Perfect Reconstructability of Control Flow from Demand Dependence Graphs

HELGE BAHMANN, Google Zürich
NICO REISSMANN, MAGNUS JAHRE, and JAN CHRISTIAN MEYER,
Norwegian University of Science and Technology

Demand-based dependence graphs (DDGs), such as the (Regionalized) Value State Dependence Graph ((R)VSDG), are intermediate representations (IRs) well suited for a wide range of program transformations. They explicitly model the flow of data and state, and only implicitly represent a restricted form of control flow. These features make DDGs especially suitable for automatic parallelization and vectorization, but cannot be leveraged by practical compilers without efficient construction and destruction algorithms. Construction algorithms remodel the arbitrarily complex control flow of a procedure to make it amenable to DDG representation, whereas destruction algorithms reestablish control flow for generating efficient object code. Existing literature presents solutions to both problems, but these impose structural constraints on the generatable control flow, and omit qualitative evaluation.

The key contribution of this article is to show that there is no intrinsic structural limitation in the control flow directly extractable from RVSDGs. This fundamental result originates from an interpretation of loop repetition and decision predicates as computed continuations, leading to the introduction of the *predicate continuation* normal form. We provide an algorithm for constructing RVSDGs in predicate continuation form, and propose a novel destruction algorithm for RVSDGs in this form. Our destruction algorithm can generate arbitrarily complex control flow; we show this by proving that the original CFG an RVSDG was derived from can, apart from overspecific detail, be reconstructed perfectly. Additionally, we prove termination and correctness of these algorithms. Furthermore, we empirically evaluate the performance, the representational overhead at compile time, and the reduction in branch instructions compared to existing solutions. In contrast to previous work, our algorithms impose no additional overhead on the control flow of the produced object code. To our knowledge, this is the first scheme that allows the original control flow of a procedure to be recovered from a DDG representation.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Compilers*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*Functional constructs*; F.3.3 [**Logics and Meanings of Programs**]: Studies of Program Constructs—*Control primitives*

General Terms: Algorithms, Theory

Additional Key Words and Phrases: Intermediate representations, control flow, demand-dependence, value state dependence graph

ACM Reference Format:
Helge Bahmann, Nico Reissmann, Magnus Jahre, and Jan Christian Meyer. 2014. Perfect reconstructability of control flow from demand dependence graphs. ACM Trans. Architec. Code Optim. 11, 4, Article 66 (December 2014), 25 pages.
DOI: http://dx.doi.org/2693261

New Paper, Not an Extension of a Conference Paper.
Authors' addresses: H. Bahmann, Google Zürich, Brandschenkestrasse 110, 8002 Zürich, Switzerland; email: hcb@chaoticmind.net; N. Reissmann and M. Jahre, Department of Computer and Information Science (IDI), NTNU, Sem Sælands vei 9, 7491 Trondheim, Norway; emails: {nico.reissmann, magnus.jahre}@idi.ntnu.no; J. C. Meyer, HPC Section, IT Department, NTNU, Høgskoleringen 7i, 7491 Trondheim, Norway; email: jan.christian.meyer@ntnu.no.

2014 ACM 1544-3566/2014/12-ART66 $15.00
DOI: http://dx.doi.org/2693261

1. INTRODUCTION

The main intermediate representation (IR) for imperative languages in modern compilers is the Control Flow Graph (CFG) [Stanier and Watson 2013]. It explicitly encodes *arbitrarily complex* control flow, admitting unconstrained control flow optimizations and efficient object code generation. Translation to static single assignment (SSA) form [Cytron et al. 1989] additionally enables efficient dataflow optimizations [Wegman and Zadeck 1991; Rosen et al. 1988], but the implicit dataflow and specific execution order of CFGs restrict their utility by complicating these optimizations unnecessarily [Lawrence 2007].

Dataflow-centric IRs have developed from the insight that many classical optimizations are based on the flow of data, rather than control. DDGs such as the (Regionalized) Value State Dependence Graph ((R)VSDG) [Johnson 2004] show promising code quality improvements and reduced implementation complexity. These IRs *implicitly* represent control flow in a structured form, which allows for simpler and more powerful implementations of dataflow optimizations such as common subexpression and dead code elimination [Stanier and Lawrence 2011; Johnson 2004; Lawrence 2007]. DDGs require a construction algorithm to translate programs from languages with extensive control flow support, and a destruction algorithm to extract the control flow necessary for object code generation.

Johnson [2004] constructs the VSDG of a C program from its abstract syntax tree (AST), and excludes goto and switch statements to obtain a reducible subset of the language. Stanier [2012] extends VSDG to irreducible control flows, constructing it from the CFG by structural analysis [Sharir 1980]. Several patterns are matched in the CFG and converted with per-pattern strategies, before individual translations are connected in the VSDG. Irreducible graphs are handled by node splitting, which can lead to exponential code blowup [Carter et al. 2003]. Johnson's VSDG destruction [Johnson 2004] adds state edges until execution order is determined. Resulting CFGs follow the control flow structure of their VSDG, potentially increasing code size and branch instruction count compared to the original program. Lawrence [2007] translates the VSDG to a *program dependence graph (PDG)* [Ferrante et al. 1987] by encoding a lazy evaluation strategy, transforms the PDG to a restricted, duplication-free form, and converts it to a CFG. Introducing the PDG to refine a VSDG into exactly one CFG substantially complicates destruction, but the overhead of the resulting control flow is not quantified. To our knowledge, previous work omits quantitative analysis, or inherently features code size growth and/or suboptimal control flow recovery. No proposed algorithm has been analyzed with respect to optimality.

In this article, we show that the RSVDG representation does not impose any structural limit on control flows obtainable from it. Interpreting loop repetition and decision predicates as computed continuations, we introduce the *predicate continuation form* as a normal form, and propose algorithms for RSVDG construction and destruction. The construction handles complex control flow without node splitting, which avoids code blowup. Destruction uses the predicate continuation form to extract control flow. Multiple CFGs map to the same RVSDG because they contain inessential detail information (evaluation order of independent operations, variable names; see Section 3) that is irretrievably lost during RVSDG construction. This leads to degrees of freedom in refining an RVSDG into a specific CFG where "arbitrary" choices lead to viable results. We show that there are choices under which our destruction algorithm perfectly reconstructs the original CFG an RVSDG has been generated from (see Theorem 5.10). This leads to the insight that our algorithm is universal in the sense that it can generate *arbitrary* control flow. We prove termination and correctness of our algorithms, experimentally evaluate performance and representational overhead, and compare the

reduction of branch instructions to previous work. For practical programs, we observe that processing time and output size empirically correlate linearly with input size. This suggests that our algorithms are fit for field application. Thus, we demonstrate that control flow optimizations can be lifted to DDG representations, enabling the use of a single IR for data and control flow optimizations. This reduces use of the CFG to a step in DDG generation for languages with complex control flow.

The article is organized as follows: Section 2 introduces terminology and definitions. Section 3 describes a destruction algorithm that produces a CFG from an RVSDG in predicate continuation form. Section 4 develops a construction algorithm that restructures a CFG and translates it to an RVSDG. Section 5 proves algorithm termination and correctness, and that our destruction scheme can always recover the original CFG. We empirically evaluate our algorithms using CFGs from SPEC2006 benchmarks in Section 6. Section 7 discusses related work, and Section 8 presents our conclusions.

2. TERMINOLOGY AND DEFINITIONS

This section provides necessary terminology and definitions. It defines the CFG and RVSDG, as well as restricted subsets used throughout the article.

2.1. Control Flow Graph

A *control flow graph* **C** is a directed graph consisting of vertices representing statements and arcs representing transitions between statements. If a vertex has multiple outgoing arcs, we assign a unique index to each. Statements take the following form:

—$v_1, v_2, \ldots, v_k := expr$ designates an assignment statement. The *expr* must evaluate to a k-tuple of value and/or states[1] that are assigned to the variables on the left.
—`branch` *expr* designates a branch, where *expr* evaluates to an integer that matches an outgoing arc index. Execution resumes at the destination statement of this arc.
—`null` designates a null operation. Its operational semantics is to perform no operation; we insert it for structural reasons[2].

All vertices, except for `branch` statements, must have at most one outgoing arc. Essentially, the CFG represents a single procedure of a program in imperative form: From a given start vertex, evaluate all statements of a vertex in sequential order by strictly evaluating the expression of each statement, updating the variable state on each assignment statement, and follow alternative arcs according to `branch` statements.

Definition 2.1. A CFG is called *closed* if it has a unique *entry* and *exit* vertex with no predecessors and successors, respectively. A CFG is called *linear* if it is closed and each vertex has at most one predecessor and one successor.

Definition 2.2. The *dynamic execution trace* of a closed CFG for given arguments is the sequence of traversed vertices and outgoing arcs, starting from the entry vertex.

Figure 1 contains an example CFG. We insert a `return` statement at the end of a procedure in a CFG to clarify its result (its operational semantics is the same as `null`). Bold capital letters like **C** denote CFGs. Letters v, a, V, and A denote vertices, arcs, vertex sets, and arc sets, respectively. Without loss of generality, we assume that all CFGs are in *closed* form. We also consider CFGs of a more constrained shape:

[1]We allow multiple states to be alive at the same time here and in our later RVSDG definition if the states are independent. Use cases include, for example, modeling disjoint memory regions after aliasing analysis.
[2]It can be regarded as a "dummy" assignment operation, but we want to distinguish such structural statements from original program statements.

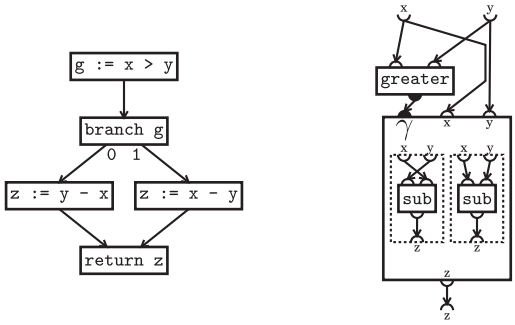

Fig. 1. CFG (*left*) and RVSDG (*right*) representations of the same function, computing $x - y$ if $x > y$ and $y - x$ otherwise.

Definition 2.3. A closed CFG is called *structured* if its shape can be contracted into a single vertex by repeatedly applying the following steps:

(1) If v' is a unique successor of v, and v is a unique predecessor of v', then remove arc $v \to v'$ and merge v and v'.
(2) If v has only successors $v_0, v_1, \ldots$ and possibly v', v' has only predecessors $v_0, v_1, \ldots$ and possibly v, and each of $v_0, v_1, \ldots$ has only a single predecessor and successor, then remove all arcs $v \to v_i$, $v_i \to v'$, $v \to v'$ and merge all vertices.
(3) If v has an arc pointing to itself and only one other successor, remove the arc $v \to v$.

Structured CFGs are a subset of reducible CFGs. They correspond to procedures with only structured control flow in the following sense: Within each structured CFG we can identify subgraphs that are structured CFGs themselves,[3] and replace them with a single vertex[4] such that the parent graph ends up in one of three possible shapes:

(1) *Linear*: A linear chain of vertices (possibly a single vertex).
(2) *Branch*: A symmetric control flow split and join, that is, an "`if/then/else`" or "`switch/case` without fall-through" construct in which each alternative path is either a linear chain or a single arc.
(3) *Loop*: A tail-controlled loop, that is, a "`do/while`" loop in which the loop body itself is a linear chain or a single arc.

Applying this recursively, we can regard each structured CFG as a *tree* of such subgraphs that we call *regions*. The CFG shown in Figure 1 is structured. We use an overline marker to designate structured CFGs: $\overline{\mathbf{C}}$.

Definition 2.4. The *structure multigraph*[5] of a closed CFG is formed by taking the original CFG and replacing all vertices with a single predecessor and successor by a direct arc from the predecessor to successor. The *projection of a dynamic execution trace* of a closed CFG is then obtained by omitting all vertices/arcs with a single predecessor and successor and projecting the remainder. Closed CFGs $\mathbf{C}$ and $\mathbf{C}'$ are *structurally equivalent* if there is an isomorphism between the two structure multigraphs such that it maps projected execution traces for the same input arguments to each other.

Intuitively, structural equivalence means that two CFGs have the same dynamic branching structure for all possible arguments.

[3]Ignoring a potential single "repetition" arc from the exit to entry vertex of a subgraph.
[4]This corresponds to contraction using the rules in Definition 2.3.
[5]We want to preserve arcs as present in the original graph, but due to the removal of vertices we may end up with multiple distinguishable arcs between two vertices. The result may not be a graph, but a multigraph.

Definition 2.5. An arc $v_1 \to v_2$ *dominates* a vertex v if either

(1) $v_2 \neq v$ and both v_1 and v_2 dominate v, or
(2) $v_2 = v$ and v_1 dominates v.

The dominator graph of arc a in $\mathbf{C}$ is the subgraph $\mathbf{S}$ with vertices V dominated by a.

Intuitively, the dominator graph of an arc a is the subgraph in $\mathbf{C}$ where every path from the entry vertex to every vertex in this subgraph must pass through arc a.

2.2. Regionalized Value State Dependence Graph

An RVSDG R is a directed, acyclic, hierarchical multigraph consisting of nodes representing computations, and edges[6] representing the forwarding of results of one computation to arguments of other computations. Each node has a number of typed *input* ports and *output* ports corresponding to parameters and results, respectively. An edge connects an input to exactly one output of matching type. The types of inputs and outputs may either be *values* or represent the *state* of an external entity involved in a computation. The distinguishing property is that values are always assumed to be copyable. The types of the input/output tuple of a node are called its *signature*. We model arguments and results of an RVSDG as free-standing ports.

The RVSDG supports two different kinds of nodes: *simple* and *complex*. Simple nodes represent primitive operations such as addition, subtraction, load, and store. They map a tuple of input values/states to a tuple of output values/states, and the node's arity and signature must therefore correspond to the represented operator. Complex nodes contain other RVSDGs, such that we can regard an RVSDG as a *tree* of such subgraphs, which we call *regions*. We require two kinds of complex nodes in this article:

—*Gamma nodes* represent decision points. Each γ-node contains two or more RVSDGs with matching result signatures. A γ-node takes as input a *decision predicate* that determines which of its regions is to be evaluated. Its other inputs are mapped to arguments of the contained RVSDGs, and its outputs are mapped to the results of the evaluated subgraph (see Figure 1).
—*Theta nodes* represent tail-controlled loops. Each θ-node contains exactly one RVSDG representing the loop body. Matching arguments/inputs and results/outputs of this region and the θ-node are used to represent the evolution of values/states through loop iterations. An additional *loop predicate* as result of the loop body determines if the loop should be repeated (Figure 2, *left*).

The RVSDG represents a single procedure of a program in demand-dependence form. Arguments and results of the root region map to arguments and results of the procedure. The semantics of an RVSDG is that evaluation is demand driven but strict, that is, all arguments of a node are evaluated before the node itself is evaluated according to the following rules:

—*Simple nodes:* After evaluation of all arguments, apply the operator represented by the node to their values, and associate the results with the node outputs.
—*Gamma nodes:* After evaluation of all arguments (including the decision predicate), the predicate value dictates which subregion is chosen for evaluation, mapping arguments of the γ–node to arguments of the sub-region. The chosen region is evaluated and its results are associated with the outputs of the $\gamma-$node.
—*Theta nodes:* After evaluation of all arguments of the θ–node, associate their values with the arguments of the subregion. Then evaluate the repetition predicate and

[6]We use the terms arc/vertex in the context of CFGs and edge/node in the context of RVSDG to assist in telling the different representations apart.

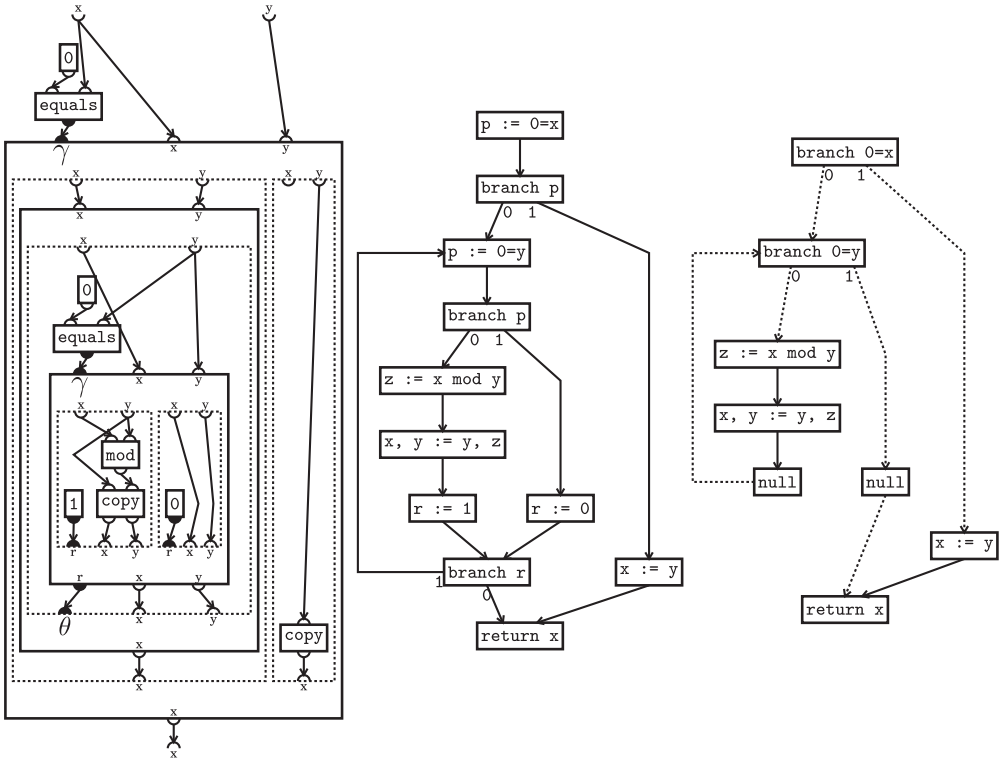

Fig. 2. Euclid's algorithm in different representations. Left: Representation of the algorithm as RVSDG. Center: CFG generated from RVSDG on the left by SCFR. Right: CFG recovered from RVSDG on the left by PCFR. The solid arcs are generated in the first pass by PredicativeControlFlowPrepare; the dotted arcs are generated in the second pass by PredicativeControlFlowFinish.

all results from the loop body. If the value of the repetition predicate is nonzero, associate result values from this loop iteration with argument values for the next loop iteration and repeat evaluation. If the value of the repetition predicate is zero, associate the results of the last repetition with the outputs of the $\theta-$node.[7]

As explained in Section 3, an RVSDG is equivalent to a structured CFG, and since every CFG can be made into a structured one by the algorithm in Section 4, the topmost region of an RVSDG allows the representation of an entire procedure. We model the arguments and results of this region as free-standing ports.

An RVSDG must obey some additional structural restrictions to satisfy the non-copyability of states. We omit a thorough discussion of how these constraints can be formulated and maintained during transformations. Instead, we only require that it satisfies the following condition: All states are used linearly, that is, in each region, a state output is connected to at most one state input.

Figure 1 illustrates a sample function in CFG and RVSDG representation: A γ-node selects the computation results from either of two embedded subgraphs, depending on its predicate input. By convention, the subgraphs embedded into a γ-node correspond

[7]Note that this definition requires that the evaluation of the loop body for one iteration is finished before the next iteration can commence. This avoids several problems, such as "deadlocks" between computation of the loop body and the predicate. It also provides well-defined behavior for nonterminating loops that keep updating external state.

from left to right to predicate values 0, 1, ... Predicate input/output ports are identified as filled sockets in diagrams.

We expect a richer type system to allow not just a distinction between values and states, but between different kind of values and/or states, for example, different value types for fixed-point and floating point numbers. For this article, we only assume the existence of a *predicate* type that allows the enumeration of alternatives. We call a node that has at least one predicate output a *predicate def node* and a node with at least one predicate input a *predicate use node*. This allows us to define a more strongly normalized form of RVSDGs:

Definition 2.6. An RVSDG is in *predicate continuation form* if

—Any predicate def node has exactly one successor
—Only one predecessor of any node may be a predicate def node
—Any predicate output must be connected to at most one predicate input, and that in turn must belong to a γ- or θ-node, or to the output of a region

Any RVSDG can easily be converted to this form by a combination of node splitting/cloning (to satisfy the requirement that any predicate def has at most one use), merging multiple independent computations into a single node (to satisfy the requirement that at most one predecessor of a node is a predicate def node), and possibly inserting nodes to convert back and forth between predicates and other kinds of values (to satisfy the last requirement). Creating this normal form is essentially a technicality that we will not discuss in further detail, since all RVSDGs occurring in this article are in this form by construction.

3. EXTRACTING CONTROL FLOW FROM THE RVSDG

In this section, we present algorithms for extracting a CFG from an RVSDG. The dependence information contained within regions serves as basis for control flow construction: It describes a partial ordering of primitive operations that any control flow recovery must adhere to in order to yield an evaluation-equivalent program.

Any γ- and θ-node-free RVSDG can be trivially transformed into a linear CFG. In the presence of γ- or θ-nodes, the resulting CFG includes branching and/or looping constructs. We explore two different approaches: *Structured control flow recovery (SCFR)* uses the hierarchical *structure* of regions contained within γ- and θ-nodes to construct control flow. *Predicative control flow recovery (PCFR)* determines control flow by *predicate* assignments.

Since the RVSDG representation is more normalizing and lacks some detail of the CFG, we must employ some auxiliary algorithms:

—EvalOrder: The exact evaluation order within an RVSDG region is underspecified. Each topological order of nodes corresponds to a valid evaluation order, and one particular order must be chosen. We add only one additional constraint to the topological order: corresponding predicate def and use nodes must be adjacent if the given RVSDG is in predicate continuation form.
—VarNames: The RVSDG is implicitly in SSA form. We therefore need to find names for variables in the CFG representation such that boundary conditions for loops and control merge points are satisfied. This requires a strategy for computing an interference-free coloring of SSA variables such that same-colored SSA names can be assigned the same name in a CFG. Additionally, the insertion of copy operations may be necessary.

We consider algorithms for these problems to be mostly outside the scope of this article, and assume that they are given. It is trivial to formulate an algorithm satisfying

EvalOrder, for example, some variant of depth first traversal. We briefly discuss VarNames in Section 3.1. Our control flow recovery algorithms are parameterized over these algorithms; we discuss how they relate to perfect CFG recovery in Section 5.

3.1. Copy Insertion and Coloring

We can formulate the conversion from an RVSDG to a CFG as a two-step process: first insert ϕ expressions at control joinpoints to obtain a CFG in SSA form, and then eliminate these ϕ expressions by inserting copy operations in the predecessors of ϕ's basic block [Cytron et al. 1991]. This process succeeds as long as no critical arcs[8] are present in a CFG [Briggs et al. 1998]. According to our definition of structured CFGs, only the loop repetition and exit arcs are critical: Structured "`if/then/else`" or "`switch/case` without fall-through" as per Definition 2.3 cannot produce critical arcs, leaving only those arcs additionally permitted through loop constructs as possibly critical. These in turn are amenable to simplistic copy insertion: choose a name for each loop variant variable, and indiscriminately assign to it on the entry, repetition, and exit path of the loop.

In order to simplify the presentation, we lift part of this process to the RVSDG representation: After determining a topological order of nodes, we insert placeholder `copy` operations into the RVSDG. These `copy` operations permit the algorithms in the following sections to directly convert an RVSDG into an SSA-free CFG, and are replaced by parallel copy operations during this conversion.[9] Having made these observations, the following rules insert a sufficient number of copy operations:

Algorithm CopyInsertion

—*For each γ-node, insert a* `copy` *operation copying each result value as the last operation in each of the alternative subregions.*
—*For each θ-node, insert a* `copy` *operation copying each loop variant value twice: Once, just before the θ-node (loop entry), and once at the last operation in the subregion (loop exit/repetition).*

This algorithm inserts many redundant copy operations. Subsequently, coalescing based on an interference graph, as per Boissinot et al. [2009] can eliminate many or even all of these copy operations.

3.2. Structured Control Flow Recovery

The naive approach to generating control flow is to treat γ- and θ-nodes as black boxes from the outside, and represent more fine-grained control flow on the inside. We call this approach *structured control flow recovery (SCFR)*, and formulate it as follows:

Algorithm StructuredControlFlow
Sequentially process all nodes of a given RVSDG. For each node, insert vertices into the CFG according to the following rules, and add arcs between them for continuation:

1. *For each simple node (including* `copy` *nodes), insert an assignment statement evaluating an expression equivalent to the semantics of the node into the CFG.*
2. *For each γ-node, recursively process the subregions into separate CFGs. Insert a* `branch` *statement corresponding to the decision predicate of the γ-node, and fan out to the entry points of the sub-CFGs. Rejoin control flow from the exits of the sub-CFGs with the next vertex to be added to the CFG.*

[8] An arc from v_1 to v_2 is critical if v_1 has multiple successors and v_2 has multiple predecessors.
[9] The evaluation semantics of these `copy` operations within the RVSDG is that they just pass through all argument values as results without change.

3. *For each θ-node, recursively process the subregion into a separate CFG. Link to the entry of the sub-CFG, and add a* `branch` *statement after the exit. This corresponds to the loop predicate of the θ-region, and either repeats the sub-CFG or continues with the next vertex to be added.*

If necessary, add a `null` vertex to rejoin any remaining control splits, or provide a loop exit point.

Figure 2 shows an example RVSDG on the left and the CFG recovered by this procedure in the center. The resulting CFG is *always* structured, and is similar to those produced by Johnson [2004]. Compared to the original CFG from which the RVSDG was constructed, this may result in a substantial overhead in terms of code size and branch instructions.

Definition 3.1. Let **R** be an RVSDG, then denote the CFG produced by algorithm StructuredControl Flow as:

$$\textsc{Scfr}(\mathbf{R}, \textsc{EvalOrder}, \textsc{VarNames})$$

3.3. Predicative Control Flow Recovery

An alternative approach is to interpret the predicate computations inside the RVSDG to determine control flow: Instead of generating `branch` vertices for γ- and θ-constructs themselves, we generate a `branch` vertex for predicate def nodes, and follow the predicate use nodes to the eventual destination. This requires an RVSDG in *predicate continuation form*, as defined in Section 2.6. The constraints of this form ensure that there is a topological order such that there is no node between predicate def/use pairs. The algorithm EvalOrder is assumed to always produce such an order, but note that CopyInsertion may insert `copy` nodes afterwards. We introduce *predicative control flow recovery (PCFR)* as the two-pass process consisting of PredicativeControlFlowPrepare and PredicativeControlFlowFinish described later. The first pass generates straight-line basic blocks; the second pass is responsible for connecting them. The diagram on the right in Figure 2 illustrates the recovery process and the generated CFG.

The first step is quite similar to StructuredControlFlow, but produces a disconnected graph. It lacks all the `branch` vertices introduced in the other algorithm:

Algorithm PredicativeControlFlowPrepare
Sequentially process all nodes of a given RVSDG according to the chosen topological order. For each node, insert vertices into the CFG according to the following rules. Add an arc to each vertex from the previously generated vertex unless the previous node in the chosen topological order is a predicate def node:

1. *For each simple node (including* `copy` *nodes), insert an assignment statement evaluating an expression equivalent to the semantics of the node into the CFG.*
2. *For each γ-node, recursively process subregions. Add the resulting sub-CFGs to the parent CFG without adding continuation arcs to the entry points.*
3. *For each θ-node, recursively process the subregion. Add the resulting sub-CFG to the parent CFG without adding any continuation arcs either to the entry or from the exit.*
4. *For each simple node that is a predicate definition, enumerate all of its possible predicate output value combinations. In case there is only one, keep track of this predicate constant. Add a* `branch` *statement with a predicate identifying the effective predicate output combination depending on its input values. In case there is only one predicate value combination, that is, a predicate constant, create a* `null` *statement instead. Keep track of the destination(s).*

Record a mapping of the vertices corresponding to each RVSDG node and region.

This leaves a graph with a number of dangling branch statements, which are then resolved in a secondary pass:

Algorithm PredicativeControlFlowFinish
Process all predicate-def simple nodes. Identify the vertex created for each in the CFG already as origin. For each possible predicate value combination, traverse to the eventual destination: Start with the subsequent node in topological order (which must be either the corresponding predicate use node or an intervening `copy` *node) and repeatedly apply the following rules until no predicate value is used further:*

1. *When reaching the origin vertex, add an arc to it, making it an infinite loop.*
2. *If n is a predicate constant, traverse to its eventual destination as if the process had originally started here.*
3. *If n is a* `copy` *node, insert a vertex with a corresponding assignment statement, and connect to it from the origin vertex. Treat the inserted vertex as the new origin, and continue tracing from the next node in topological order.*
4. *If n is not a predicate use node, terminate search at the corresponding vertex, and connect to it from the origin vertex.*
5. *If n is a γ- or θ-node, trace to the first node within the correct subregion (or the subsequent node in the parent if the subregion is empty).*
6. *If n is the result of a γ-region, traverse to the subsequent place in the parent region.*
7. *If n is the result of a θ-region, determine the value of the loop repetition predicate. If the loop is to be repeated, traverse to the entry of the region. Otherwise, traverse to the subsequent place in the parent region.*

Subsequently, prune all vertices that are not reachable from the entry point and short-circuit all `null` *vertices.*

Note that this tracing process inserts exactly one linear path per predicate-def node and possible predicate value combination generated by it. Particularly, if there is no `copy` node encountered during tracing, then it inserts exactly one arc. Noting that `copy` nodes are inserted for the purpose of SSA destruction, we observe that the additionally inserted vertices correspond to arc splits in other approaches [Briggs et al. 1998].

Definition 3.2. Let **R** be an RVSDG, then denote the CFG produced by PCFR as:

$$\textsc{Pcfr}(\mathbf{R}, \textsc{EvalOrder}, \textsc{VarNames})$$

4. TRANSFORMING CFGS TO RVSDGS

We observed that Algorithm StructuredControlFlow converts an RVSDG into a structured CFG. There is a corresponding direct transformation from any structured CFG to an RVSDG by utilizing the decomposition into regions following Definition 2.3: After logically contracting all **Branch** and **Loop** subregions into single vertices, we can recursively convert each **Linear** region into an RVSDG as follows:

—Set up a symbol table to map each variable name in the CFG to its definition place in the RVSDG. All initially defined variables will be marked as *parameters*.
—Process all vertices in topological order by the following rules:
  —For an assignment statement, generate a simple node in the RVSDG with an operation equivalent to its right-hand side. Update the symbol table.
  —If the vertex represents a **Branch** subregion, take note of the branch predicate. Recursively process each alternative path. Afterwards, generate a γ-node that uses the predicate and all variables required in the subregions as input according to the symbol table. Update the symbol table with all variables assigned to any of the alternate paths to use the new value/state defined by the γ-node.

—If the vertex represents a **Loop** subregion, recursively process the loop body. Take note of the predicate variable controlling repetition within the symbol table used in processing the subregion and generate a corresponding θ-node containing the generated RVSDG. Use the symbol table to determine the initial values/states of all loop variables as input to the θ-node, and update the symbol table to reflect the state after exiting the loop.
—Generate the results of the RVSDG using the symbol table.

Essentially, we perform a symbolic execution of the procedure represented by the CFG. Note that the algorithm does *not* assume a CFG in SSA form. This form is automatically established during construction, due to an RVSDG being implicitly in it.

Definition 4.1. For a structured CFG $\overline{\mathbf{C}}$, let the RVSDG resulting from the earlier algorithm be designated by $\textsc{BuildRvsdg}^*(\overline{\mathbf{C}})$.

The remainder of this section describes an algorithm that converts an arbitrary CFG into a structured one. In contrast to other approaches, it avoids cloning any existing nodes in the graph. Instead, it introduces fresh *auxiliary predicate variables,* which we name `p`, `q`, and `r`. These are used on the left-hand side of assignment statements and in branch statements. We refer to statements involving one of these variables as *auxiliary assignments* and *auxiliary branches*.

The algorithm consists of two parts: The first identifies and handles loops, while the second operates on acyclic graphs and only processes branch constructs. We use the following notational convention: Original graphs, arcs, and vertices are marked with plain letters such as $\mathbf{C}$, a, v, while transformed graphs and newly inserted arcs and vertices are denoted as $\mathbf{C}^*$, a^*, v^*.

Definition 4.2. For any closed CFG $\mathbf{C}$, let the structured CFG resulting from the algorithm described to follow be designated by: $\textsc{RestructureCfg}(\mathbf{C})$. Furthermore, denote by

$$\textsc{BuildRvsdg}(\mathbf{C}) := \textsc{BuildRvsdg}^*(\textsc{RestructureCfg}(\mathbf{C}))$$

the RVSDG built by combining both steps.

4.1. Loop Restructuring

Given a closed CFG $\mathbf{C}$, we start by identifying all strongly connected components (SCCs) within $\mathbf{C}$ using the algorithm described by Tarjan [1972]. By necessity, neither the entry nor the exit vertex of $\mathbf{C}$ are part of any SCC. Each SCC is restructured into a loop with a single head and tail vertex, such that the head vertex is both the single point of entry and starting point for subsequent iterations, and the tail vertex is the single vertex controlling repetition and exit from the loop.

For each SCC, we identify the following sets of arcs and vertices:

—Entry arcs $A^E = \{a_0^E, a_1^E, \ldots\}$: All arcs from a vertex outside the SCC into the SCC
—Entry vertices $v_0^E, v_1^E, \ldots, v_{m-1}^E$: All vertices that are the target of one or more entry arcs
—Exit arcs $A^X = \{a_0^X, a_1^X, \ldots\}$: All arcs from a vertex inside the SCC pointing out of the SCC
—Exit vertices $v_0^X, v_1^X, \ldots, v_{n-1}^X$: All vertices that are the target of one or more exit arcs
—Repetition arcs $A^R = \{a_0^R, a_1^R, \ldots\}$: All arcs from a vertex inside the SCC to any entry vertex

The left-hand side of Figure 3 illustrates the arc and vertex sets under consideration.

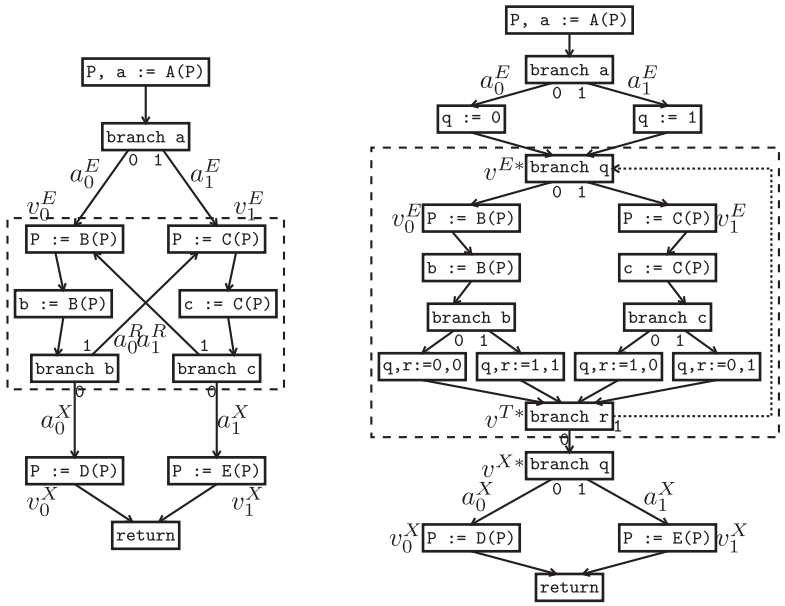

Fig. 3. Restructuring of loop control flow. Left: Unstructured CFG (loop with two entry and exit paths). The vertices within the dashed box form an SCC. Right: Restructuring the left CFG. The subgraph in the dashed box corresponds to the loop body and is treated as if it were collapsed into a single vertex from the point of view of the outer graph. The dotted arc is the single repetition arc.

Unless the loop already meets our structuring requirements, we restructure it by possibly introducing a `branch` statement as single point of entry v^{E*} and single point of exit v^{X*}. They demultiplex to the original entry vertices or exit vertices, respectively. Another `branch` statement is introduced as single control point v^{T*} at the end of the loop. A single *repetition arc* a^{R*} leads from v^{T*} to v^{E*}, and a single *exit arc* a^{X*} leads from v^{T*} to v^{X*}. Two auxiliary predicates q and r are added to facilitate demultiplexing and repetition. The original entry, exit, and repetition arcs are all replaced as follows:

—For each entry arc, identify the entry vertex v_k^E it points to. Replace the arc with an assignment `q` $:= k$ that proceeds to v^{E*}, which in turn evaluates k on entry to determine continuation at v_k^E.
—For each repetition arc, identify the entry vertex v_k^E it points to. Replace the arc with an assignment `q, r` $:= k$, `1` and funnel control flow through v^{T*}, which in turn evaluates r to determine whether the loop is to be repeated and subsequently continued at v_k^E.
—For each exit arc, identify the exit vertex v_k^X it points to. Replace the arc with an assignment `q, r` $:= k$, `0` and funnel control flow through v^{T*}, which in turn evaluates r to leave the loop. Subsequently, control flow is demultiplexed by v^{X*} to reach v_k^X.

The result of this process is illustrated on the right-hand side of Figure 3. The loop body itself is now contained in a subgraph, which we denote as $\mathbf{L}^*$. It has exactly one entry vertex and one exit arc from/to the remainder of the whole graph. For all purposes of further processing of $\mathbf{L}^*$, we will treat a^{R*} as absent.

Note that $\mathbf{L}^*$ is not necessarily acyclic: While all repetition arcs A^R were removed, there may still be nested loops within the loop body. Since $\mathbf{L}^*$ (minus a^{R*}) is a closed CFG, the algorithm can be applied recursively on $\mathbf{L}^*$. This eventually produces an

evaluation equivalent structured graph $\overline{\mathbf{L}}$, which can be substituted for $\mathbf{L}^*$ in $\mathbf{C}$. We call the result graph after the substitution of all SCCs $\mathbf{C}^*$. For further processing of $\mathbf{C}^*$, we treat all $\mathbf{L}^*$ subgraphs as if each were a single vertex.

Under this interpretation, $\mathbf{C}^*$ is now acyclic and we can apply the algorithm described in Section 4.2 to eventually produce a structured CFG $\overline{\mathbf{C}}$. Note that for the purpose of constructing an RVSDG, there is no need to actually create $\overline{\mathbf{C}}$ or even $\mathbf{C}^*$. Instead, we can recursively build for each $\mathbf{L}^*$ subordinate RVSDGs, then wrap them individually into a θ-node.

4.2. Branch Restructuring

Given an acyclic closed CFG $\mathbf{C}$, we partition the graph into a linear *head* subgraph $\mathbf{H}$, multiple *branch* subgraphs $\mathbf{B}_k$ and a *tail* subgraph $\mathbf{T}$. If $\mathbf{C}$ is linear, the partitioning results in zero branch subgraphs and an empty subgraph $\mathbf{T}$. In this case, $\mathbf{C}$ is already structured and no further steps are necessary. The branch and tail subgraphs are restructured to closed CFGs, resulting in branch subgraphs with exactly one entry arc from $\mathbf{H}$ and one exit arc to the restructured tail subgraph $\mathbf{T}^*$. The algorithm is then applied recursively to each branch and tail subgraph until we eventually obtain a structured graph $\overline{\mathbf{C}}$.

During partitioning, we first identify $\mathbf{H}$ by searching for the longest linear subgraph from the entry vertex. The last vertex of $\mathbf{H}$ must be a `branch` statement with m outgoing *fan-out arcs* $a_0^F, a_1^F, \ldots, a_{m-1}^F$. We initially identify the branch subgraph $\mathbf{B}_j$ as the dominator graphs of the arc a_j^F. As explained later, we may have to trim the sets of vertices covered by each $\mathbf{B}_j$ slightly (we denote the "pure" dominator subgraphs without trimming as $\mathbf{B}'_j$). Note that some branch subgraphs might also be empty, but we nevertheless keep track of the defining arc a_j^F. The remainder of $\mathbf{C}$ forms then the tail subgraph $\mathbf{T}$. The diagram on the left in Figure 4 illustrates the partitioning.

Let $v_0^T, v_1^T, \ldots, v_{n-1}^T$ be the *continuation points* in the tail subgraph: these are the vertices within $\mathbf{T}$ with at least one arc from either one of the branch subgraphs, or one of the fan-out arcs $a_0^F, a_1^F, \ldots, a_{m-1}^F$ pointing to them. There must be at least one such continuation point and if there is exactly one, $\mathbf{T}$ already has the desired structure.

If there are multiple continuation points, then restructure $\mathbf{T}$ and all $\mathbf{B}_j$ as follows:

—Insert a `branch p` statement as v^{T*} into $\mathbf{T}$ that demultiplexes to the original continuation points $v_0^T, v_1^T, \ldots, v_{n-1}^T$.
—Substitute each arc from any $\mathbf{B}_j$ to every v_k^T with an assignment `p` $:= k$, and funnel control flow through v^{T*}.
—If some $\mathbf{B}_j$ is empty, the fan-out arc a_j^F must point to some v_k^T. Substitute this arc with a `p` $:= k$ statement and replace the previously empty branch subgraph with this single vertex.
—If any branch subgraph has more than one exit arc to the tail subgraph, funnel all paths through a single `null` statement.

If any of the inserted `p` $:= \ldots$ statements immediately follows a `q` $:= \ldots$ statement, we fuse these two into a single vertex. This results in a new graph $\mathbf{C}^*$ with subgraphs $\mathbf{H}^*$, $\mathbf{B}_j^*$, and $\mathbf{T}^*$ corresponding to the head, branch, and tail subgraphs and $\mathbf{H}^*$ being structurally identical to $\mathbf{H}$. The diagram on the right in Figure 4 illustrates the result of this process as well as the notation. The new branch and tail subgraphs are closed; recursively applying the process to them yields structured graphs $\overline{\mathbf{B}}_0, \overline{\mathbf{B}}_1, \ldots \overline{\mathbf{B}}_{m-1}$ and $\overline{\mathbf{T}}$ (in case $\mathbf{B}_i^*$ is empty, set $\overline{\mathbf{B}}_1 := \mathbf{B}_i^*$). These subgraphs can then be substituted in $\mathbf{C}^*$ for the original branch and tail subgraphs, resulting in a structured graph $\overline{\mathbf{C}}$.

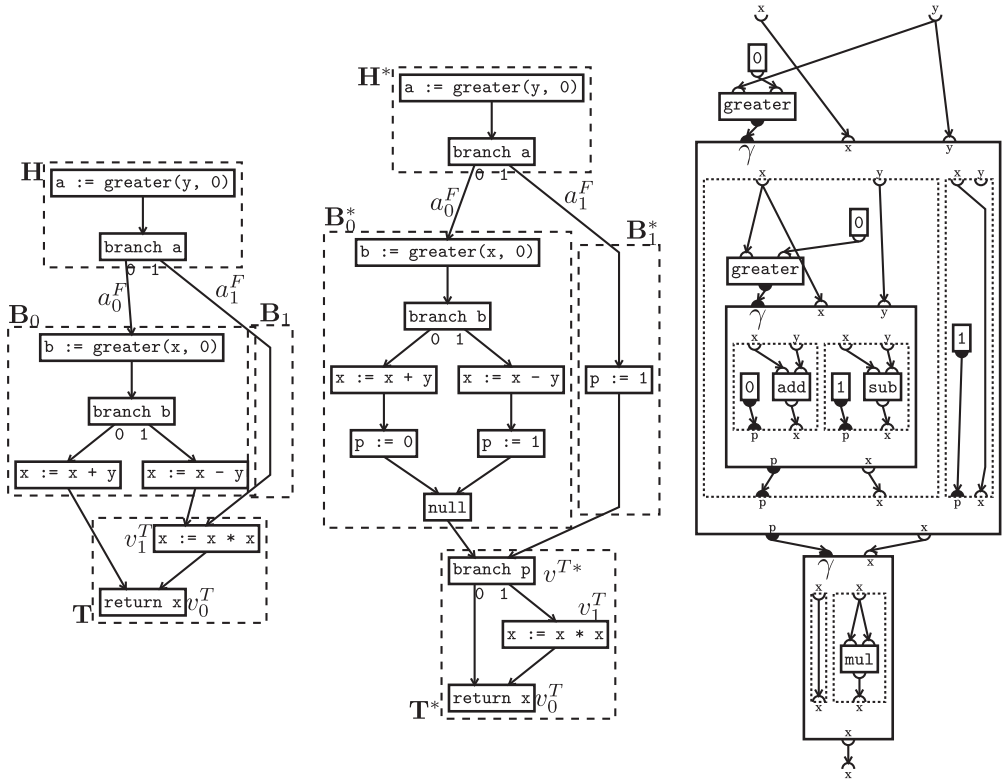

Fig. 4. Restructuring of branch control flow. Left: Unstructured CFG (inconsistent selection paths). Middle: Restructuring the left CFG. Right: The RVSDG equivalent to the CFGs to the left.

As shown in Section 4.1, it is not necessary to build the structured CFG $\overline{\mathbf{C}}$ for the purpose of RVSDG construction. Instead, the algorithm recursively builds RVSDGs. The transformed branch subgraphs are contained within a γ-node, whereas the transformed head and tail subgraphs precede and succeed the γ-node, respectively. This is illustrated in Figure 4.

4.2.1. Trimming of Branch Subgraphs. The earlier algorithm always yields a graph with the desired structure, but may interleave the defs and uses of different auxiliary predicates improperly. This results in an RVSDG that is not in predicate continuation form, which necessitates trimming the vertex sets of the branch subgraphs as follows:

—Determine from the set of continuation points the subset of vertices that are an immediate successor of an auxiliary assignment statement.
—For each such continuation point, pull its immediate predecessor from the branch subgraphs into the tail subgraph *unless all* immediate predecessors are in the branch subgraphs.

Figure 5 shows a CFG in which trimming leads to eviction of a vertex from one of the branch subgraphs and illustrates its effect: It prevents improper interleaving of auxiliary assignments and branches by ensuring that either *all* defs of a predicate are at the end of any branch subgraph or *none* are. The only possible interaction is an assignment/branch on p nested properly within an assignment/branch on q. Note

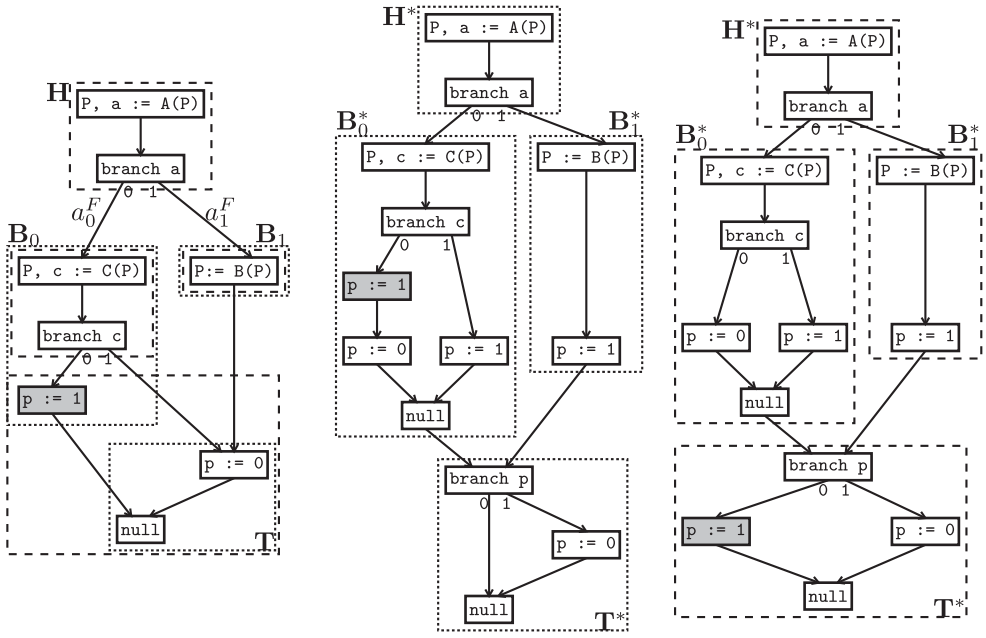

Fig. 5. Left: A CFG in which the last `null` statement is successor to auxiliary assignment statement `p := 0` and `p := 1`. Only one of these is dominated by any of the fan-out arcs a_0^F and a_1^F and is therefore a *critical assignment*. The dotted boxes illustrate the subgraphs without any trimming; the dashed boxes show the effect of trimming. The critical assignment is shaded. Middle: Effect of applying branch restructuring without trimming. The improper interleaving of operations leads to the critical assignment being "lost." Right: Applying branch restructuring after trimming. No vertex is inserted on a path from the critical assignment to the exit vertex.

that according to the rules outlined previously, the assignments are fused into a single vertex and translated as a single predicate defining node in the RVSDG.

5. PROOF OF CORRECTNESS AND INVERTIBILITY

In this section, we prove correctness of the presented algorithms and that the proposed RVSDG construction and destruction algorithms are mutually inverse to each other.

5.1. Termination

Definition 5.1. A CFG is said to be *pqr-complete* in this case: If one predecessor of a vertex is an *auxiliary assignment* statement, then all of its predecessors are.

THEOREM 5.2. *For a given acyclic, closed, pqr-complete and nonlinear CFG* $\mathbf{C}$*, the partitioning step in Section 4.2 yields graphs* $\mathbf{T}$*,* $\mathbf{B}_0$*,* $\mathbf{B}_1, \ldots,$ $\mathbf{H}$ *such that* $\mathbf{H}$*,* $\mathbf{T}$ *and at least one* $\mathbf{B}_i$ *is nonempty.*

PROOF. $\mathbf{C}$ is closed, let v^E and v^X be the entry and exit vertices. By construction, it is evident that $v^E \in \mathbf{H}$ and $v^X \in \mathbf{T}$, so both are nonempty. The only remaining proof obligation is to show that one $\mathbf{B}_i$ is nonempty. We proceed with a few auxiliary propositions, then show that some $\mathbf{B}_i'$ (dominator graph before trimming) is nonempty, then show the same for some $\mathbf{B}_i$ (dominator graph after trimming).

Let v^B be the uniquely determined last `branch` vertex of $\mathbf{H}$. For each vertex, let $d(v)$ be the depth (length of longest path from entry) of vertex v. Since $\mathbf{H}$ is linear with only v^B having immediate successors outside of $\mathbf{H}$ by construction, it must hold $d(v) < |\mathbf{H}|$

for all $v \in \mathbf{H}$, $d(v) \geq |\mathbf{H}|$ for all $v \notin \mathbf{H}$ and $d(v) < d(v^X)$ for all $v \neq v^X$. Furthermore, if $d(v) \geq |\mathbf{H}|$ for some vertex v, then for any of its immediate predecessors v' it must be that either $d(v') \geq |\mathbf{H}|$ or $v' = v^B$.

We first show that for any vertex v with $d(v) = |\mathbf{H}|$ that $v \in \mathbf{B}'_i$ for some i: v cannot be in $\mathbf{H}$, but all of its predecessors must be. Since $\mathbf{H}$ is linear with only the fan-out arcs $a^F_0, a^F_1, \ldots, a^F_{m-1}$ of its last `branch` vertex pointing to any vertex outside of $\mathbf{H}$, this `branch` vertex must be the single predecessor of v. Since there can be at most one arc between any two vertices, v must be dominated by some arc a^F_i, hence $v \in \mathbf{B}'_i$.

There must be at least one arc v^R with $d(v^R) = |\mathbf{H}|$ because $d(v^X) > |\mathbf{H}|$, therefore at least one $\mathbf{B}'_i$ must be nonempty. Since $v^X \notin \mathbf{B}'_j$ for any j, this in turn means that there must be a vertex $d(v^S) = |\mathbf{H}| + 1$.

Let P be the set of immediate predecessors of v^S. For each $v \in P$, it is either $d(v) = d(v^S) - 1 = |\mathbf{H}|$ or $v = v^B$, and there must be at least one $v^P \in P$ such that $d(v^P) = |\mathbf{H}|$. If none of the elements of P are auxiliary assignments, then $v^P \in \mathbf{B}_i$ because it is not subject to trimming, completing our proof. Otherwise by pqr-completeness all elements of P must be auxiliary assignment vertices. This means that $v^B \notin P$ (because v^B is a `branch` statement), and therefore $d(v) = |\mathbf{H}|$ for all $v \in P$. This means that each $v \in P$ is also in some dominator subgraph $\mathbf{B}'_i$. This in turn means that they are not removed by trimming; therefore, $v^P \in \mathbf{B}_i$ for some i as well. □

THEOREM 5.3. *The branch restructuring algorithm given in Section 4.2 always terminates for a given acyclic, closed and pqr-complete CFG* $\mathbf{C}$.

PROOF. We prove this by showing that there is a strictly decreasing well-ordered metric for the arguments of the recursive function calls, and that the arguments to these calls satisfy the preconditions of this theorem.

For any $\mathbf{C}$ let $b(\mathbf{C})$ denote the number of `branch` vertices, and $|\mathbf{C}|$ denote the number of vertices. Define $m(\mathbf{C}) := (b(\mathbf{C}), |\mathbf{C}|)$ with the well-ordering relation

$$m(\mathbf{C}) < m(\mathbf{C}') \leftrightarrow b(\mathbf{C}) < b(\mathbf{C}') \vee (b(\mathbf{C}) = b(\mathbf{C}') \wedge |\mathbf{C}| < |\mathbf{C}'|)$$

and will show that $m(\mathbf{T}^*) < m(\mathbf{C})$ and $m(\mathbf{B}^*_i) < m(\mathbf{C})$.

If $b(\mathbf{C}) = 0$, then the graph is linear and will be returned unchanged without recursion. If $b(\mathbf{C}) \neq 0$, we know by Theorem 5.2 that $\mathbf{H}$, $\mathbf{T}$ and at least one $\mathbf{B}_i$ contain at least one vertex of $\mathbf{C}$ each, and $\mathbf{H}$ contains a `branch` vertex. This means that $|\mathbf{T}| \leq |\mathbf{C}| - 2$, $b(\mathbf{T}) \leq b(\mathbf{C}) - 1$; since we insert at most one `branch` vertex into the tail subgraph, it follows that $|\mathbf{T}^*| \leq |\mathbf{C}| - 1 \wedge b(\mathbf{T}^*) \leq b(\mathbf{C}) - 1$. We only insert assignment and `null` statements into the branch subgraphs; therefore $b(\mathbf{B}^*_j) \leq b(\mathbf{C}) - 1$.

The head, tail, and branch subgraphs of $\mathbf{C}$ are initially pqr-complete. The only possible insertion into the tail subgraph is one `branch` vertex and into the branch subgraphs are auxiliary assignment statements that fan in to a `null` statement. All of these insertions preserve the initial pqr-complete property. □

THEOREM 5.4. *The loop restructuring algorithm given in Section 4.1 terminates for any given closed pqr-complete CFG* $\mathbf{C}$.

PROOF. As presented earlier, we give a well-ordered metric for the arguments of recursive calls. Let $s(\mathbf{C}) := \sum_{S \in SCC\mathbf{C}} |S|$ be the sum of all vertices in any nontrivial, strongly connected component. By necessity, $s(\mathbf{C}) \leq |\mathbf{C}| - 2$, since the entry and exit vertex cannot be in any SCC. Furthermore, the total size is also an upper bound for the size of each SCC individually.

$\mathbf{L}^*$ consists of all vertices of one SCC plus at most two auxiliary vertices. Neither the newly added auxiliary entry and exit vertices can be part of any SCC, nor can the

original entry vertices as we remove all repetition arcs. Therefore, $s(\mathbf{L}^*) \leq |\mathbf{L}^*| - 2 - 1 \leq s(\mathbf{C}) - 1$.

The introduced auxiliary assignment statements amount to assignments to `q` which fan into the entry of the loop, and assignments to `q`, `r` which fan to the tail vertex of the loop. Consequently, all graphs processed either by recursion or branch restructuring share this property. By assumption, the auxiliary predicates `p`, `q`, and `r` are not used anywhere in any original CFG on which restructuring operates, concluding the proof that the algorithm terminates. □

These theorems show that the algorithms are structurally recursive, and we can prove properties about them inductively.

5.2. CFG Restructuring Correctness and Evaluation Equivalence

THEOREM 5.5. *For any closed CFG* $\mathbf{C}$*, the restructuring algorithm of Section 4 yields a structured CFG* $\overline{\mathbf{C}}$.

PROOF. We prove by induction over the recursive call tree, first for branch restructuring as per Section 4.2: All linear graphs at the call tree leafs are structured by rule 1 of Definition 2.3. $\mathbf{H}^*$ is linear; by induction hypothesis all of $\overline{\mathbf{B}}_i$ and $\overline{\mathbf{T}}$ are either structured or empty. Contracting all of these subgraphs leaves a graph shaped according to rule 2, which is therefore structured as well.

Loop restructuring as per Section 4.1 calls into branch restructuring at its leafs, and by induction hypothesis we can again presume $\overline{\mathbf{L}}$ to be structured. Rule 3 allows removing the repetition arc after contracting the loop body. By applying branch restructuring while treating all loops as indivisible vertices, we therefore arrive at a structured CFG again. □

THEOREM 5.6. *For each arc from* v_1 *and* v_2 *in* $\mathbf{C}$*, there is either a corresponding arc in* $\overline{\mathbf{C}}$ *or a path from* v_1 *to* v_2 *that consists entirely of matching auxiliary assignment and auxiliary branch vertices as well as null statements such that control must flow from* v_1 *to* v_2.

PROOF. Evidently, each individual insertion of auxiliary statements during the restructuring process already satisfies the proposition. We only need to show that there is no interaction that disturbs the proper matching of assignment and `branch` statements. The only places where auxiliary vertices can be inserted improperly is during branch restructuring between subgraphs $\mathbf{B}_i$ and $\mathbf{T}$. Due to pqr-completeness, the trimming step ensures that all insertions occur in a uniformly nested fashion within other existing auxiliary assignment and `branch` statements. Thus, we only need to prove that there is no path with assignments to `p` without an intervening `branch p` statement. We assert that auxiliary statements only occur in CFGs processed recursively as:

(1) `branch p` as the entry vertex of a graph, or
(2) `p` := ... as the exit vertex of a graph, or
(3) `p` := ... as predecessors to a `null` exit vertex

In the first two cases, there is nothing to prove since no vertices are inserted before entry or after the exit vertex. In the third case, the trimming step ensures that either all or none of the `p` := ... statements is contained in the branch subgraphs. In the first case, there is nothing to prove since there are no insertions between vertices in the tail subgraph. In the second case, pqr-completeness ensures that the tail subgraph consists of a single `null` vertex and that there is consequently no auxiliary assignment or branch statement inserted. □

The last theorem not only proves the evaluation equivalence of the original and restructured CFG, but also shows that there is a simple mechanical procedure for recovering the original CFG. The following algorithm is a very restricted form of constant propagation that only considers constant assignments to any of the auxiliary variables:

Algorithm ShortCircuitCFG
Repeatedly apply these transformations:

—*Replace every arc to a* `null` *vertex with an arc to its successor, and remove the vertex from the graph.*
—*Let* $x_1, x_2, \ldots$ *be names of variables and* $expr_1, expr_2, \ldots$ *be expressions for a statement of the form* $x_1, x_2, \ldots := expr_1, expr_2, \ldots$ *followed by a* `branch` x_i *vertex, then substitute* x_i *for* $expr_i$ *in the* `branch` *statement and remove* x_i *and* $expr_i$ *from the assignment statement. If no variables and expressions remain, replace every arc to it with an arc to its successor and remove the assignment statement from the graph.*
—*Let c be a constant expression, then replace a* `branch` *c statement with a* `null` *statement with an outgoing arc to the destination of the branch corresponding to c.*

Corollary 5.7. *For any closed CFG* $\mathbf{C}$, *the following is required to hold:*

$$\mathbf{C} = \textsc{ShortCircuitCfg}(\textsc{RestructureCfg}(\mathbf{C}))$$

5.3. CFG Reconstruction by PCFR

Theorem 5.8. *For any given structured CFG* $\overline{\mathbf{C}}$, *there are oracles for* EvalOrder *and* VarNames *such that*

(1) VarNames *succeeds without introducing* `copy` *operations, and*
(2) *the following equation holds:*

$$\textsc{Scfr}(\textsc{BuildRvsdg}^*(\overline{\mathbf{C}}), \textsc{EvalOrder}, \textsc{VarNames}) = \overline{\mathbf{C}}$$

Proof. Let $\mathbf{R} := \textsc{BuildRvsdg}^*(\overline{\mathbf{C}})$. The nodes of each region of $\mathbf{R}$ were constructed by traversing the corresponding part of $\overline{\mathbf{C}}$ in control flow order, and recording just the value and state dependencies of operations. Therefore, a topological order of RVSDG nodes corresponding to the original control flow order exists and can be supplied by an oracle for EvalOrder.

During RVSDG construction, we kept updating a symbol table maintaining the mapping from variable name in the original CFG and output of a node in the RVSDG. Inverting this mapping means that we can recover the assignment of def sites in the RVSDG to names in the original CFG. This yields an interference-free coloring, showing that VarNames can succeed without insertion of `copy` operations and recover the original CFG names.

Together, this shows that Scfr perfectly reconstructs $\overline{\mathbf{C}}$. □

Theorem 5.9. *For an RVSDG* $\mathbf{R}$ *in predicate continuation form and algorithms* EvalOrder *and* VarNames *such that* VarNames *succeeds without introduction of* `copy` *nodes, the following identity is required to hold:*

$$\begin{aligned}&\textsc{Pcfr}(\mathbf{R}, \textsc{EvalOrder}, \textsc{VarNames})\\ &= \textsc{ShortCircuitCfg}(\textsc{Scfr}(\mathbf{R}, \textsc{EvalOrder}, \textsc{VarNames}))\end{aligned}$$

Proof. Let $\overline{\mathbf{C}} := \textsc{Scfr}(\mathbf{R}, \textsc{EvalOrder}, \textsc{VarNames})$. By assumption, the chosen topological order keeps predicate def and use nodes adjacent, and also VarNames does not insert nodes. This ensures that $\overline{\mathbf{C}}$ in turn has assignments to predicate variables immediately

succeeded by any `branch` statement(s) using the value—with possibly some intervening `null` statements that are eliminated as the first step of ShortCircuitCfg.

PredicativeControlFlowPrepare produces the same set of vertices as SCFR, except that predicate defining RVSDG nodes are translated as `branch`/`null` instead of assignment statements and that no `branch` vertices are generated at the head or tail of γ- or θ-constructs, respectively (see Figure 2 for an illustration).

PredicativeControlFlowFinish evaluates the same expressions for determining the continuation points as ShortCircuitCfg does during `branch` replacement steps. Combined with the structural correspondence, this shows that both arrive at the same CFG. □

With these preparations, we can formulate and prove the main theorem of this article:

Theorem 5.10. *For each closed CFG* $\mathbf{C}$, *there exist oracles* EvalOrder *and* VarNames *such that:*

$$\mathbf{C} = \text{Pcfr}(\text{BuildRvsdg}(\mathbf{C}), \text{EvalOrder}, \text{VarNames})$$

.

Proof. Combining the earlier theorems, we know there exist oracles for EvalOrder and VarNames such that we can rewrite:

$$\begin{aligned}\mathbf{C} &= \text{Pcfr}(\text{BuildRvsdg}(\mathbf{C}), \text{EvalOrder}, \text{VarNames})\\ &= \text{ShortCircuitCfg}(\text{Scfr}(\text{BuildRvsdg}^*(\text{RestructureCfg}(\mathbf{C})),\\ &\quad \text{EvalOrder}, \text{VarNames}))\\ &= \text{ShortCircuitCfg}(\text{RestructureCfg}(\mathbf{C}))\\ &= \mathbf{C} \quad \square\end{aligned}$$

The next theorem shows that the degrees of freedom in undoing SSA has no material influence on the control flow structure:

Theorem 5.11. *For each closed CFG* $\mathbf{C}$, *there exists an oracle* EvalOrder *such that for any* VarNames *and* $\mathbf{C}' = \text{Pcfr}(\text{BuildRvsdg}(\mathbf{C}), \text{EvalOrder}, \text{VarNames})$ *it holds that* $\mathbf{C}'$ *and* $\mathbf{C}$ *are structurally equivalent (see Definition 2.4).*

Proof. Using the same oracle for EvalOrder as in the previous theorem, we observe that Pcfr processes all RVSDG nodes in the same sequence in both cases. This leads to the same order of CFG nodes, but $\mathbf{C}'$ may differ from $\mathbf{C}$ in two aspects: It may use different variable names and contain several variable copy operations (see Section 3.1) that are not present in $\mathbf{C}$. This still leads to the same structure multigraph and unchanged evaluation semantics, hence the two CFGs are structurally equivalent. □

Exact reconstruction of the original control flow depends on selecting a specific topological ordering of nodes per region. An RVSDG in which topological ordering is sufficiently constrained will always faithfully reproduce the original control flow structure. Otherwise, different orders may result in wildly different CFG shapes. Still, we retain as an invariant that the number of branching points does not increase, which shows that our destruction algorithms never deteriorate control flow as measured by static branch counts:

Theorem 5.12. *For each closed CFG* $\mathbf{C}$, *any* EvalOrder, *any* VarNames *and* $\mathbf{C}' = \text{Pcfr}(\text{BuildRvsdg}(\mathbf{C}), \text{EvalOrder}, \text{VarNames})$, *it holds that* $\mathbf{C}'$ *has the same number of* `branch` *statements.*

	bzip2	cactusADM	calculix	gcc	gobmk	gromacs	h264ref	hmmer	lbm	libquantum	mcf	milc	perlbench	sjeng	sphinx3	wrf
Linear	25	632	9	1353	1552	334	81	71	2	26	1	78	329	28	60	80
Structured	34	429	534	1362	616	238	184	124	5	17	2	42	517	43	63	180
Reducible	41	288	577	1582	474	626	301	329	12	58	21	115	394	73	190	175
Irreducible	2	-	-	2	-	-	-	-	-	-	-	-	10	-	-	-
Total	102	1349	1120	4299	2642	1198	566	524	19	101	24	235	1250	144	313	435

Fig. 6. Classification of extracted CFGs.

PROOF. The number of branches generated by PCFR only depends on the number of predicate assignments within the RVSDG, and the number of auxiliary predicate assignments "skipped" by algorithm PREDICATIVECONTROLFLOWFINISH. The former is evidently independent of algorithm choices for EVALORDER and VARNAMES. The independence of the latter is a consequence of EVALORDER keeping predicate defs and uses adjacent. □

6. EMPIRICAL EVALUATION

This section describes the results of applying PCFR and SCFR to CFGs extracted from the SPEC2006 benchmark suite [Henning 2006] in order to evaluate practical implications for the number of generated branches, IR size, and compile-time overhead.

6.1. CFG Extraction and Representation

CFGs were extracted from all C benchmarks in the SPEC2006 using *libclang*[10], which resulted in a set of 14321 CFGs. We ensured that all were closed by eliminating statically unreachable basic blocks. All operations were modeled as having side effects, consuming a single program state, and producing one for the next operation. This was done to ensure a unique topological order for reconstructing the exact CFG.

CFGs were classified as *Linear* per Definition 2.1, *Structured* per Definition 2.3, *Reducible,* or *Irreducible*. Structured graphs were identified by structural analysis [Sharir 1980], and irreducibility was determined by T1/T2 analysis [Aho et al. 2006]. Figure 6 shows distributions of each class per program. The CFGs were converted to RVSDGs as described in Section 4, and restored to CFGs using PCFR and SCFR. All PCFR results were verified to equal their original CFGs.

6.2. Key Observations

PCFR removes every auxiliary assignment and `branch` statement introduced by CFG restructuring, avoiding any code size increases from node splitting or additional branches to preserve program logic. Figure 7 shows its improvement over SCFR in terms of `branch` statements. Static branch counts are not directly proportional to program runtime, but we regard them as indicative of overheads incurred by other methods [Janssen and Corporaal 1997; Unger and Mueller 2002; Stanier and Watson 2011]. We note that graphs generated by SCFR greatly resemble results produced by Johnson's algorithm [Johnson 2004], suggesting that their overheads are comparable.

RVSDG construction as per Section 4 can add constructs with no equivalent in the original CFG, depending on the complexity of the original control flow. Consequently,

[10]http://clang.llvm.org/.

	bzip2	cactusADM	calculix	gcc	gobmk	gromacs	h264ref	hmmer	lbm	libquantum	mcf	milc	perlbench	sjeng	sphinx3	wrf
SCFR	1440	5802	9373	35827	9679	5486	6136	3766	56	305	190	1160	9374	2236	1543	1284
PCFR	1100	5127	8752	31960	8329	5277	5867	3343	55	280	160	1109	8248	1913	1436	1135
Savings %	**30.9**	**13.2**	**7.1**	**12.1**	**16.2**	**4.0**	**4.6**	**12.7**	**1.8**	**8.9**	**18.8**	**4.6**	**13.7**	**16.9**	**7.5**	**13.1**

Fig. 7. Relative static branch counts of PCFR to SCFR.

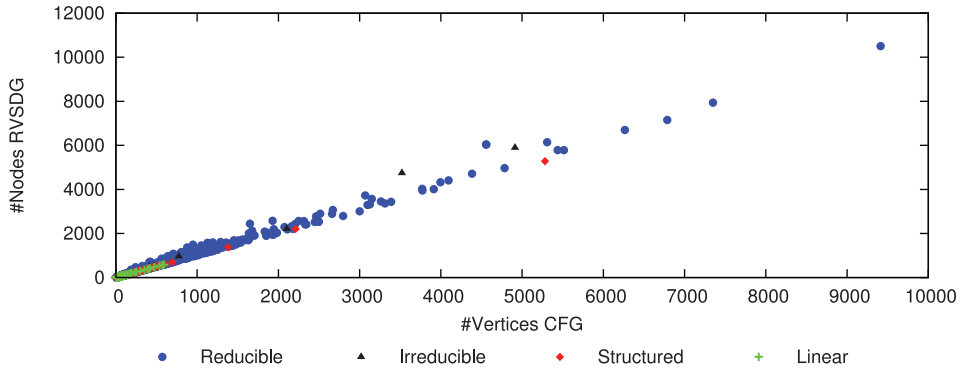

Fig. 8. Corresponding CFG and RSVDG sizes.

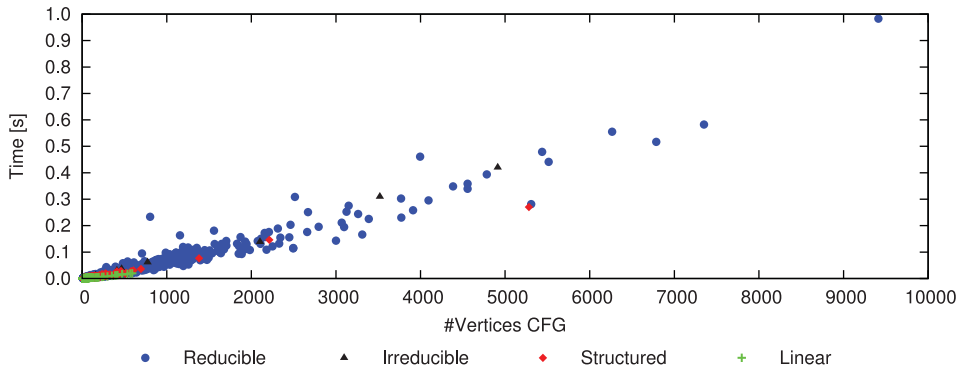

Fig. 9. Execution times for CFG to RVSDG conversions.

our RVSDG form carries an expected representational overhead. This is quantified in Figure 8, which relates the number of CFG vertices to RVSDG nodes for each of our CFGs. There is a clear linear relationship for all of the 14321 cases, suggesting that our construction algorithm is practically feasible in terms of space requirements.

Construction and destruction were timed for each CFG to assess how compile-time overhead grows with graph size. Figure 9 shows timings of RVSDG construction versus input size. For the majority of CFGs, a linear dependency is recognizable, with a few notable deviations from the expected line. Deviations below the line are either structured or nearly structured graphs. For these graphs, restructuring only discovers hierarchical structure without compounding it, resulting in lower processing time. The

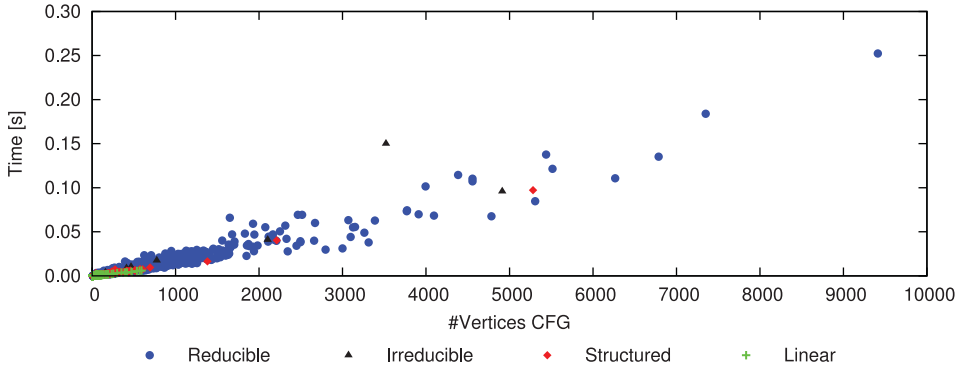

Fig. 10. Execution times for RVSDG to CFG conversions.

7 deviations above the line are either produced by large `if/then/else if` statements, or complex control flow of interacting loops, `switch` and `goto` statements, which produced graphs with hundreds of cascaded equality tests on one side. Through branch restructuring, such imbalanced graphs lead to mostly empty branch subgraphs, and one full branch subgraph. The dominator graph is repeatedly recomputed for the full branch subgraph, and thus for most vertices in the CFG. This implies a theoretical worst-case quadratic complexity for such cases.

Figure 10 shows timings of RVSDG destruction via PCFR versus input size. A linear tendency is also visible here, with a few deviations below and above the line. The deviations below the line are due to graphs with a high degree of linearity. In comparison to the number of total nodes, few predicate def nodes need to be traced to their corresponding use nodes, which results in low processing time. The few points above the line are again due to extremely unstructured and complex control flow. Considering that PredicativeControlFlowFinish repeatedly traces through nested RVSDG regions for similar use/def chains starting at different nesting depths, we expect its computational complexity to be worst-case quadratic in terms of input size. Experiments with different compilers suggest that these also exhibit nonlinear processing time for similarly complex control flow.

6.3. Summary

The overall conclusion of the empirical study is that PCFR permits CFGs to be reconstructed from their RVSDG representation without producing any control flow overhead for actual benchmark programs. In addition, the IR size and compile-time processing overheads are predominantly linear in terms of input size. We, therefore, conclude that PCFR enables an RSVDG representation to precisely capture a CFG structure without substantial disadvantages.

7. RELATED WORK

7.1. Intermediate Representations

Many different IRs emerged for simplifying program analysis and optimizations [Stanier and Watson 2013]. The RVSDG is related to SSA [Cytron et al. 1989], gated SSA [Tu and Padua 1995], and thinned gated SSA [Havlak 1993]. It shares the same γ function with (thinned) gated SSA and its θ construct is closely related to their μ and η functions. However, all three of these SSA variants are bound to the CFG as an underlying representation. Many optimizations such as constant propagation [Wegman and

Zadeck 1991] or value numbering [Rosen et al. 1988] rely on the SSA form to achieve improved results or an efficient implementation.

The Program Dependence Graph (PDG) [Ferrante et al. 1987] is a directed graph incorporating control and dataflow in one representation. It has statement, predicate expression, and region nodes, and edges represent data or control dependencies. Control flow is explicitly represented through a control dependence subgraph, resulting in cycles in the presence of loops. The PDG has been used for generating vectorized code [Baxter and Bauer 1989] and partition programs for parallel execution [Sarkar 1991].

The Program Dependence Web (PDW) [Ottenstein et al. 1990] combines gated SSA with the PDG and is suitable for the development of different architecture back-ends. It allows a direct interpretation under three different execution models: control-, data-, or demand-driven. However, its construction requires 5 passes over the PDG, resulting in a time complexity of $O(N^3)$ in the size of the program.

Click's IR [Click and Paleczny 1995] is a PDG variant with a Petri net-based model of execution. Nodes represent operations, and edges divide into control and data dependence subsets. It was designed for simplicity and speed of compilation, and a modified version is used in the Java HotSpot server compiler [Paleczny et al. 2001].

The Value Dependence Graph (VDG) [Weise et al. 1994] is a directed graph in which nodes represent operations and edges dependencies between them. It uses γ-nodes to represent selection, but lacks a special construct for loops. Instead, they are expressed as tail-recursive functions with λ-nodes. The significant problem with the VDG is that it fails to preserve the termination properties of a procedure.

7.2. Control Flow Restructuring

Many control flow restructuring algorithms have been proposed to facilitate optimizations. Most work focuses on removing goto statements, or irreducible control flow.

Erosa and Hendren [1994] eliminate goto statements in C programs by applying AST transformations. The algorithm works on the language-dependent AST, but is also applicable to other languages [Ceccato et al. 2008]. However, it replicates code, and the simple elimination of gotos is insufficient for RVSDG construction.

Zhang and D'Hollander [1994, 2004] create single-entry/-exit subgraphs within CFGs. As with Erosa's work, the method suffers from code expansion and produces CFGs not suitable for RVSDG construction.

The algorithm presented by Ammarguellat [1992] operates on its own input language, but is closest to our approach in terms of produced control flow. The paper presents an algebraic framework for normalizing control flow to only three structures: single-entry/-exit loops, conditionals and assignments [Böhm and Jacopini 1979]. The algorithm represents a procedure as a set of continuation equations and solves this system with a Gaussian elimination-like method. In contrast to Erosa's and Zhang's work, code is only duplicated for irreducible graphs.

Janssen and Corporaal [1997] and Unger and Mueller [2002] use controlled node splitting to transform irreducible CFGs to reducible ones. Although irreducible control flow is extremely rare in practice [Stanier and Watson 2011] and their results are encouraging in terms of code bloat, it is proven that node splitting results in exponential blowup for particular CFGs [Carter et al. 2003].

8. CONCLUSION

In this article, we presented algorithms for constructing RVSDGs from CFGs and vice versa. We proved the correctness of these algorithms, and that the exact, original control flow can be recovered after roundtrip conversion. Empirically, we found that our algorithms successfully process CFGs extracted from selected SPEC2006 benchmarks. Processing time and output size correlate linearly with input size, with acceptable

deviations for less than 0.01% of test cases. While pathological inputs may cause worse asymptotic behavior, we conclude that the algorithms are applicable to practical programs.

Our main result is that our algorithms obtain control flows that are not limited by any structural properties of the RVSDG. Expressing control flow transformations as predicate computations permits related optimizations to be moved entirely into the RVSDG domain. This provides a means to translate optimizations that depend on established control flow, such as Johnson and Mycroft's VSDG-based code motion and register allocation scheme [Johnson and Mycroft 2003], and may remove the need to perform control flow transformations in any other representation. Ultimately, our work enables a compiler to use a demand-dependence representation up until object code generation, when deciding on a specific flow of control is inevitable.

REFERENCES

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools* (2nd ed.). Addison-Wesley Longman, Inc., New York, NY.

Zahira Ammarguellat. 1992. A control-flow normalization algorithm and its complexity. *IEEE Transactions on Software Engineering* 18, 237–251.

W. Baxter and H. R. Bauer, III. 1989. The program dependence graph and vectorization. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'89)*. ACM Press, New York, NY, 1–11.

orrado Böhm and Giuseppe Jacopini. 1979. Classics in software engineering. Chapter Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules. Yourdon Press, New York, NY, 11–25.

Benoit Boissinot, Alain Darte, Fabrice Rastello, Beno Dupont de Dinechin, and Christophe Guillon. 2009. Revisiting Out-of-SSA translation for correctness, code quality and efficiency. In *CGO* (2010-03-22). IEEE Computer Society, 114–125. Retrieved November 29, 2014 from http://dblp.uni-trier.de/db/conf/cgo/cgo2009.html#BoissinotDRDG09.

Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. 1998. Practical improvements to the construction and destruction of static single assignment form. *Softw. Pract. Exper.* 28, 8, 859–881.

Larry Carter, Jeanne Ferrante, and Clark D. Thomborson. 2003. Folklore confirmed: Reducible flow graphs are exponentially larger. In *POPL*. ACM, 106–114. ACM SIGPLAN Notices 38(1).

Mariano Ceccato, Paolo Tonella, and Cristina Matteotti. 2008. Goto elimination strategies in the migration of legacy code to Java. In *Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR'08)*. 3–62.

Cliff Click and Michael Paleczny. 1995. A simple graph-based intermediate representation. *SIGPLAN Not.* 30, 3, 35–49.

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1989. An efficient method of computing static single assignment form. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of ProgrammingLanguages (POPL'89)*. ACM Press, New York, NY, 25–35.

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. *Efficiently Computing Static Single Assignment Form and the Control Dependence Graph*. Technical Report. Providence, RI.

Ana Erosa and Laurie J. Hendren. 1994. Taming control flow: A structured approach to eliminating goto statements. In *Proceedings of 1994 IEEE International Conference on Computer Languages*. IEEE Computer Society Press, 229–240.

Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. 1987. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.* 9, 3, 319–349.

Paul Havlak. 1993. Construction of thinned gated single-assignment form. In *Proceedings of the 6th Workshop on Programming Languages and Compilers for Parallel Computing*. Springer-Verlag, 477–499.

John L. Henning. 2006. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News* 34, 4, 1–17.

Johan Janssen and Henk Corporaal. 1997. Making graphs reducible with controlled node splitting. *ACM Trans. Program. Lang. Syst.* 19, 6, 1031–1052.

Neil Johnson and Alan Mycroft. 2003. Combined code motion and register allocation using the value state dependence graph. In *CC (Lecture Notes in Computer Science)*, Vol. 2622. Springer, New York, 1–16.

Neil E. Johnson. 2004. *Code size optimization for embedded processors*. Technical Report UCAM-CL-TR-607. University of Cambridge, Computer Laboratory, Cambridge, England.

Alan C. Lawrence. 2007. *Optimizing Compilation with the Value State Dependence Graph*. Technical Report UCAM-CL-TR-705. University of Cambridge, Computer Laboratory, Cambridge, England.

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. *SIGPLAN Not.* 25, 6, 257–271.

Michael Paleczny, Christopher Vick, and Cliff Click. 2001. The Java hotspotTM server compiler. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium—Volume 1 (JVM'01)*. USENIX Association, 1–1.

Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1988. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'88)*. ACM Press, New York, NY, 12–27.

Vivek Sarkar. 1991. Automatic partitioning of a program dependence graph into parallel tasks. *IBM J. Res. Dev.* 35, 5–6, 779–804.

Micha Sharir. 1980. Structural analysis: A new approach to flow analysis in optimizing compilers. *Comput. Lang.* 5, 3–4, 141–153.

James Stanier. 2012. *Removing and Restoring Control Flow with the Value State Dependence Graph*. Ph.D. Dissertation. University of Sussex, East Sussex, England.

James Stanier and Alan Lawrence. 2011. The value state dependence graph revisited. In *Proceedings of the Workshop on Intermediate Representations*, Florent Bouchez, Sebastian Hack, and Eelco Visser (Eds.). 53–60.

James Stanier and Des Watson. 2011. A study of irreducibility in C programs. *Softw: Pract. Exper.* 42, 1, 117–130.

James Stanier and Des Watson. 2013. Intermediate representations in imperative compilers: A survey. *ACM Comput. Surv.* 45, 3, 26:1–26:27.

Robert Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM J. Comput.* 1, 2, 146–160.

Peng Tu and David Padua. 1995. Efficient building and placing of gating functions. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation (PLDI'95)*. ACM, New York, NY, 47–55.

Sebastian Unger and Frank Mueller. 2002. Handling irreducible loops: Optimized node splitting versus DJ-graphs. *ACM Trans. Program. Lang. Syst.* 24, 4, 299–333.

Mark N. Wegman and F. Kenneth Zadeck. 1991. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.* 13, 2, 181–210.

Daniel Weise, Roger F. Crew, Michael Ernst, and Bjarne Steensgaard. 1994. Value dependence graphs: Representation without taxation. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'94)*. ACM, New York, NY, 297–310.

Fubo Zhang and Erik H. D'Hollander. 1994. Extracting the parallelism in program with unstructured control statements. In *Proceedings of the 1994 International Conference on Parallel and Distributed Systems*. 264–270.

Fubo Zhang and Erik H. D'Hollander. 2004. Using hammock graphs to structure programs. *IEEE Trans. Softw. Eng.* 30, 4, 231–245.

Received May 2014; revised November 2014; accepted November 2014