree nodes in post-order as follows:*
- ***Basic Block****: For each instruction i processed bottom-up, the read set is $R = (R \setminus W_i) \cup R_i$. The write set is $W = \bigcup W_i$.*
- ***Linear Node****: For each child c processed right to left, the read set is $R = (R \setminus W_c) \cup R_c$. The write set is $W = \bigcup W_c$.*
- ***Branch Node****: For each child c, the read set and write set is $R = \bigcup R_c$ and $W = \bigcap W_c$, respectively.*
- ***Loop Node****: For the child c, the read set and write set is $R = R_c$ and $W = W_c$, respectively.*

(2) ***Demand-Set Annotation****: Process the control tree nodes with an empty demand set D_t as follows:*
- ***Basic Block****: Set $D = D_t = (D_t \setminus W) \cup R$ and continue processing.*
- ***Linear Node****: Recursively process the children right to left. Set $D = D_t = (D_t \setminus W) \cup R$. and continue processing.*
- ***Branch Node****: Set $D_{tmp} = D_t$. Recursively process each child with a copy of D_t. Set $D = D_t = (D_{tmp} \setminus W) \cup R$ and continue processing.*
- ***Loop Node****: Set $D = D_t \cup R$. Recursively process the child with $D_t = D$ and continue processing.*

Algorithm II shows the details of the two passes. The read-write pass annotates each node with the read set R and write set W. It processes the tree in post-order, building up the two sets from the innermost to the outermost nested control flow region. For linear nodes, the children are processed from right to left, i.e. bottom-up in the restructured CFG, to create the two sets. For branch nodes, a variable is only considered to be written, if it is in the write set of *all* the node's children, i.e. it was written in all alternatives of a conditional.

The demand-set pass uses the read set R and write set W to construct a demand set D for each node. The algorithm is initialized with an empty set D_t, which is used to keep track of demanded variables during traversal. The demand-set pass traverses the tree such that it follows a bottom-up traversal of the restructured CFG, adding and removing variables from D_t during this traversal according to each node's rules. For branch nodes, each child is processed with a copy of D_t, as the corresponding alternatives of the conditional are independent from another. For loop nodes, the θ-node's requirement that inputs and outputs must have the same signature necessitates that R is added to D_t *before* the loop's body is processed. The right of Figure 5c shows the traversal order for the two passes along with the read, write, and demand set for each node of the control tree on the left.

Control Tree Translation: After demand annotation, each node of the control tree is annotated with the set of variables that its instructions require, i.e. their data dependencies. Finally, the control tree translation constructs a δ-/λ-region from the control tree along with its annotated demand sets. Algorithm III shows the details.

Algorithm III: Control Tree Translation

Process the control tree nodes as follows:

- ***Basic Block**: Process the node's operations top-down creating simple nodes in the RVSDG.*
- ***Linear Node**: Recursively process the node's children top-down.*
- ***Branch Node**:Begin a γ-node with inputs according to the node's demand set. Create subregions by recursively processing the node's children. Finish the γ-node with outputs according to its right sibling node's demand set.*
- ***Loop Node**: Begin a θ-node with inputs according to the node's demand set. Create its region by recursively processing its child. Finish the θ-node with outputs according to its demand set.*

The algorithm processes each node in the control tree creating γ- and θ-nodes for all branch and loop nodes, respectively. It uses the demand set of the right sibling for the outputs of gamma nodes, corresponding to the branch region's join node in the CFG. Figure 5d shows the resulting RVSDG nodes for the example.

5.1.3 Modeling Stateful Computations. Algorithm I adds an additional state argument and result to every λ-node. This state is used to sequentialize all stateful computations within a function. Nodes with side-effects consume this state and produce a new state for consumption by the next node. This single state ensures that the order of operations with side-effects in the RVSDG is according to the total order specified in the original program, ensuring correct observable behavior. Specifically, the use of a single state for sequentializing stateful operations ensures that the order of these operations in the RVSDG is equivalent to the order in the restructured CFG.

The utilization of a single state is, however, overly conservative, as different computations can have mutually exclusive side-effects. For example, the side-effect of a non-terminating loop is unrelated to a non-dereferencable load. These stateful computations can be modeled independently with the help of distinct states, as depicted in Figure 1f. This results in the explicit exposure of more concurrent computations, as loops with no memory operations would become independent from other loops with memory operations. Moreover, the possibility of encoding independent states can also be leveraged by analyses and optimizations. For example, alias analysis can directly encode independent memory operations into the RVSDG by introducing additional memory states. Pure functions could be easily recognized and optimized, as they would contain no operations that use the added states and therefore would only pass it through, i.e., the origin of the state result would be the λ-region's argument.

5.2 Destruction

The destruction stage reestablishes control flow by extracting an IPG from an RVSDG and generating CFGs from individual λ-regions. Inter-Procedural Control Flow Recovery (Inter-PCFR) creates an IPG from λ-nodes, while Intra-Procedural Control Flow Recovery (Intra-PCFR) extracts control flow from γ- and θ-nodes and generates basic blocks with corresponding operations for primitive nodes. A λ-region without γ- and θ-nodes trivially transforms into a linear CFG, while λ-regions with these nodes require construction of branches and/or loops. This section discusses Inter-PCFR in detail. Detailed discussion of Intra-PCFR is found in Bahmann et al. [3].

5.2.1 Inter-Procedural Control Flow Recovery. Inter-PCFR recovers an IPG from an RVSDG. IPG nodes are created for λ-/δ-nodes as well as arguments of the ω-region, while IPG edges are inserted to capture the dependencies between these nodes. Algorithm IV starts by creating IPG nodes for all arguments of the ω-region, i.e., all external functions. It continues by recursively traversing the region tree, creating IPG nodes for encountered λ-/δ-nodes and IPG edges for their dependencies. For the region of every λ-/δ-node, it invokes Intra-PCFR to create a CFG.

5.2.2 Intra-Procedural Control Flow Recovery. Bahmann et al. [3] explored two different approaches for CFG generation: *Structured Control Flow Recovery (SCFR)* and *Predicative Control Flow Recovery (PCFR)*. SCFR uses the region hierarchy within a λ-region to recover control flow, while PCFR generates branches for predicate producers and follows the predicate consumers to the eventual destination. Both schemes reestablish evaluation-equivalent CFGs, but differ in the recoverable control flow. SCFR recovers only control flow that resembles the structural

Algorithm IV: Inter-Procedural Control Flow Recovery

(1) Create IPG nodes for all arguments of the ω-region.

(2) Process all nodes of the ω-region in topological order as follows:

- *λ-**nodes**: Create an IPG node, and mark it exported if the λ-node's output has a ω-region's result as user. For every context variable $cv = (i, a)$, add an edge from the λ-node's IPG node to the corresponding IPG node of the producer of i. Create a CFG from the λ-node's subregion and attach it to the IPG node.*
- *δ-**nodes**: Create an IPG node, and mark it exported if the δ-node's output has a ω-region's result as user. For every context variable $cv = (i, a)$, add an edge from the δ-node's IPG node to the corresponding IPG node of the producer of i. Create the expression from the δ-node's subregion and attach it to the IPG node.*
- *ϕ-**nodes**: For every argument of the ϕ-region, create an IPG node for the corresponding δ-/λ-node and add IPG edges from this node to the corresponding IPG nodes of the context variables. Translate the δ-/λ-nodes in the ϕ-region according to the rules above. Mark the IPG node as exported if the corresponding ϕ-node's output has a ω-region's result as user.*

nodes in λ-regions, i.e., control flow equivalent to `if-then-else`, `switch`, and `do-while` statements, while PCFR can recover arbitrary complex control flow, i.e., control flow that is not restricted to RVSDG constructs. PCFR reduces the number of static branches in the resulting control flow [3], but might also result in undesirable control flow for certain architectures, such as graphic processing units [36]. For the sake of brevity, we omit a discussion of SCFR and PCFR as the algorithms are extensively described by Bahmann et al. [3].

6 OPTIMIZATIONS

The properties of the RVSDG make it an appealing IR for optimizing compilers. Many optimizations can be expressed as simple graph traversals, where subgraphs are rewritten, nodes are moved between regions, nodes or edges are marked, or edges are diverted. In this section, we present Common and Dead Node Elimination optimizations that exploit the RVSDG's properties to unify traditionally distinct transformations.

6.1 Common Node Elimination

Common Node Elimination (CNE) permits the removal of redundant computations by detecting congruent nodes. These nodes always produce the same results, enabling the redirection of their result edges to a single node. This renders the other nodes dead, permitting Dead Node Elimination to remove them. CNE is similar to common subexpression elimination and value numbering [2] in that it detects equivalent computations, but as the RVSDG represents all computations uniformly as nodes, it can be extended to conditionals [38], loops, and functions.

We consider two simple nodes n_1 and n_2 congruent, or $n_1 \cong n_2$, if they represent the same computation, have the same number of inputs, i.e., $\left|I_{n_1}\right| = \left|I_{n_2}\right|$, and the inputs $i^k_{n_1}$ and $i^k_{n_2}$ are congruent, or $i^k_{n_1} \cong i^k_{n_2}$, for all $k = [0..\left|I_{n_1}\right|]$. Two inputs are congruent if their respective origins $g^k_{n_1}$ and $g^k_{n_2}$ are congruent, i.e., $g^k_{n_1} \cong g^k_{n_2}$. By definition, the origins of inputs are either outputs of simple or structural nodes, or arguments of regions. Origins from simple nodes are only equivalent when their respective producers are computationally equivalent, whereas for the other cases, it must be guaranteed that they always receive the same value.

The implementation of CNE consists of two phases: mark and divert. The mark phase identifies congruent simple nodes, while the divert phase diverts all edges of their origins to a single node, rendering all other nodes dead. Both phases of Algorithm V perform a simple top-down traversal, recursively processing subregions of structural nodes annotating inputs, outputs, arguments, and results, as well as simple nodes as congruent. For γ-nodes, the algorithm marks only computations within a *single* region as congruent and performs no analysis between regions. In the case of θ-nodes, computations are only congruent when they are congruent before and after the loop execution, i.e., the inputs and results of two loop variables must be congruent. Figure 6b shows the RVSDG for the code in Figure 6a, and Figure 6c the RVSDG after CNE. Two of the four multiplications take the same inputs and therefore are congruent to each other, resulting in the redirection of their result edges.

Algorithm V: Common Node Elimination

(1) ***Mark****: Process all nodes in topological order as follows:*

- ***Simple nodes****: Denote this node as* n*. Mark* n *as congruent to all nodes* n' *which represent the same operation and where* $|I_n| = |I_{n'}| \wedge i_n^k \cong i_{n'}^k$ *for all* $k = [0..\,|I_n|]$*. Mark all outputs* $o_n^k \cong o_{n'}^k$ *for all* $k = [0..\,|O_n|]$*.*
- γ***-node****: For all entry variables* $ev_1, ev_2 \in EV$ *where* $i_{ev_1} \cong i_{ev_2}$*, mark* $a_{ev_1}^k \cong a_{ev_2}^k$ *for all* $k \in [0..\,|A_{ev_1}|]$*. Recursively process the* γ*-regions. For all exit variables* $ex_1, ex_2 \in EX$ *where* $r_{ex_1}^k \cong r_{ex_2}^k$ *for all* $k \in [0..\,|R_{ex_1}|]$*, mark* $o_{ex_1} \cong o_{ex_2}$*.*
- θ***-node****: For all loop variables* $lv_1, lv_2 \in LV$ *where* $i_{lv_1} \cong i_{lv_2} \wedge r_{lv_1} \cong r_{lv_2}$*, mark* $a_{lv_1} \cong a_{lv_2}$ *and* $o_{lv_1} \cong o_{lv_2}$*. Recursively process the* θ*-region.*
- λ***-node****: For all context variables* $cv_1, cv_2 \in CV$ *where* $i_{cv_1} \cong i_{cv_2}$*, mark* $a_{cv_1} \cong a_{cv_2}$*. Recursively process the* λ*-region.*
- ϕ***-node****: For all context variables* $cv_1, cv_2 \in CV$ *where* $i_{cv_1} \cong i_{cv_2}$*, mark* $a_{cv_1} \cong a_{cv_2}$*. Recursively process the* ϕ*-region.*
- ω***-node****: Recursively process the* ω*-region.*

(2) ***Divert****: Process all nodes in topological order as follows:*

- ***Simple nodes****: Denote this node as* n*. For all nodes* n' *which are congruent to* n*, divert all outputs* $o_{n'}^k$ *to* o_n^k *for all* $k = [0..\,|O_n|]$*.*
- γ***-node****: For all entry variables* $ev_1, ev_2 \in EV$ *where* $i_{ev_1} \cong i_{ev_2}$*, divert all edges from* $a_{ev_2}^k$ *to* $a_{ev_1}^k$ *for all* $k \in [0..\,|A_{ev_1}|]$*. Recursively process the* γ*-regions. For all exit variables* $ex_1, ex_2 \in EX$ *where* $r_{ex_1}^k \cong r_{ex_2}^k$ *for all* $k \in [0..\,|R_{ex_1}|]$*, divert all edges from* o_{ex_2} *to* o_{ex_1}*.*
- θ***-node****: For all induction variables* $lv_1, lv_2 \in LV$ *where* $a_{lv_1} \cong a_{lv_2} \wedge o_{lv_1} \cong o_{lv_2}$*, divert all edges from* a_{lv_2} *to* a_{lv_1} *and from* o_{lv_2} *to* o_{lv_1}*. Recursively process the* θ*-region.*
- λ***-node****: For all context variables* $cv_1, cv_2 \in CV$ *where* $i_{cv_1} \cong i_{cv_2}$*, divert all edges from* a_{cv_2} *to* a_{cv_1}*. Recursively process the* λ*-region.*
- ϕ***-node****: For all context variables* $cv_1, cv_2 \in CV$ *where* $i_{cv_1} \cong i_{cv_2}$*, divert all edges from* a_{cv_2} *to* a_{cv_1}*. Recursively process the* ϕ*-region.*
- ω***-node****: Recursively process the* ω*-region.*

For simple nodes, the algorithm marks all nodes within a region that are congruent to a node n. In order to avoid costly traversals of all nodes for every node n, the mark phase takes the candidates from the users of the origin of n's first input. If there is another input from a simple node n' with the same operation and number of inputs among them, the other inputs from both nodes can be compared for congruence. Moreover, a region must store constant nodes, i.e. nodes without inputs, separately from other nodes so that the candidate nodes for constants are available. For commutative simple nodes, the inputs should be sorted before their comparison.

The presented algorithm only detects congruent simple nodes within a region. For γ-nodes, congruence can also exist between nodes of different γ-regions, and extending the algorithm would eliminate these redundancies. Another extension would be to detect congruent structural nodes, to implement conditional fusion [38] and loop fusion [28]. In the case of γ-nodes, it is sufficient to ensure that two nodes have congruent predicates, while θ-nodes require congruence detection between different θ-regions to ensure that their predicates are the same.

6.2 Dead Node Elimination

Dead Node Elimination (DNE) is a combination of dead and unreachable code elimination, and removes all nodes that do not contribute to the result of a computation. Dead nodes are generated by unreachable and dead code from the input program, as well as by other optimizations such as Common Node Elimination. An operation is considered dead code when its results are either not used or only by other dead operations. Thus, an output of a node is dead, if it has no users or all its users are dead. We consider a node to be dead, if all its outputs are dead. It follows that a node's inputs are dead, if the node itself is dead. We call all entities that are not dead alive.

The implementation of DNE consists of two phases: mark and sweep. The mark phase identifies all outputs and arguments that are alive, while the sweep phase removes all dead entities. The mark phase traverses RVSDG edges according to the rules in Algorithm VI. If a structural node is dead, the mark phase skips the traversal of its subregions as well as all of the contained computations, as it never reaches the node in the first place. The mark phase is invoked for all result origins of the ω-region.

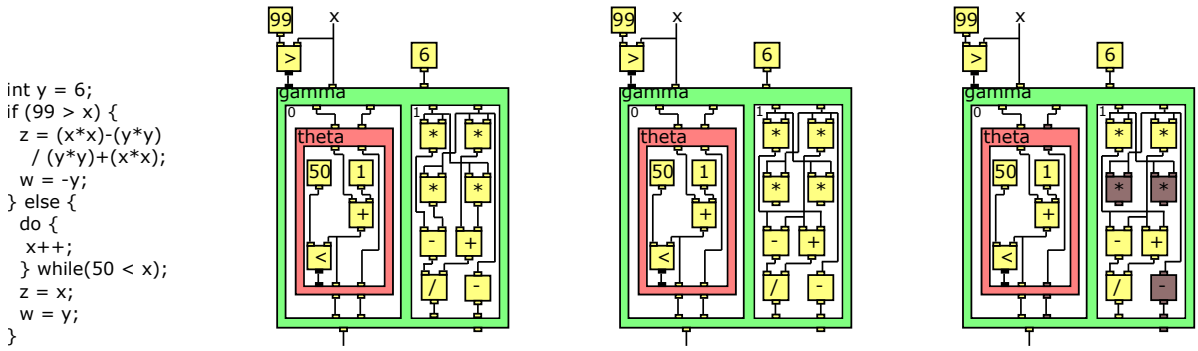


Fig. 6. Dead and Common Node Elimination

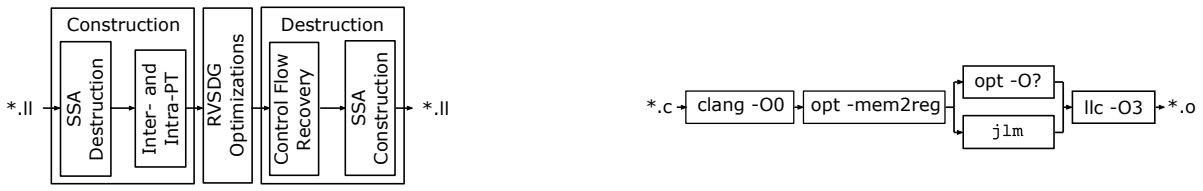
Algorithm VI: Dead Node Elimination

(1) ***Mark****: Mark output or argument as alive and continue as follows:*
- *ω-**region argument**: Stop marking.*
- *ϕ-**node output**: Mark the result origin of the corresponding recursion variable.*
- *ϕ-**region argument**: Mark the input origin if the argument belongs to a context variable. Otherwise, mark the output of the corresponding recursion variable.*
- *λ-**node output**: Mark all result origins of the λ-region.*
- *λ-**region argument**: Mark the input origin if the argument is a dependency.*
- *θ-**node output**: Mark the θ-node's predicate origin as well as the result and input origin of the corresponding loop variable.*
- *θ-**region argument**: Mark the input origin and output of the corresponding loop variable.*
- *γ-**node output**: Mark the γ-node's predicate origin as well as the origins of all results of the corresponding exit variable.*
- *γ-**region argument**: Mark the input origin of the corresponding entry variable.*
- ***Simple node output**: Mark the origin of all inputs.*

(2) ***Sweep****: Process all nodes in reverse topological order and remove them if they are dead. Otherwise, process them as follows:*
- *ω-**node**: Recursively process the ω-region. Remove all dead arguments.*
- *γ-**node**: For all exit variables $(R, o) \in EX$ where o is dead, remove o and all $r \in R$. Recursively process the γ-regions. For all entry variables $(i, A) \in EV$ where all $a \in A$ are dead, remove all $a \in A$ and i.*
- *θ-**node**: For all loop variables $(i, a, r, o) \in LV$ where a and o are dead, remove o and r. Recursively process the θ-region. Remove i and a.*
- *λ-**node**: Recursively process the λ-region. For all context variables $(i, a) \in CV$ where a is dead, remove a and i.*
- *ϕ-**node**: For all recursion variables $(r, a, o) \in RV$ where a and o are dead, remove o and r. Recursively process the ϕ-region. Remove a. For all context variables $(i, a) \in CV$ where a is dead, remove a and i.*

The sweep phase performs a simple bottom-up traversal of an RVSDG, recursively processing subregions of structural nodes as long as these nodes are alive. A dead structural node is removed with all its contained computations. The RVSDG's uniform representation of all computations as nodes permits DNE to not only remove simple computations, but also compound computations such as conditionals, loops, or even entire functions. Moreover, its nested structure avoids the processing of entire branches of the region tree if they are dead.

Figure 6d shows the RVSDG from Figure 6c after the mark phase. Grey colored entities are dead. The mark phase traverses the graph's edges, marking the γ-node's leftmost output alive. This renders the corresponding result origins of the γ-regions alive, then the leftmost output of the θ-node, and so forth. After the mark phase annotated all outputs and arguments as alive, the sweep phase removes all dead entities.



(a) Jlm compilation pipeline.

(b) Evaluation setup.

Fig. 7. Jlm's compilation pipeline and evaluation setup.

7 IMPLEMENTATION AND EVALUATION

This section aims to demonstrate that the RVSDG has no inherent impediment that prevents it from producing competitive code, and that it can serve as the IR in a compiler's optimization stage. The goal is not to outperform mature compilers like LLVM or GCC, as it would require engineering effort far beyond the scope of this article. We evaluate the RVSDG by generated code performance and size, compilation time, and representational overhead.

7.1 Implementation

We have implemented *jlm*, a publicly available prototype compiler [35] that uses the RVSDG for optimizations. Its compilation pipeline is outlined in Figure 7a. Jlm takes LLVM IR as input, constructs an RVSDG, transforms and optimizes this RVSDG, and destructs it again to LLVM IR. The SSA form of the input is destructed before RVSDG construction proceeds with Inter- and Intra-PT. This additional step is required due to the control flow restructuring phase of Intra-PT. Destruction discovers control flow by employing SCFR before it constructs SSA form to output LLVM IR. Jlm supports LLVM IR function, integer, floating point, pointer, array, structure, and vector types as well as their corresponding operations. Exceptions and intrinsic functions are currently unsupported. The compiler uses two distinct states to model side-effects: one for memory accesses and one for non-terminating loops. We implemented the following optimizations in addition to CNE and DNE:

- *Inlining* (ILN): Simple function inlining.
- *Invariant Value Redirection* (INV): Redirects invariant values from θ- and γ-nodes.
- *Node Push-out* (PSH): Moves all invariant nodes out of γ- and θ-regions.
- *Node Pull-in* (PLL): Moves all nodes that are only used in one γ-region into the γ-node. This ensures their conditional execution, while avoiding code bloat.
- *Node Reduction* (RED): Performs simplifications, such as constant folding or strength reduction, similarly to LLVM's redundant instruction combinator (`-instcombine`), albeit by far not as many.
- *Loop Unrolling* (URL): Unrolls all inner loops by a factor of four. Higher factors gave no significant performance improvements in return for the increased code size.
- $\theta - \gamma$ *Inversion* (IVT): Inverts γ- and θ-nodes where both nodes have the same predicate origin. This replaces the loop containing a conditional with a conditional that has a loop in its then-case.

We use optimization order ILN,INV,RED,DNE,IVT,INV,DNE,PSH,INV,DNE,URL,INV,RED,CNE,DNE,PLL,INV,DNE.

7.2 Evaluation Setup

Figure 7b outlines our evaluation setup. We use clang 7.0.1 [9] to convert C files to LLVM IR, pre-optimize the IR with LLVM's `opt`, and then optimize it either with `jlm`, or `opt` using different optimization levels. The optimized output is converted to an object file with LLVM's `llc`. The pre-optimization step is necessary to avoid a re-implementation of LLVM's `mem2reg` pass, since clang allocates all values on the stack by default due to

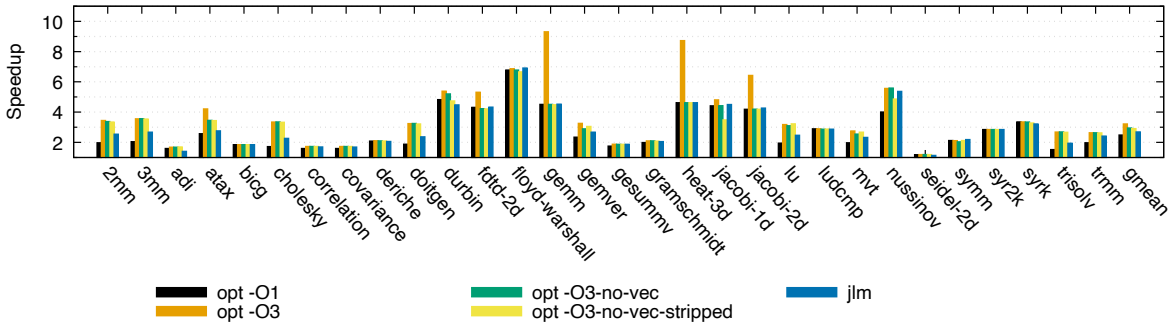


Fig. 8. Speedup relative to `O0` at different optimization levels.

LLVM IR only supporting CFGs in SSA form. We use the polybench 4.2.1 beta benchmark suite [33] to evaluate the RVSDG's usability and efficacy. This benchmark suite provides structurally small benchmarks, and therefore reduces the implementation effort for RVSDG construction and destruction, as well as the number and complexity of optimizations. The experiments are performed on an Intel Xeon E5-2695v4 running CentOS 7.4. The core frequency is pinned to 2.0 GHz to avoid performance variations and thermal throttling effects. All outputs of the benchmark runs are verified to equal the corresponding outputs of the executables produced by clang.

7.3 Performance

Figure 8 shows the speedup at five different optimization levels. The `O0` optimization level serves as baseline. The `O3-no-vec` optimization level is the same as `O3`, but without slp- and loop-vectorization. Optimization level `O3-no-vec-stripped` is the same as `O3-no-vec`, but the IR is stripped of named metadata and attribute groups before invoking `llc`. Since jlm does not support metadata and attributes yet, this optimization level permits us to compare the pure optimized IR against jlm without the optimizer providing hints to `llc`. We omit optimization level `O2` as it was very similar to `O3`. The gmean column in Figure 8 shows the geometric mean of all benchmarks.

The results show that the executables produced by jlm (gmean 2.70) are faster than `O1` (gmean 2.49), but slower than `O3` (gmean 3.22), `O3-no-vec` (gmean 2.95), and `O3-no-vec-stripped` (gmean 2.91). Optimization level `O3` attempts to vectorize twenty benchmarks, but only produces measurable results for eight of them: atax, durbin, fdtd-2d, gemm, gemver, heat-3d, jacobi-1d, and jacobi-2d. Jlm would require a vectorizer to achieve such speedups.

Disabling vectorization with `O3-no-vec` and `O3-no-vec-stripped` shows that jlm achieves similar speedups for fdtd-2d, gemm, heat-3d, javobi-1d, and jacobi-2d. The metadata transferred between the optimizer and `llc` only makes a significant difference for durbin, floyd-warshall, gesummv, jacobi-1d, and nussinov. In the case of gesummv and jacobi-1d, performance drops below jlm. Jlm is outperformed by optimization level `O1` at four benchmarks: adi, durbin, seidel-2d, and syrk. We inspected the output files and found the following causes:

- *adi*: Jlm fails to eliminate load instructions from the two innermost loops. These loads have loop-carried dependencies with a distance of one to store instructions in the same loop, and can be eliminated by propagating the stored value to the users of the load's output. The corresponding LLVM pass is loop load elimination (`-loop-load-elim`). Jlm performance equals `O1` if this transformation is performed by hand.
- *durbin*: Jlm fails to transform a loop that copies values between arrays to a `memcpy` intrinsic. This impedes LLVM's code generator to produce better code. The LLVM pass responsible for this transformation is the loop-idiom pass (`-loop-idiom`). If the loop is replaced with a call to `memcpy`, then jlm is better than `O1`.

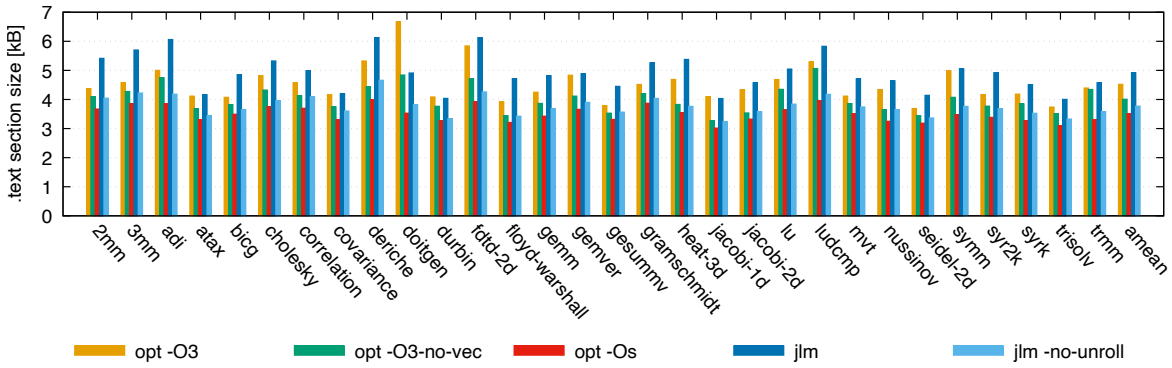


Fig. 9. Code size at different optimization levels.

- *seidel-2d*: Similarly to adi, jlm fails to eliminate load instructions from the innermost loop. If the load elimination is performed by hand, then jlm achieves the same performance as `O1`.
- *syrk*: Jlm fails to satisfactorily apply CNE due to an overly strict sequentialization of load and store instructions. Loads from the same address are not detected as congruent due to different state edge origins. An alias analysis pass would resolve this problem.

Figure 8 shows that it is feasible to produce competitive code using the RVSDG, but also that more optimizations and analyses are required in order to reliably do so. Performance differences are not caused by inherent RVSDG characteristics, but can be attributed to missing analyses, optimizations, and heuristics for their application. The above results and Table 1 indicate that an alias analysis pass is particluarly required.

7.4 Code Size

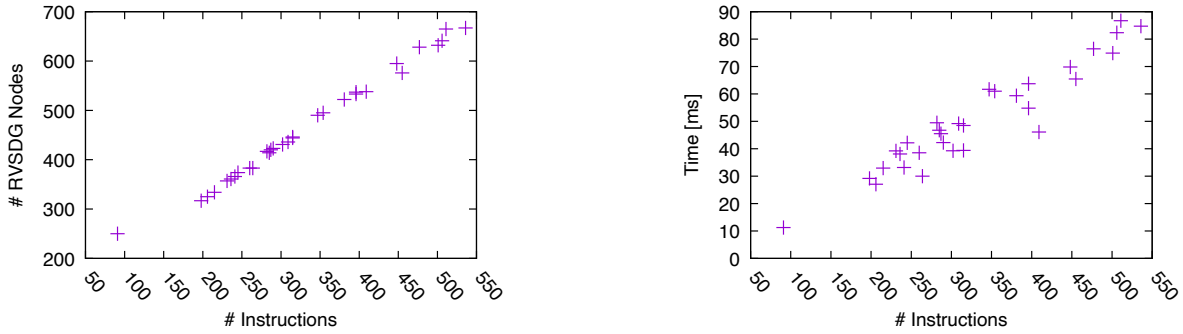
Figure 9 shows the code size for `O3`, `O3-no-vec`, `Os`, and for jlm with and without loop unrolling. The amean column shows the arithmetic mean of all benchmarks. Optimization level `O3` produces on average text sections that are 11% bigger than `O3-no-vec`. Vectorization often requires loop transformations to make loops amenable to the vectorizer, and the insertion of pre- and post-loop code. This affects code size negatively, but can result in better performance. The results also show that `Os` produces smaller text sections than `O3-no-vec`. This is due to more conservative optimization heuristics and the omission of optimizations, e.g., aggressive instruction combination (`-aggressive-instcombine`) or the promotion of by-reference arguments to scalars (`-argpromotion`).

Jlm produces ca. 39% bigger text sections compared to `Os`. The results without loop unrolling show that this can be attributed to the naive heuristic used. Jlm does not take code size into account and unrolls every inner loop unconditionally four times, leading to excessive code expansion. Avoiding unrolling completely results in text sections that are on average between `O3-no-vec` and `Os`. This indicates that the excessive code size is due to naive heuristics and shortcomings in the implementation, but not to inherent characteristics of the RVSDG.

7.5 Compilation Overhead

Figure 10 shows overhead in terms of IR size and time, with Figure 10a relating LLVM instruction count to number of RVSDG nodes, and Figure 10b relating it to time spent on RVSDG translation and optimizations.

Figure 10a shows a clear linear relationship for all cases, confirming the observations by Bahmann et al. [3] that the RVSDG is feasible in terms of space requirements. Figure 10b also indicates a linear dependency, but with larger variations for similar input sizes. We attribute this variation to the fact that construction, optimizations,



(a) Representational overhead.

(b) Compilation times.

Fig. 10. Compilation overhead of jlm.

and destruction are also compounded by input structure. Structural differences in the inter-procedure and control flow graphs lead to runtime variations in RVSDG construction and destruction, as well as different runtimes for optimizations. For example, the presence of loops in a translation unit determines whether loop unrolling is performed, while their absence incurs no runtime overhead. Figure 10 shows that the RVSDG is feasible as an IR for optimizing compilers in terms of compilation overhead.

7.6 Comparison to LLVM

LLVM 7.0.1 invokes 85 different analyses and optimization passes at optimization level `O3`. Many of these passes are repeatedly invoked, resulting in a total of 266 invocations. Section 2.1 already highlighted that 57, or 21%, of these invocations are from six helper passes that only establish invariants and detect structures necessary for optimizations. Table 1 in Section 2.1 also shows that LLVM necessitates SSA restoration in fourteen optimization passes. Because LLVM's CFG representation does not maintain necessary invariants and structures, it requires (re-)computation of their information, which leads to a high number of helper pass invocations. This can be observed in LLVM's optimization pipeline for loop optimizations: ... *-loops -loop-simplify -lcssa-verification -lcssa* ... -loop-rotate -licm -loop-unswitch *-loop-simplify -lcsse-verification -lcssa* .... -loop-idiom -loop-deletion -loop-unroll ... *-loops -loop-simplify -lcssa-verification -lcssa* ... -loop-rotate -loop-accesses ... . Depending on an optimization's position in the pipeline, several helper passes must be executed before an optimization can be invoked. This is necessary to ensure that the required information for an optimization is present, and up to date after invocation of other optimizations, e.g., jump threading (`-jump-threading`) or CFG simplification (`-simplifycfg`). Thus, each added loop optimization can necessitate several more helper passes in the optimization pipeline. A similar pattern is seen with (basic) alias analysis (`-basicaa` and `-aa`), which are invoked 37 times in total.

In contrast, the RVSDG establishes the necessary invariants and structures during construction, and maintains them throughout compilation. The result is that jlm requires none of the aforementioned helper passes and SSA restoration, e.g., loop unrolling (URL) can be readily performed without the need to detect loops, their entry, and exits. The cost is a more elaborate construction, which requires the detection of the necessary information, and destruction, which requires the recovery of control flow. However, RVSDG construction and destruction only need to be performed once and, as demonstrated in Section 10 and in Bahman et al. [3], are practically feasible.

8 CONCLUSION

This paper presents a complete specification of the RVSDG IR for an optimizing compiler. We provide construction and destruction algorithms, and show the RVSDG's efficacy as an IR for analyses and optimizations by presenting Dead Node and Common Node Elimination. We implemented jlm, a publicly available compiler [35] that uses the RVSDG for optimizations, and evaluate it in terms of performance, code size, compilation time, and representational overhead. The results suggest that the RVSDG combines the abstractions of data centric IRs with the CFG's advantages to optimize and generate efficient control flow. This makes the RVSDG an appealing IR for optimizing compilers. A natural direction for future work is to explore how features such as exceptions can be efficiently mapped to the RVSDG. Another research direction would be to extend the number of optimizations and their heuristics in jlm to a competitive level with CFG-based compilers. This would provide further information about the number of necessary optimizations, their complexity, and consequently the required engineering effort.

ACKNOWLEDGMENTS

This work was funded by Vetenskapsrådet under grant agreement 2015-05159. The computations were performed on resources provided by the NTNU EPIC compute cluster [41].

REFERENCES

[1] Frances E Allen. 1970. Control flow analysis. In *ACM Sigplan Notices*, Vol. 5. ACM, 1–19.
[2] B. Alpern, M. N. Wegman, and F. K. Zadeck. 1988. Detecting Equality of Variables in Programs. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM, 1–11.
[3] Helge Bahmann, Nico Reissmann, Magnus Jahre, and Jan Christian Meyer. 2015. Perfect Reconstructability of Control Flow from Demand Dependence Graphs. *ACM Transactions on Architecture and Code Optimization* 11, 4 (2015), 66:1–66:25.
[4] W. Baxter and H. R. Bauer, III. 1989. The Program Dependence Graph and Vectorization. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM, 1–11.
[5] Philip L Campbell, Ksheerabdhi Krishna, and Robert A Ballance. 1993. *Refining and Defining the Program Dependence Web*. Technical Report. University of New Mexico.
[6] Larry Carter, Jeanne Ferrante, and Clark Thomborson. 2003. Folklore Confirmed: Reducible Flow Graphs are Exponentially Larger. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*, Vol. 38. ACM, 106–114.
[7] Jeronimo Castrillon. 2013. *Programming Heterogeneous MPSoCs: Tool Flows to Close the Software Productivity Gap*. Ph.D. Dissertation. RWTH Aachen Univeristy.
[8] Jong-Deok Choi, Vivek Sarkar, and Edith Schonberg. 1996. Incremental computation of static single assignment form. In *Proceedings of the International Conference on Compiler Construction*. Springer-Verlag, 223–237.
[9] Clang. 2017. Clang: A C Language Family Frontend for LLVM. https://clang.llvm.org. Accessed: 2019-10-30.
[10] GNU Compiler Collection. 2018. https://gcc.gnu.org/. Accesssed: 2019-08-05.
[11] Daniel Cordes, Peter Marwedel, and Arindam Mallik. 2010. Automatic parallelization of embedded software using hierarchical task graphs and integer linear programming. In *Proceedings of the ACM/IEEE International Conference on Hardware/Software Codesign and System Synthesis*. 267–276.
[12] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. K Zadeck. 1991. *Efficiently Computing Static Single Assignment Form and the Control Dependence Graph*. Technical Report.
[13] Jack Bonnell Dennis. 1980. Data flow supercomputers. *Computer* 13, 11 (1980), 48–56.
[14] Shuhan Ding, John Earnest, and Soner Önder. 2014. Single Assignment Compiler, Single Assignment Architecture: Future Gated Single Assignment Form. In *Proceedings of the International Symposium on Code Generation and Optimization*. ACM.
[15] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems* 9, 3 (1987), 319–349.
[16] Paul Havlak. 1993. Construction of Thinned Gated Single-Assignment Form. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing-Revised Papers*. Springer, 477–499.
[17] S. Horwitz, J. Prins, and T. Reps. 1988. On the Adequacy of Program Dependence Graphs for Representing Programs. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages*. ACM, 146–157.
[18] S. Horwitz, T. Reps, and D. Binkley. 1988. Interprocedural Slicing Using Dependence Graphs. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* 23, 7, 35–46.

[19] Neil Johnson and Alan Mycroft. 2003. Combined Code Motion and Register Allocation Using the Value State Dependence Graph. In *Proceedings of the International Conference on Compiler Construction*. Springer-Verlag, 1–16.
[20] Neil E. Johnson. 2004. *Code size optimization for embedded processors.* Technical Report. University of Cambridge, Computer Laboratory.
[21] Richard Johnson, David Pearson, and Keshav Pingali. 1994. The Program Structure Tree: Computing Control Regions in Linear Time. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 171–185.
[22] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization.*
[23] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A Compiler Infrastructure for the End of Moore's Law. arXiv:2002.11054 [cs.PL]
[24] Alan C. Lawrence. 2007. *Optimizing compilation with the Value State Dependence Graph.* Technical Report. University of Cambridge.
[25] LLVM. 2018. https://bugs.llvm.org/show_bug.cgi?id=31851. Accesssed: 2020-04-05.
[26] LLVM. 2018. https://bugs.llvm.org/show_bug.cgi?id=37202. Accesssed: 2020-04-05.
[27] LLVM. 2018. https://bugs.llvm.org/show_bug.cgi?id=31183. Accesssed: 2020-04-05.
[28] Naraig Manjikian and Tarek S Abdelrahman. 1997. Fusion of loops for parallelism and locality. *IEEE Transactions on Parallel and Distributed Systems* 8, 2 (1997), 193–209.
[29] Steven S. Muchnick. 1997. *Advanced Compiler Design and Implementation.* Morgan Kaufmann.
[30] R. Namballa, N. Ranganathan, and A. Ejnioui. 2004. Control and Data Flow Graph Extraction for High-Level Synthesis. In *IEEE Computer Society Annual Symposium on VLSI.* 187–192.
[31] Karl J. Ottenstein, Robert A. Ballance, and Arthur B. MacCabe. 1990. The Program Dependence Web: A Representation Supporting Control-, Data-, and Demand-driven Interpretation of Imperative Languages. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation.* ACM, 257–271.
[32] Guilherme Ottoni, Ram Rangan, Adam Stoler, and David I. August. 2005. Automatic Thread Extraction with Decoupled Software Pipelining. In *Proceedings of the ACM/IEEE International Symposium on Microarchitecture.* IEEE, 105–118.
[33] Louis-Noël Pouchet. 2017. Polybench/C 4.2. http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/. Accessed: 2019-11-11.
[34] Nico Reissmann. 2012. Utilizing the Value State Dependence Graph for Haskell. (2012).
[35] Nico Reissmann. 2017. jlm. https://github.com/phate/jlm. Accessed: 2020-04-05.
[36] Nico Reissmann, Thomas L. Falch, Benjamin A. Bjørnseth, Helge Bahmann, Jan Christian Meyer, and Magnus Jahre. 2016. Efficient control flow restructuring for GPUs. In *Proceedings of the International Conference on High Performance Computing and Simulation.* 48–57.
[37] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. 1988. Global value numbers and redundant computations. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages.* ACM, 12–27.
[38] Radu Rugina and Martin C. Rinard. 2001. Recursion Unrolling for Divide and Conquer Programs. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing-Revised Papers.* Springer-Verlag, 34–48.
[39] V. Sarkar. 1991. Automatic Partitioning of a Program Dependence Graph into Parallel Tasks. *IBM Journal of Research and Development* 35, 5-6 (1991), 779–804.
[40] M. Sharir. 1980. Structural analysis: A new approach to flow analysis in optimizing compilers. *Computer Languages* 5, 3 (1980), 141–153.
[41] Magnus Själander, Magnus Jahre, Gunnar Tufte, and Nico Reissmann. 2019. EPIC: An Energy-Efficient, High-Performance GPGPU Computing Research Infrastructure. arXiv:1912.05848 [cs.DC]
[42] James Stanier. 2012. *Removing and Restoring Control Flow with the Value State Dependence Graph.* Ph.D. Dissertation. University of Sussex.
[43] James Stanier and Alan Lawrence. 2011. The Value State Dependence Graph Revisited. In *Proceedings of the Workshop on Intermediate Representations.* 53–60.
[44] James Stanier and Des Watson. 2013. Intermediate Representations in Imperative Compilers: A Survey. *Comput. Surveys* 45, 3 (2013).
[45] R. Tarjan. 1972. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.* 1, 2 (1972), 146–160.
[46] Peng Tu and David Padua. 1995. Efficient Building and Placing of Gating Functions. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation.* ACM, 47–55.
[47] Mark N. Wegman and F. Kenneth Zadeck. 1991. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems* 13, 2 (1991), 181–210.
[48] Daniel Weise, Roger F. Crew, Michael Ernst, and Bjarne Steensgaard. 1994. Value Dependence Graphs: Representation Without Taxation. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages.* ACM, 297–310.
[49] Ali Mustafa Zaidi. 2015. *Accelerating control-flow intensive code in spatial hardware.* Technical Report. University of Cambridge.
[50] Ali Mustafa Zaidi and David Greaves. 2015. Value State Flow Graph: A Dataflow Compiler IR for Accelerating Control-Intensive Code in Spatial Hardware. *ACM Transactions on Reconfigurable Technology and Systems* 9, 2 (2015), 14:1–14:22.
[51] F. Zhang and E.H. D'Hollander. 2004. Using hammock graphs to structure programs. *IEEE Transactions on Software Engineering* 30, 4 (2004), 231–245.