Removing and Restoring Control Flow with the Value State Dependence Graph

James Stanier
Foundations of Software Systems
School of Informatics
University of Sussex

A thesis submitted, on July 23rd, 2011, in partial fulfilment of the requirements for the degree of Doctor of Philosophy (DPhil) in the School of Informatics at the University of Sussex.

To Mum, Dad and Rebecca

Statement of Originality

This thesis is my own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and the Acknowledgements.

This thesis is not substantially the same as any that I have submitted or am currently submitting for a degree, diploma or any other qualification at any other university. No part of this dissertation has already been, or is being submitted for any such degree, diploma or qualification.

July 23rd, 2011

Abstract

This thesis studies the practicality of compiling with only data flow information. Specifically, we focus on the challenges that arise when using the Value State Dependence Graph (VSDG) as an intermediate representation (IR).

We perform a detailed survey of IRs in the literature in order to discover trends over time, and we classify them by their features in a taxonomy. We see how the VSDG fits into the IR landscape, and look at the divide between academia and the "real world" in terms of compiler technology. Since most data flow IRs cannot be constructed for irreducible programs, we perform an empirical study of irreducibility in current versions of open source software, and then compare them with older versions of the same software. We also study machine-generated C code from a variety of different software tools. We show that irreducibility is no longer a problem, and is becoming less so with time. We then address the problem of constructing the VSDG. Since previous approaches in the literature have been poorly documented or ignored altogether, we give our approach to constructing the VSDG from a common IR: the Control Flow Graph. We show how our approach is independent of the source and target language, how it is able to handle unstructured control flow, and how it is able to transform irreducible programs on the fly. Once the VSDG is constructed, we implement Lawrence's proceduralisation algorithm in order to encode an evaluation strategy whilst translating the program into a parallel representation: the Program Dependence Graph. From here, we implement scheduling and then code generation using the LLVM compiler. We compare our compiler framework against several existing compilers, and show how removing control flow with the VSDG and then restoring it later can produce high quality code. We also examine specific situations where the VSDG can put pressure on existing code generators. Our results show that the VSDG represents a radically different, yet practical, approach to compilation.

Acknowledgements

Firstly I would like to thank Dr. Des Watson, who has shown great encouragement for the project, and has always provided invaluable guidance. If I didn't have such a great experience being supervised as an undergraduate, then I probably would have never shown an interest in academia in the first place. However, I did, and look where we are now.

My parents deserve countless thanks for everything they have done for me since 1985. They always encouraged me at school and convinced me that I could make something of myself. My hope for the future is that I can continue to make them feel proud.

Thank you to past and present students in the Foundations of Software Systems research group at the University of Sussex, who have become good friends since 2008. These are Dr. Anirban Basu, Dr. Jian Li, Dr. Roya Feizy, Dr. Yasir Malkani, Dr. Lachhman Dhomeja, Stephen Naicken, Simon Fleming, Danny Matthews, Ben Horsfall, Tom Harvey and Morteza Kheirkhah. Additionally, I'd like to thank Dr. Alan Lawrence, Dr. Neil Johnson and Prof. Alan Mycroft for their feedback and advice.

I also extend my thanks to the ACM, who have provided me with a great distraction as Departments Chief of ACM XRDS magazine.

And last, but not least, thank you to Rebecca Harrison for reminding me that there are more important things in life than just science.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 The importance of compilers

When Grace Murray Hopper developed the compiler for the `A-0 System` language in 1952 [102], considered the first ever compiler for an electronic computer, it is unlikely she was aware of the compelling, complicated field of computing that compiler technology would become. When programming in today's most popular modern languages such as Java, C or C++ [8], one can easily forget the extent of the processing and translation that occurs before the written program can run on the target hardware, comprised of increasingly complicated and powerful digital circuitry. Our rapid-pace, technology-driven society would not be possible without the essential tool that evolved from Hopper's original work.

`A-0` was not like today's complex compilers. By storing subroutines on tape, the programmer could write in mathematical notation in order to be able to call stored subroutines. Further development resulted in the ability for a computer to recognise English commands, showing that programming could potentially be more understandable to humans. In addition to the fact that compilers allow languages to be more literate, they also aid in writing more powerful, concise code. Soon after Hopper's original paper was published, John Backus proposed a new language for the mainframe computers at IBM. This would eventually become FORTRAN [92]. The language sported one of the first compilers to use *optimisation*, a technique which attempts to improve generated code by means of analysis and transformation. The true meaning of "improve" is dependent on the optimising criteria; for example, the user may want code that executes in the fastest possible time, or code that takes up the least possible space in memory.

The principal rewards of compiler technology – the ability to support more complex and useful programming languages, combined with the need to produce faster, better code – has encouraged computer scientists to continually develop compilers to accept more intricate language syntax, and to perform

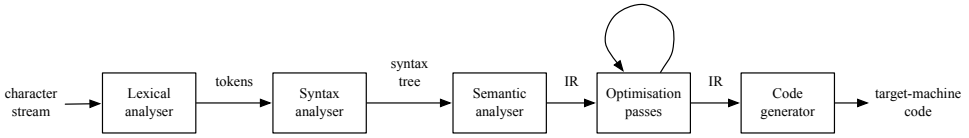


Figure 1.1: A typical compiler structure split into phases.

highly complex optimisations, often exploiting specific properties of the target architecture for maximum possible gain.

1.1.1 A compiler's function

At an abstract level, a compiler is only doing one apparent task: reading the source program given to it, and then generating a semantically equivalent program in the required target language as output. However, in practice, compilers perform many different stages of analysis and transformation in order to achieve a correct result.

Most modern compilers are similar in their general structure. Broadly speaking, a typical compiler is split into two logically disparate phases: the analysis of the source program and the synthesis of the desired target program. These two phases are commonly referred to as the "front-end" and "back-end" of a compiler. In Figure 1.1 we show a typical compiler structure. The front-end breaks up the source program into basic segments (often called tokens) and then checks that their grammatical structure is valid with reference to the input language specification. Additionally, information is extracted and stored about the input program for use in later stages of compilation. The result of this analysis stage is normally a machine-independent representation of the source program, which is then used by the latter phases of the compiler to construct an equivalent program in the target language.

While the theory and application of the compiler front-end has remained mostly "solved" for a time, the implementation of the intermediate representation (IR) phase and the quest to achieve high-quality code generation still attracts active research. Programming languages have remained fairly similar in structure over time, but hardware is constantly evolving, resulting in the need for more flexible IRs and a wider range of code generators.

1.1.2 Optimising compilers

Producing *correct* code is the fundamental goal of a compiler. However, the ability to optimise code, both during and after compilation, yields a number of benefits. The aim of optimisation is to produce target code that has better *performance* than if the optimisation passes were not undertaken. In a general sense, good performance indicates that the optimised code will run faster than

before. However, the metric by which performance is judged can differ depending on the context in which the optimisations are being performed. For a program running on a desktop computer with an abundance of storage, speed may be the primary concern. Yet, with an increasing number of embedded systems with limitations on power and space, the overall size of the code may be important, or the desire that the code uses as little power as possible when running. Therefore, it is necessary to treat optimisations as passes over a representation in the compiler that is machine-independent, as analysis and transformation is therefore free from being bounded by specifics of the target code, and optimisations can take place without worrying about machine-level specifics. In most modern compilers, the majority of optimisations are carried on the IR, which is often a graph-like structure.

Most, if not all, modern compilers perform optimisations on an IR called the Control Flow Graph (CFG) [12]. This seminal IR was introduced to explicitly represent all possible control flow paths in a program. A CFG is a directed graph $G = (V, E)$ consisting of nodes V and edges E. Nodes are commonly called basic blocks and contain instructions. Edges show possible paths of execution. When control enters a basic block it does so at the first instruction and can only leave through the last instruction. An edge (a, b) indicates that control may pass from a to b once the last instruction in a has executed. Representing a program as a CFG is now a highly commonplace activity: it is the IR of choice in many mainstream and research compilers such as GCC [2] and LLVM [82]. Yet, many now-standard optimisations are based on data flow information; that is, identifying which parts of the program depend other parts in order to execute. Thus, graph-based data flow IRs were designed to make data flow optimisations easier to perform. In these graphs, nodes represent operations such as `add`, and edges represent the flow of data from the result of one operation to the input of another. An operation can execute at any time after all of its input data values have been computed. When an operation executes, it produces a new value which is propagated to other connected operations. However, early data flow IRs were unable to deal with explicit control flow concepts such as loops and conditional branches, meaning these IRs had to be used in conjunction with the CFG.

Since compilers must generate correct code, compiler writers have always been conservative with optimisations. More specifically, compilers are traditionally very conservative with the order of instructions in the input program, hence the almost universal adoption of the CFG as the IR of choice. But, on a grander scale, should we pay so much attention to the specific instruction order that the programmer has specified? We may be able to do better by stripping away all but essential control flow, optimising with greater freedom, and then restoring the control flow later. As compilers are always under scrutiny to produce better and faster code, this is a bold, yet exciting, avenue to explore.

Functional, *demand-based* IRs have shown promise. These IRs try to discard explicit representation of control flow and infer control flow from properties of

data flow, and thus eliminate the CFG as the basis of analysis and transformation. For example, the Value Dependence Graph (VDG) [125] used *selectors* to implement `if` and `switch` statements: a condition is evaluated, then the value is *demanded* from the true or false connection of the selector, depending on the result of this condition. Loops were implemented as function calls with tail recursion. However, the authors noted that they suffered from a termination problem: *"evaluation of the VDG may terminate even if the original program did not"*.

The Value State Dependence Graph (VSDG) [72] extended the concept of the VDG by introducing *state edges*. If two nodes are connected by a state edge, then this enforces that those operations must be performed in that order. This overcame the termination problem with the VDG. The VSDG is a sparse data flow IR that contains *enough* program information to *eliminate* the CFG, while exposing the program to a variety of simple, powerful optimisations.

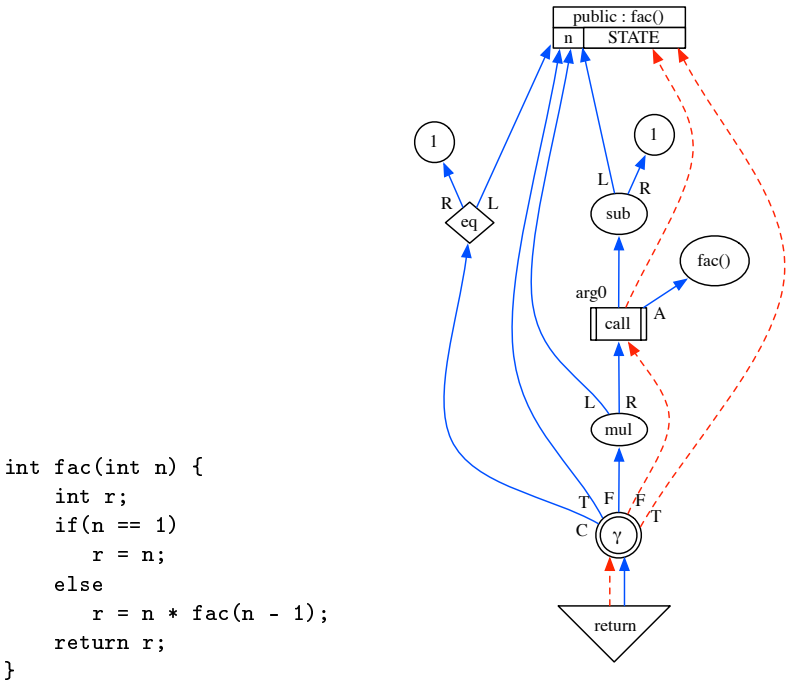
1.2 The VSDG: some motivation

The VSDG is a functional IR that represents programs with reducible control flow [65] purely as data flow. Formally, a VSDG is a labelled directed graph $G = (N, E_V, E_S, \ell, N_0, N_\infty)$ consisting of nodes N (with unique entry node N_0 and exit node N_∞), value dependency edges $E_V \subseteq N \times N$ and state dependency edges $E_S \subseteq N \times N$. The labelling function ℓ associates each node with an operator.

There are four different types of node in the VSDG. The first are value nodes. These represent pure operations such as `add` and `sub`. The second are state nodes. These are side effecting operations such as `load` and `store`. The third are γ nodes which implement the behaviour of conditional statements by demanding a value from one or other input according to the value returned from their condition input. The fourth are θ nodes which implement the behaviour of loops.

Two different types of edge are present in the graph. Value edges indicate the flow of data between nodes in the same way as other data flow IRs. For example, an `add` instruction will have two operands connected by separate value edges, such as `5` and `2`. State edges represent sequential dependencies in the program. For example, assume a memory location `x`. A particular `load x` operation may come before a `store x`, and a state edge between these operations ensures that this is always the case. Linearity constraints ensure only one state can be "live" at any given time, i.e. stateful operations consume and destroy the previous state; this ensures the semantics of the input program are maintained.

As an example, we show C code for a factorial function in Figure 1.2 and its corresponding VSDG. In this example, we can see most of the different VSDG nodes. N_0 is the box at the top of the diagram and N_∞ is the `return`




```
int fac(int n) {
    int r;
    if(n == 1)
       r = n;
    else
       r = n * fac(n - 1);
    return r;
}
```

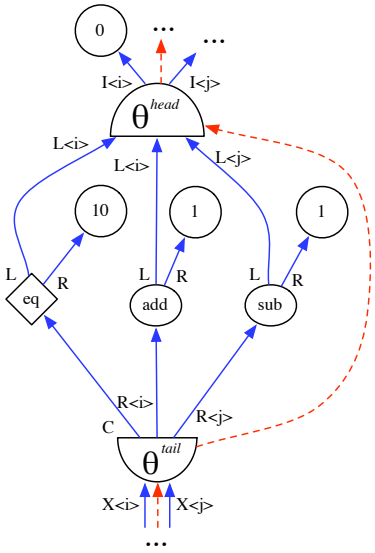
(a) (b)

Figure 1.2: C code and the corresponding VSDG for a factorial function.

node. Edges connect to and from labelled ports, which are annotated in the diagram. Value edges are solid (blue) and state edges are dashed (red). For example, the `sub` (subtract) node has L and R ports for the left- and right-hand operands respectively. The `call` node has an $arg0$ port for the first and only argument to the function being called, and an A port to specify the name of this function. The box at the top of the VSDG represents the function inputs, in this case the value `n` and the state. Edges are drawn in the *direction of demand*. The `return` node demands both state and value, represented by the dashed state edge and the solid value edge. These are connected to the γ node which represents the `if` statement in the program. The γ node demands the condition value through the C port; if this yields true, it demands and returns a value from the T port, otherwise from F. The value edge labelled C demands the result of the `eq` node, which in turn demands the constant value `1` and the value `n` passed as a parameter to the function. If the γ node condition is true, then the parameter value of `n` is demanded. If it is false, then the parameter value of `n` is multiplied with the result of the `call` to the function labelled `fac()` with one argument: `n` $-$ `1`. Note that the γ node also has T and F state edges. If the C port evaluates to true, the edge to the original function state reflects that the recursive call *must not be made*; if false, the state edge to `call` reflects that it must be.

```
j = ...
for(i = 0; i < 10; i++) {
   -j;
}
...  = j;
```

(a)



(b)

Figure 1.3: A `for` loop and corresponding θ nodes.

1.2.1 Loops

Loops are represented by θ nodes. A θ^{head} node acts as a loop header, and the θ^{tail} node acts as a loop exit. In Figure 1.3 we show a `for` loop and its corresponding θ node representation. θ nodes are interpreted as follows. On the first iteration, the loop values of `i` and `j` are set to the initial values demanded through the θ^{head} I ports. These ports are annotated as `I<i>` and `I<j>` in the diagram. These initial values are now available to be demanded through the θ^{head} L ports. Next, the condition demanded through θ^{tail} C port is evaluated. If this evaluates to true then all θ^{tail} R port values are demanded, causing evaluation of the loop body. These new R port values for `i` and `j` become the θ^{head} L port values for the next iteration. This behaviour continues until the θ^{tail} C port evaluates to false, and the current values for `i` and `j` are returned through the θ^{tail} X port. Loops always consume and produce state, unless their termination is guaranteed by appropriate analysis. In this particular example loop the body has no side effects, so the state edge from θ^{tail} points to θ^{head}. Semantics for the VSDG have been given along with a more detailed definition and well-formedness conditions, but we omit these for space, referring the reader instead to Johnson's thesis [72].

1.2.2 Optimisation

The VSDG is an excellent tool for optimisation. Many classical optimisations can be performed on the VSDG using a combination of graph rewriting and node or edge marking. Graph rewriting optimisations involve replacing a subgraph of the VSDG with an alternative one that is considered better by some given criteria. For example, the strength reduction optimisation [10] can be performed by finding and then replacing operators with cheaper alternatives. Rewriting optimisations can be continually performed until no further subgraphs are matched. Marking optimisations walk over the graph using traditional graph traversal techniques whilst marking nodes and edges according to some criteria. As an example optimisation, we will show how dead node elimination can be performed.

Dead node elimination is a combination of dead code elimination and unreachable code elimination. Dead code is that which has no effect on the result of the function it is in. Unreachable code is code that will never execute. Because the VSDG is a data flow IR, dead code generates nodes which have no value or state edges connected to them because their values are never demanded. Unreachable code generates VSDG nodes that are either dead, or become dead after other optimisations. For example, a γ node may have a constant value being demanded through the C port, so the branch can be removed. Therefore, dead node elimination is a simple node marking algorithm (Algorithm 1). This runs in $O(N)$ time, where N is the number of nodes in the VSDG. The VSDG has been used to implement a number of classical and novel optimisations [72, 123]. Many come "for free" just by representing a program as a VSDG, such as this dead node elimination technique.

Algorithm 1: Dead node elimination on the VSDG.

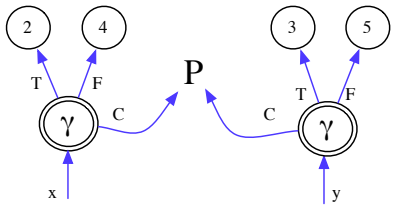
Input : A VSDG $G(N, E_V, E_S, N_\infty)$ with zero or more dead nodes.
Output: A VSDG with no dead nodes.

1. WalkAndMark(n) =
   if n is marked then return;
   mark n;
   $\forall\{m | m \in N \wedge (n, m) \in (E_V \cup E_S)\}$ do
   WalkAndMark(m);
   in WalkAndMark(N_∞);
2. $\forall n \in N$ do
   if n is unmarked then *delete*(n).

```
a) if (P)
       x = 2; y = 3;
   else
       x = 4; y = 5;

b) if(P) x = 2; else x = 4;
   if(P) y = 3; else y = 5;
```

(a)



(b)

Figure 1.4: Two syntactically different `if` statements become the same VSDG.

1.2.3 Normalisation

We say a compiler *normalises* input programs $P_1, \ldots, P_n$ (with different source codes) if they are translated into the same output program (executable). If this happens, then those source programs were different methods of implementing the same computation, and had the same observable semantics. The more normalising a compiler is, the less the programmer has to worry about optimising their code by hand: the compiler deals with this for them. Thus, we claim that the most optimising compiler is the most normalising compiler, assuming correct and efficient programs are being generated. This allows the programmer to think about their code at higher levels of abstraction.

The VSDG is a highly normalising IR. Since programs are represented by data flow and only the essential control flow enforced by state edges, different source programs computing the same underlying idea can take the same form in the VSDG. For example, consider the two syntactically different programs in Figure 1.4. Despite being written in different ways, the underlying computations are the same, and have the same observable semantics. This means that in the VSDG representation they have the same structure.

Broadly speaking, we say that the VSDG represents *what* needs to be computed in a given program. Conversely, the CFG represents *how* it has to be computed. By saying *what* needs to be computed rather than *how*, optimisers have greater freedom to move and rewrite instructions to improve the program, without having to worry about the original order of instructions that the source program specified. Thus, the VSDG is more normalising than the CFG and other control flow based IRs. This property makes the VSDG an excellent tool for optimisation.

1.3 Generating code from the VSDG

If we eliminate the CFG by translating a program into a VSDG, we are faced with a challenge in the back-end of the compiler: restoration of control flow. This process is called *sequentialisation*. Being able to produce an *optimal*

sequentialisation was shown to be NP-complete [122]. A sequentialisation is optimal if it is:

Dynamically optimal No redundant computations are performed on any path in the CFG.

Statically optimal It has the smallest size amongst all dynamically optimal sequentialisations [123].

Since an optimal sequentialisation involves some element of search, Lawrence [83] investigated a different approach to generating code from the VSDG, proposing that the back-end of a VSDG compiler could consist of two distinct parts:

1. Translating the VSDG into a Program Dependence Graph (PDG) [57], an existing *parallel* IR, representing both control flow and data flow information. This stage encodes a lazy evaluation strategy in a similar manner to the evaluation of functional programming languages.

2. Translating this PDG into a CFG using existing techniques, from which code can then be generated.

But how effective is this technique? Also, aside from the challenge of generating sequential code from a VSDG, there exist various other complications that pose difficulties for a potential VSDG compiler. What are these complications, and how can we overcome them?

1.4 Contributions

In this thesis we are interested in practical factors that relate to implementing a VSDG compiler. We are concerned with the *whole* compiler, from VSDG construction through to code generation. This thesis makes the following contributions.

The IR landscape Which IRs have been used in compilers over time, and how can we classify them? In Chapter 2 we study IR developments from the birth of compiler technology through to the present day, in order to see how the VSDG fits into the IR landscape. We categorise IRs by their features in a taxonomy, and also look at the divide between academia and the "real world" in terms of compiler technology. We are aware of no similar survey in the literature.

Irreducibility Currently, the VSDG, like many other data flow IRs, cannot be built for irreducible programs. In Chapter 3 we explore the meaning of irreducibility and show why it causes a problem for compilers. We

perform an empirical study of irreducibility in current versions of open source software, and then compare them with older versions of the same software. We also study machine-generated C code from a variety of different software tools. We use this to decide whether irreducibility is a problem that can prevent IRs like the VSDG from becoming widely used.

Construction Previous approaches to constructing the VSDG and similar IRs have been poorly documented or ignored in the literature altogether. In Chapter 4 we provide a thorough approach to constructing the VSDG from the CFG of an input program. We show how our approach is independent of the source and target language, how it handles unstructured control flow, and also how it is able to transform irreducible programs on the fly. We know of no other construction algorithm that can perform all of these techniques.

Proceduralisation With the VSDG constructed, in Chapter 5 we implement Lawrence's proceduralisation algorithms [83] on the VSDG. We show how a naïve approach to proceduralisation results in an illegal PDG, as defined by Ferrante et al [56]. Then, we implement Lawrence's effective algorithm by analysing the VSDG using gating conditions, and then using this information to guide the construction of the PDG. We show how this avoids illegal PDGs by a process called γ-ordering. We extend Lawrence's algorithm to include loops in the PDG, and give an example of how to proceduralise nested loops, thus bridging the gap between Johnson's and Lawrence's work. No implementation of this technique has been previously performed.

Sequentialisation With the PDG generated from Chapter 5, Chapter 6 details our implementation of sequentialisation from the PDG. We show how Lawrence's algorithm can generate PDGs that are not well-formed, and give our strategy for restoring well-formedness. Then, we give our approach to scheduling the PDG, and show how to generate code from statements, conditional branches and loops. We also show how to generate LLVM IR, suitable for input into existing code generators.

Evaluation In Chapter 7 we compare our compiler framework against several existing compilers. We study the effects of removing and restoring control flow with the VSDG in comparison to CFG-based compilers. We then look specifically at the PDGs generated by γ-ordering, and how this technique puts pressure on code generators.

Conclusion Chapter 8 concludes and suggests potential directions for further research.

1.5 Published work

Parts of this thesis are published in or are related to:

1. **Stanier, J**. *Graphs and Gating Functions.* Section in the forthcoming textbook "SSA-based Compiler Design", Springer, 2012.

2. **Stanier, J** and Watson, D. 2011. *A Study of Irreducibility in C Programs.* To appear in Software: Practice and Experience.

3. **Stanier, J** and Lawrence, A. 2011. *The Value State Dependence Graph Revisited.* In Proceedings of the Workshop on Intermediate Representations (co-located with CGO 2011), pages 53-60.

4. **Stanier, J** and Watson, D. 2011. *Intermediate Representations in Compilers: A Survey.* Accepted for publication in ACM Computing Surveys.

Chapter 2

Intermediate Representations in Compilers: A Survey

2.1 Introduction

Compilers are an essential tool in software development. Without compilers, we would not have the efficient support for the wide variety of expressive high-level programming languages available today. Better compilers have allowed for more powerful programming languages, raising the level of abstraction for the programmer, and making code easier to write. However, compilers do more than just translating source code into target code. They are also able to optimise the source program in order to make it better by some criteria. For example, the user may want the smallest code possible, or the fastest code. Since compilers can (and should) optimise code, it makes sense to translate the source program into a data structure which makes optimisation easier.

A compiler commonly constructs an **intermediate representation** (IR) [10] which is an internal form of a program created during compilation [119]. This structure forms the start and end point of a number of analyses and transformations performed during compilation. Many compilers use more than one IR during the course of compilation [118].

Figure 1.1 shows a typical compiler structure as a number of phases. To begin with, the source program is read into the compiler and split into its atomic syntactic components by the lexical analyser. These syntactic components are called tokens and are passed to the syntax analyser, where their order is inspected against a formal definition of the language to ensure the source program is syntactically correct. Typically a syntax tree and symbol table are produced by this stage, and these data structures are used by the semantic analyser to perform type checking and type conversions, amongst other analyses concerned with correctness. After this stage, the IR is generated. The IR can then be used to perform a number of analyses and transformations to improve the program. Then, the target machine code is generated from the IR.

The *front-end* of a compiler is considered to be all phases before optimisation on the IR: the lexical analyser, syntax analyser and semantic analyser. The *back-end* is commonly considered to be all phases after machine-independent optimisation: namely all stages of code generation.

The IR is not just present to act as a vehicle for code optimisation. It can play a key role in compiler implementation, where front-ends (for different source languages) and back-ends (for different target architectures) can share a common IR, resulting in a significant reduction of effort when implementing multiple compilers. This idea is not new [41], but it has never been satisfactorily implemented although the use of a common IR for a small range of similar programming languages is not uncommon. Furthermore, there are many examples of compilers where back-ends, based on the same IR, are available for a wide range of target architectures. Some compiler projects have made use of the IR to support rapid implementation via interpretation. The IR produced by the front-end can be *interpreted*, rather than code generated. For example, the BCPL language [96] could be implemented on a new machine using a bootstrapping process based on an IR called INTCODE. This is a simple and compact representation for which an interpreter can be written easily. Once an interpretive implementation is available, work can start on a conventional back-end based on the use of the more complex OCODE IR.

Currently, there are a wide variety of different IRs in use, both in the literature and in real world compilers. Different IRs are used for different purposes and each has its own benefits and drawbacks. The first objective of this chapter is to provide a detailed overview of the IR landscape. To accomplish this we propose an IR taxonomy, then identify the components and design purposes of different IRs and group them accordingly. The second objective is to look at the size of the technology gap between academic or commercial research and real world compilers in terms of their IRs. We do this by providing a timeline of IR developments in research, and surveying the IR technology used in current compilers. We are unaware of any survey of IRs, apart from two existing bibliographies [37, 89].

This chapter presents the following:

- We outline some common terms and definitions which we will use throughout this thesis.
- We explain our taxonomy for classifying IRs.
- Then, we look at a number of popular linear and graphical IRs, detailing their design and common uses.
- We look at the uses of these IRs in the literature and also in "real-world" compilers. We use this information to comment on the technology gap between the two areas.

```
a = b + c;
x = a * d;
if(x == y)
     z = e;
else
     z = f;
y = z + 1;
return y;
```

(a)

Figure 2.1: Some example code containing an `if` statement.

2.2 Terms and definitions

Before we present our taxonomy, we will outline some terms and definitions that are used repeatedly throughout this thesis. Firstly, we frequently refer to the definition of a *directed graph* which is an ordered pair $G = (V, E)$ comprising of a set V of nodes and a set E of edges where $E \in V \times V$. Many IRs are represented as graphs. A *path* from a vertex v_0 to v_n in a graph is a sequence of nodes $v_0, v_1, ..., v_{n-1}, v_n$ which all are connected by edges in E. If a path $v_0, ..., v_0$ exists in E then the graph has a cycle and is therefore called cyclic. If no such path exists, the graph is acyclic [26]. A *strongly connected component* of a graph is a subgraph in which all nodes in the subgraph are reachable by all other nodes in the subgraph. A *back edge* in a directed graph is one that points to an ancestor in a depth-first traversal. A *bipartite* graph is a set of graph nodes, which when decomposed into two disjoint sets, no two graph nodes within the same set are adjacent. *Single-entry single-exit* (SESE) analysis finds subgraphs of a directed graph that have exactly one incoming edge and one outgoing edge.

We also refer to dependence relationships in programs. The first of these is *control dependence*. Control dependence arises from execution order constraints within the program. For example, consider the code fragment in Figure 2.1. Here, the value that is assigned to `z` is control dependent on the outcome of the `if` statement guard: if it evaluates to true, then `z = e`, else `z = f`. The second is *data dependence*. This arises from the flow of data in the program. In Figure 2.1 the statement `x = a * d` has a data dependence on the previous statement `a = b + c`, since the latter must execute for the result of `a` to be available for the execution of the former. Related to data dependence, a *def-use chain* for a variable connects a definition of that variable to all of the uses it may reach.

We also refer to whether a program exhibits *reducible* control flow. If a

program does not have reducible control flow, then it is *irreducible*. Irreducible programs prevent compilers from optimising loops. Given a directed graph, it is reducible if we can repeatedly perform transformations T_1 and T_2 until the graph has been transformed into a single node. The resulting graph is called the *limit graph*. Assuming we are analysing a directed graph G, the transformations are as follows:

T_1 Suppose n is a node in G with a *self-loop*, that is, an edge from n to itself. Transformation T_1 on node n is the removal of this self-loop.

T_2 Let n_1 and n_2 be nodes in G such that n_2 has the unique direct ancestor n_1, and n_2 is not the initial node. Then transformation T_2 on node pair (n_1, n_2) is merging nodes n_1 and n_2 into one node, named n_1/n_2, and deleting the unique edge between them [65].

These transformations are confluent: the same limit graph will be reached regardless of the order of application. If there is more than one node in the limit graph, the CFG is said to be *irreducible*.

Trees are data structures that feature often in compilation. A tree consists of one or more nodes. Exactly one node is the root of the tree. All nodes except the root have exactly one parent; the root has no parents. Edges connect parents to children. A node with no children is called a leaf. Nodes with one or more children are called interior nodes. Preorder and postorder traversals are two special cases of depth-first search in which the children of each node are visited left to right. Compilers often traverse trees and then perform some action at each node. If an action is done when a node is first visited, then the traversal is preorder. If it is done when the node is left for the last time, then the traversal is postorder.

Construction of an IR is the process that builds it from whichever form an input program is in. Destruction of an IR is the process that translates it into some target format, whether that be machine code or another IR.

2.3 IR taxonomy

We now present a simple IR taxonomy (Figure 2.2) which we will use to group IRs. The taxonomy consists of three categories in which IRs can exhibit characteristics. The first category is **structure**. The structure of an IR can be divided into two broad sub-categories:

Linear The IR represents pseudocode for a machine. This varies from relatively high-level instructions to low-level instructions similar to assembly language.

Graphical The IR represents program information in the form of a graph.

Structure	Dependence	Content
Linear	None	Full
Graphical (cyclic)	Control	Partial
Graphical (acyclic)	Data	
	Hybrid	

Figure 2.2: A simple IR taxonomy.

We choose to split graphical IRs into two further sub-categories: **cyclic** and **acyclic**, based on these graph theoretic properties.

The second category is the **dependence** information represented:

None The IR is not designed to highlight any dependence information.

Control The IR represents relationships explicitly in terms of the control dependencies between variables or sequences of instructions in the program.

Data The IR represents relationships explicitly in terms of the data dependencies between variables or sequences of instructions in the program.

Hybrid The IR highlights both control and data dependency information.

The third category is the program **content** contained within the IR:

Full The compiler is able to generate target code with only the information present in the IR.

Partial The compiler requires more information, stored externally from the IR, in order to generate target code.

In this chapter, we categorise the IRs mentioned according to this taxonomy, and use this information to identify IR trends over time.

2.4 Linear IRs

All linear IRs consist of sequences of instructions. However, the format and complexity of these instructions varies. In the early days of compilation many linear IRs were developed as part of commercial or unpublished software, so the exact original specifications are not always available. We summarise these under classical representations.

2.4.1 Classical representations

One of the earliest forms of linear IR was based on **Polish notation**. This was originally developed as a parenthesis-free mathematical notation [54] and exists in prefix and postfix forms. This notation was used in a number of early compilers. The expression

```
(1 + 2) * 3
```

can be represented in prefix notation in the following way:

```
* + 1 2 3
```

Alternatively, it can be represented in in postfix notation as:

```
1 2 + 3 *
```

Postfix Polish notation has been used as it is potentially an efficient IR for generating code for a stack-based machine architecture (e.g. Burroughs mainframe computers [22]). To generate code, the IR is simply scanned left to right, with operands being placed on to the stack sequentially and operators being applied immediately to the operands on the stack. Since computer programs contain non-arithmetic operations, **extended Polish** describes any extension of Polish notation that can handle additional operations such as conditional branching, loops, and assignment. All Polish notation statements are referenced by their position in the execution order. Although Polish notation based IRs are compact, they are difficult to optimise. Additionally, most modern processors use register-based, rather than stack-based, architectures. Construction of prefix Polish notation can be achieved by a linear preorder walk of the abstract syntax tree (Section 2.5.1). Construction of postfix Polish notation involves a postorder walk.

Another classical IR is based on **triples** [10]. These instructions have three fields: an operator $\mathtt{op}$ and two arguments $\mathtt{a_1}$ and $\mathtt{a_2}$, represented as $< \mathtt{op}, \mathtt{a_1}, \mathtt{a_2} >$ and also referenced by position. The expression $(1 + 2) * 3$ would be represented by two triples, where the parenthesised numbers in the $\mathtt{a_1}$ or $\mathtt{a_2}$ fields refers to the position of another triple as an operand:

$(0)\ < +, 1, 2 >$
$(1)\ < *, (0), 3 >$

Quadruples extend triples by having four fields: an operator $\mathtt{op}$, two arguments $\mathtt{a_1}$ and $\mathtt{a_2}$ and a result $\mathtt{r}$, represented as $< \mathtt{op}, \mathtt{a_1}, \mathtt{a_2}, \mathtt{r} >$. The result field $\mathtt{r}$ stores the result of the instruction. $(1 + 2) * 3$ would be represented as follows:

$$< +, \mathtt{1}, \mathtt{2}, \mathtt{t_0} >$$
$$< *, \mathtt{t_0}, \mathtt{3}, \mathtt{t_1} >$$

Similarly, **three-address code** (3AC) is a linear IR consisting of a sequence of instructions where there is at most one operator on the right-hand side of an instruction. For example, the expression $(\mathtt{1} + \mathtt{2}) * \mathtt{3}$ would have to be represented by two 3AC instructions as there are two operators. This is shown as:

$$\mathtt{t_0} = \mathtt{1} + \mathtt{2}$$
$$\mathtt{t_1} = \mathtt{t_0} * \mathtt{3}$$

where $\mathtt{t_0}$ and $\mathtt{t_1}$ are temporary variables generated by the compiler. It is possible to represent whole program information using 3AC. Aho et al. [10] specify a 3AC form that supports assignments, operations, jumps, procedure calls, array indexing and address and pointer assignments. Triples, quadruples and 3AC can be generated from a linear inorder walk of the abstract syntax tree.

Many modern compilers use some kind of linear instructions as an IR, either alone or as part of a graph based IR (Section 2.5). For example, the popular open source LLVM compiler [82] uses 3AC written as pseudo-assembly instructions and the Java language uses Java bytecode as an intermediate form fed to the Java virtual machine. Register Transfer Language [48] is a linear form close to assembly language that has appeared in many compilers including GCC. These linear forms support modularity in compiler design, allowing a clean separation between phases. Some compilers may use high level languages, such as C, as IRs also.

2.4.2 Static Single Assignment

The linear IRs above do not explicitly show any dependence information. Many compiler optimisations require knowledge of the data dependencies in a program. Static Single Assignment form (SSA) [47] is a method that transforms linear IR variables to ensure that each is only assigned to once. SSA is not a language, it is a technique that can be applied to a linear IR. This allows for data dependence information to be easily discovered since the each use of a variable points to the exact definition. It does this by transforming each variable V into a variable V_i which only has one assignment. The most recent V_i variable is said to be the most *dominating*, and is always used in a given reference to the variable V. SSA uses *pseudo-assignment* to handle points in the program where control flow merges. If some assignment to V is dependent on a preceding choice in control flow such as an `if` statement then a ϕ-function is

```
a1 = b1 + c1;
x1 = a1 * d1;
if(x1 == y1)
   z1 = e1;
else
   z2 = f1;
z3 = φ(z1,z2);
y1 = z3 + 1;
return y1;
```

(a)

Figure 2.3: Transformation of Figure 2.1 into SSA.

used. This function will create a new definition of V depending on the control flow path taken. We transform our sample code from Figure 2.1 into SSA in Figure 2.3. Here, as `z` is assigned to in the `if` statement, a ϕ-function generates z_3 for the use in the following assignment to y_1.

SSA explicitly shows the data dependence relationship between uses of a variable, which neatly shows def-use chains. Since SSA is in *single assignment* form, there cannot be any redefinition of a variable, hence the def-use chain is explicit by observation. Converting to SSA form makes various optimisations easier and more powerful, such as global value numbering [98] and constant propagation [124]. SSA can be used in conjunction with any other IR containing linear statements.

Construction of SSA form involves two steps: ϕ-functions being inserted at join nodes in the control flow graph (Section 2.5.3), and new variables V_i being generated. Cytron et al. [47] show that SSA can be constructed in $O(R)$ time where R is the maximum of: N, the number of nodes in the control flow graph, E, the number of edges, A_{orig}, the number of original variable assignments and M_{orig}, the number of original mentions of a variable. However, this construction technique can result in unnecessary ϕ-nodes being inserted. Bilardi and Pingali [27] presented an algorithm that only computes the *necessary* ϕ-functions and competes with the speed of Cytron et al.

SSA form cannot be directly interpreted in order to generate code. Therefore it must be destructed before compilation can continue. This involves using an algorithm that converts ϕ-functions into appropriately-placed copy instructions. Briggs et al. [30] showed that the original algorithm for SSA destruction produced incorrect results in some situations and presented a new algorithm. Sreedhar et al. [109] produced an algorithm that reduced the number of generated copy instructions. Boissinot et al. [28] revisit the problem of destructing SSA form, improving on speed and memory usage. SSA is a well-studied IR

and is used in a number of mainstream compilers such as GCC[1] and LLVM.

2.4.3 Gated Single Assignment

In SSA form, ϕ-functions are used to identify points where variable definitions converge. However, they cannot be directly interpreted as they do not specify the condition which determines which of the variable definitions to choose. Thus, after SSA has been constructed and used for optimisations, it must be destructed before code generation can begin. Gated Single Assignment [20] replaces ϕ-functions with gating functions. These gating functions are used to represent conditional branches and loops. GSA can be directly interpreted without having to perform any destruction techniques as is necessary in the case of SSA. We take our definition of the gating functions from Tu and Padua [120]:

- The γ function explicitly represents the condition which determines which ϕ value to select. A γ function is of the form $\gamma(P, V_1, V_2)$ where P is a predicate, and V_1 and V_2 are the values to be selected if the predicate evaluates to true or false respectively. This can be read simply as `if-then-else`.
- The μ function is inserted at loop headers to select the initial and loop carried values. A μ function is of the form $\mu(V_{init}, V_{iter})$, where V_{init} is the initial input value for the loop, and V_{iter} is the iterative input. ϕ-functions at loop headers are replaced with μ functions.
- The η function determines the value of a variable when a loop terminates. A η function is of the form $\eta(P, V_{final})$ where P is a predicate and V_{final} is the definition reaching beyond the loop.

We show our example code in Figure 2.1 translated into GSA in Figure 2.4. Here, the variables are subject to the same renaming as in SSA. However, the assignment to z_3 is represented as a γ function rather than a ϕ function. The γ function has a direct reference to the predicate `P` that decides the control choice in the `if` statement, along with the choice between z_1 and z_2 that exists in SSA form.

Construction of GSA is used as an intermediate step in the construction of the Program Dependence Web (Section 2.5.8) where an SSA form Program Dependence Graph (PDG) (Section 2.5.7) is translated to a GSA form PDG. This step takes $O(VN^2)$ operations where V is the number of variables in the program and N is the number of nodes in the PDG's control dependence graph. Interpreting a GSA in this situation is not discussed in detail.

[1] `http://gcc.gnu.org`

```
a1 = b1 + c1;
x1 = a1 * d1;
P = (x1 == y1)
if(P)
   z1 = e1;
else
   z2 = f1;
z3 = γ(P, z1, z2);
y1 = z3 + 1;
return y1;
```

(a)

Figure 2.4: Transformation of Figure 2.1 into GSA.

Thinned-GSA [62], a more compact version of GSA, has been used to perform value numbering. Construction of Thinned-GSA is shown to take linear time from SSA form. Destruction is not discussed.

2.5 Graphical IRs

Graphical IRs use nodes and edges to represent a variety of different relationships within a program. Some of the earliest recorded graphical IRs are trees, directed acyclic graphs and flowgraphs.

2.5.1 Trees

After lexical analysis and during syntax analysis of the source program, a compiler will commonly generate some form of tree which represents the syntactic structure of the program. There are generally two types of tree which may be constructed: the *abstract syntax tree* (AST) and the *parse tree*. ASTs differ from parse trees as the interior nodes represent only the essential programming constructs rather than non-terminals in the grammar for the input language. Given some expression, each AST interior node represents an operator, and the children of that node represent the operands of that expression. For example, assume the following expression language grammar in EBNF:

```
E = T {"+" T} | T.
T = P {"*" P} | P.
P = a | b | c.
```

Given the sentence `a * b + c` derived from this grammar, we can show the AST and parse tree in Figure 2.5. Notice that the AST only contains the

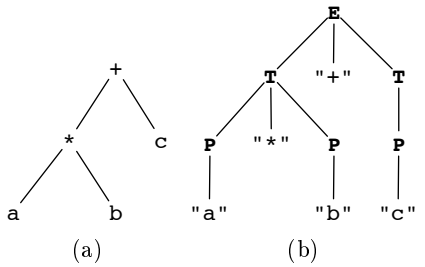


Figure 2.5: The sentence `a * b + c` represented as two different tree types.

essential information (the `+` and `*` operators and variable names), whereas the parse tree contains all of the non-terminals used in the parse. Syntax tree or parse tree construction is straightforward, especially when a parser is written in a top-down recursive manner. Here, code for creating and annotating the tree can be placed in the actions of the recognising methods. Code generation is possible directly from the syntax tree, however, optimisation is more difficult than with other IRs since the compiler may need to access data at various points of the tree at any given time, making for complicated tree walking algorithms. In practice, syntax trees are often used for type checking and semantic analysis, then flattened into a different IR such as 3AC or a directed acyclic graph before continuing with compilation. Flattening refers to the action of translating a tree structure into linear code. Typically construction and flattening of the AST or parse tree are linear processes, except when backtracking parsers are used.

2.5.2 Directed Acyclic Graphs

As seen previously, an AST is a structure that has a close correspondence to the input program. However, this means that there may be redundant computations within it, such as multiple copies of particular expressions. If code is generated naïvely from a tree with redundant computations, the resulting code after flattening will contain unnecessary instructions. A directed acyclic graph (DAG) avoids this duplication by allowing nodes to have multiple parent nodes. This allows identical subtrees in the graph to be reused. As well as making the DAG more compact than the corresponding AST resulting in less memory usage, it means that the compiler can generate code that evaluates the subtree once and then uses the result multiple times.

For example, consider the expression $(\mathtt{a} + \mathtt{b}) * (\mathtt{b} + \mathtt{a}) * (\mathtt{c} + \mathtt{d})$. Here, the subexpressions $\mathtt{a} + \mathtt{b}$ and $\mathtt{b} + \mathtt{a}$ are equivalent due to the + operator being commutative, even though they are syntactically different. Figure 2.6 shows the DAG for this expression.

A DAG can be constructed instead of a syntax tree if, when creating a new

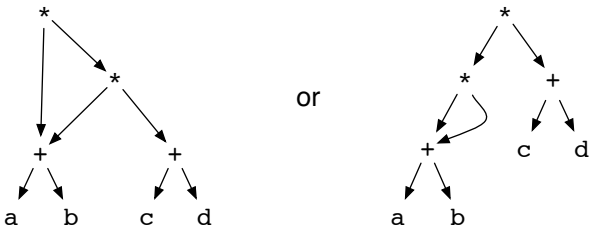


Figure 2.6: DAGs for the expression $(\texttt{a} + \texttt{b}) * (\texttt{b} + \texttt{a}) * (\texttt{c} + \texttt{d})$, showing both left-associativity and right-associativity.

node, the parser checks whether an identical node exists. If it does, then edges are connected from this instead. Alternatively, a syntax tree can be translated into a DAG using a value numbering method [10]. This technique runs close to linear time when using a hash table to record syntax tree nodes along with their associated numbers. Flattening the DAG into a linear IR requires a linear walk as per syntax trees.

The DAG is used as the IR in the lcc compiler [58, 59]. lcc generates only the necessary fragments of the DAG as it parses the program, processes them, then deletes them before continuing. Some compilers use the DAG as a method of improving the existing IR. This is achieved by building the DAG to expose potential redundancies in the code, then transforming the existing IR accordingly. Afterwards it is discarded [118].

2.5.3 Control Flow Graph

The control flow graph (CFG) [12] is a directed graph $G = (V, E)$ consisting of nodes V and edges E, with two nodes `entry` and `exit` in V where all control flow enters and exits the graph respectively. Nodes are commonly called *basic blocks* and contain instructions. Edges show possible paths of execution. A CFG is therefore a representation of the control flow structure in the program. When control enters a basic block it does so at the first instruction and can only leave through the last instruction. Any jump or branching instruction may only appear at the end of a basic block. An edge (a, b) indicates that control may pass from a to b once the last instruction in a has executed. In the CFG the compiler has usually translated instructions from the input program into a simple linear IR such as 3AC. Figure 2.8 shows the example code with a loop in Figure 2.7 represented as a CFG. Edges are drawn dotted to show similarities with control flow edges in other IRs presented later. For brevity, we have represented the conditional test with a `?` suffix. The edges labelled **T** and **F** represent the path taken when the condition evaluates to true or false, respectively. The CFG has a total ordering of instructions, which has usually been enforced by the order in which the programmer (or machine)

```
int main(int a, int b) {
    a = b + 1;
    while(a < 100)
       a++;
    return a;
}
```

(a)

Figure 2.7: Some example code containing a loop.

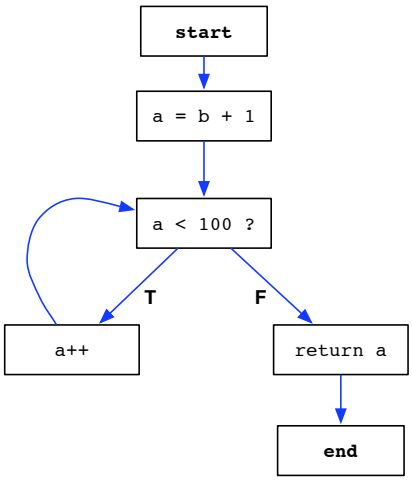


Figure 2.8: The CFG representation of Figure 2.7.

wrote them in the input program. A CFG represents a single function. For inter-procedural control flow to be described, a separate structure called a *call graph* is often used. This is a directed graph with nodes representing functions, and an edge (p, q) exists if function p can call function q.

CFG construction usually occurs from a linear list of instructions such as 3AC, or it can occur from the AST. Both of these methods take close to linear time. Code can be generated directly from the CFG due to its structure and simplicity. The CFG allows a wide variety of optimisations and transformations can be performed. It is widely used in the literature and in mainstream compilers.

2.5.4 Superblocks

The superblock is an IR developed to yield high instruction-level parallelism (ILP) on superscalar and VLIW processors. Within the basic blocks of a CFG there is a limited amount of ILP as each instruction follows sequentially. Superblocks allow ILP optimisation over the existing basic block boundaries. In order to generate superblocks, a CFG is statically analysed so that a numerical value is associated with each basic block representing the instruction frequency of that block. This then separates groups of basic blocks into *traces* which represent common paths of execution. Each trace is then combined into a superblock on which optimisation is performed. Optimisations include enlarging operations, which increase the size of superblocks so the scheduler can manage larger numbers of instructions, and dependence removing operations, which eliminate data dependencies between instructions in frequently executed superblocks, increasing the ILP. Superblocks were implemented in the IMPACT-1 compiler, and benchmark tests showed a 13% to 143% increase in ILP compared to existing techniques [68].

Branch-heavy code can decrease the effectiveness of superblock optimisations because the probability of executing any given path is reduced. Hyperblocks [85] are constructed by performing if-conversion [15], which is a technique for converting control dependence into data dependence by eliminating branches where possible. If branches are eliminated, increased instructions are available to the scheduler. In tests, hyperblocks are shown to perform better than superblocks for higher issue rate processors.

Construction of superblocks begins with the CFG, and the time efficiency of construction is dependent on the static analysis of the program required beforehand. Destruction is not necessary, as like the CFG, it can be directly executed.

2.5.5 Data Flow Graph

The data flow graph (DFG) [50] is also a directed graph $G = (V, E)$ except edges E now represent the flow of data from the result of one operation to the input of another. An instruction executes once all of its input data values have been consumed. When an instruction executes it produces a new data value which is propagated to other connected instructions. The earliest work on data flow computing is credited to Dennis [49]. We show the first two instructions of Figure 2.1 as a DFG in Figure 2.9. In the diagram, the $+$ and $*$ operations have been annotated with the variable they are being stored into in the original code for clarity. Edges are drawn dashed to show similarities with data flow edges in other IRs presented later.

Whereas the CFG imposes a total ordering on instructions, the DFG has no such concept, nor does the DFG contain whole program information. Thus, target code cannot be generated directly from the DFG. The DFG can be seen

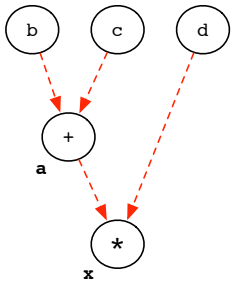


Figure 2.9: The DFG representation of the first two instructions in Figure 2.1.

as a companion to the CFG, and they can be generated alongside each other. With access to both graphs, many optimisations can be performed effectively. However, keeping both the CFG and the DFG updated and synchronised during optimisation can be costly and complicated.

2.5.6 SSA Graph

An SSA Graph [126] consists of vertices which represent operations (such as `add` and `load`) or ϕ-functions, and directed edges connect uses of values to their definitions. The edges to a vertex represent the arguments required for that operation, and the edge from a vertex represents the propagation of that operation's result after it has been computed. This graph is therefore a *demand-based* representation. In order to compute a vertex, we must first *demand* the results of the operands and then perform the operation indicated on that vertex. The SSA Graph can be constructed from a program in SSA form by *explicitly* adding use-definition chains. There are no explicit nodes for variables in the graph. Instead, an operator node can be seen as the "location" of the value stored in a variable.

The textual representation of SSA is much easier for a human to read compared to a graphical form. However, the primary benefit of representing the input program in this form is that the compiler writer is able to apply a wide array of graph-based optimisations by using standard graph traversal and transformation techniques. It is possible to augment the SSA Graph to model memory dependencies. This is achieved by adding additional *state edges* that enforce an order on the sequence of operations reading and writing from memory.

In the literature, the SSA Graph has been used to detect a variety of induction variables in loops [126], also for performing instruction selection techniques [52, 104], operator strength reduction [45], rematerialization [31], and has been combined with an extended SSA language to aid compilation in a parallelizing compiler [113]. The reader should note that the exact specification of what

constitutes an SSA Graph changes from paper to paper. The essence of the IR has been presented here, as each author tends to make small modifications for their particular implementation.

2.5.7 Program Dependence Graph

The Program Dependence Graph (PDG) [57] represents both control and data dependencies together in one graph. The PDG was developed to aid optimisations requiring reordering of instructions and graph rewriting for parallelism, as the strict ordering of the CFG is relaxed and accompanied by the addition of data dependence information. The PDG is a directed graph $G = (V, E)$ where nodes V are statements, predicate expressions or region nodes, and edges E represent either control or data dependencies. Thus, the set of all edges E has two distinct subsets: the control dependence subgraph E_C and the data dependence subgraph E_D. E_C can be cyclic if a loop is present in the program, since a loop in the PDG is defined by a control back edge forming a strongly connected region. E_D is always acyclic, and can be seen as a series of data dependency DAGs for each basic block, which are then connected together based on the data flow through the program. Similar to the CFG, a PDG also has two nodes `ENTRY` and `EXIT`, through which data flow enters and exits the program respectively.

Statement nodes represent instructions in the program. Predicate nodes test a conditional statement and have true and false edges to represent the choice taken on evaluation of the predicate. Region nodes group all nodes with the same control dependencies together, and order them into a hierarchy. If the control dependence for a region node is satisfied, then it follows that all of its children can be executed. Thus, if a region node has three different control-independent statements as immediate children, then these could potentially be executed in parallel. Our example code with a loop is shown as a PDG in Figure 2.10. Rectangular nodes represent statements, diamond nodes predicates, and circular nodes are region nodes. Solid edges represent control dependence, and dashed edges represent data dependence.

Construction of the PDG is tackled in two steps from the CFG: construction of the control dependence subgraph and construction of the data dependence subgraph. Ferrante et al. [57] construct the control dependence subgraph in $O(N^2)$ time. The data dependence subgraph can be constructed after aliasing, procedure calls and side effects are analysed in the program. This involves constructing a DAG for each basic block and then linking them together. Thus the construction of the data dependence subgraph relies on the type of data dependence analysis used. Harrold et al. [61] construct the PDG during parsing. Many algorithms were proposed in the literature for generating code from the PDG [55, 56, 107, 19], but they all were later shown to contain flaws. The only algorithm that claims to be complete and able to handle irreducible programs is that of Steensgaard [112]. Generating the *minimal size* CFG from

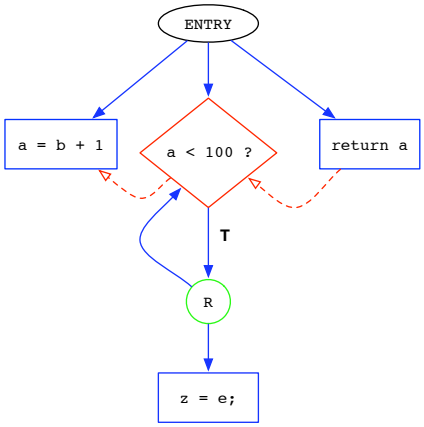


Figure 2.10: The PDG representation of Figure 2.7.

a PDG is an NP-complete problem.

The PDG's structure has been exploited for generating code for vectorisation [24, 103], and has also been used in order to perform accurate program slicing [91] and testing [23].

2.5.8 Program Dependence Web

The Program Dependence Web (PDW) [90] is generated by translating the data dependencies present in the PDG into GSA (Section 2.4.3). Thus, it can be seen as a combination of the PDG and GSA in one IR. The motivation for the development of the PDW is that it can be interpreted under three different execution models: control-, data- and demand-driven. This gives the compiler writer flexibility when developing back-ends for different architectures. Depending on the execution model required, a different *interpretable program graph* (IPG) is extracted from the PDW. The IR was used to compile FORTRAN for data flow architectures, but is limited to programs with reducible control flow. The PDW was later modified [34] to improve the handling of loops.

Constructing the PDW is costly. It requires five passes over the PDG to generate the corresponding PDW resulting in a time complexity of $O(N^3)$. Since the PDG is directly interpreted, no destruction techniques are discussed. In the demand-driven execution, the IPG consists of the data dependence graph augmented with the gating nodes of GSA, and is very similar to the Value Dependence Graph (Section 2.5.9).

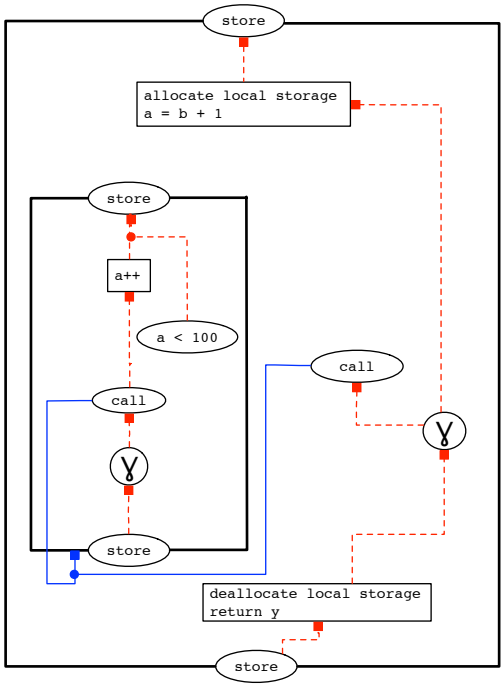


Figure 2.11: The VDG representation of Figure 2.7.

.

2.5.9 Value Dependence Graph

The Value Dependence Graph (VDG) [125] is a sparse, functional data dependence representation, developed to eliminate the CFG as the basis of analysis and transformation. Representing a program as a VDG only specifies the value (data) flow in a program. A VDG is a directed bipartite graph $G = (V, E)$ consisting of nodes V and edges E. Nodes either represent operations, or are *ports* representing operands. Edges connect operation nodes to their operand ports. Each port is produced by exactly one node, or it is not produced by any node (it is a free value). Primitive nodes implement basic operations such as arithmetic and constants. Conditional expressions are implemented by γ nodes which function in the same manner as those in GSA form. Function calls are implemented with a `call` node which takes the name of the function and the function parameters, and produces result ports. Parameter nodes take no operands and produce a parameter value. Function values are produced by λ nodes. Every VDG also contains at least one `return` node. We show the VDG for our example program with a loop in Figure 2.11. Value edges are drawn in dashed. Solid edges link `call` nodes to the node they call.

Implicit machine quantities such as store contents and I/O channels must be explicit in the VDG in order to ensure that operations occur in the correct

order. Loops are translated into tail-recursive function calls. The authors state that optimising a program in VDG form is simpler to implement, easier to express formally, and faster than equivalent CFG analysis. An interesting property of the VDG is that is is implicitly in SSA form: for every operator node, that node will have zero or more successors using its value.

The original construction algorithm for the VDG begins from a CFG, where single-entry single-exit (SESE) analysis is performed. Then, region information is used to decide placement of γ nodes, λ nodes and `call` nodes. Next, calls corresponding to unstructured control flow are consolidated. Next, the intermediate graph is symbolically executed in order to produce the VDG. The running time of this construction algorithm is not discussed. A syntax-directed construction approach is considered by Byers et al. [33], however, it requires a large quantity of post-processing phases to remove redundant nodes.

Weise et al. [125] transform the VDG into a demand-based PDG, where the control flow subgraph is replaced by a demand dependence graph. Then, a PDG sequentialisation technique [112] is used to turn this into a CFG. The VDG was used in an experimental C compiler in order to perform partial redundancy elimination without performing redundant code motion. However, the VDG did suffer from a problem in that *"evaluation of the VDG may terminate even if the original program did not"* [125], making it unsuitable for non-experimental use; the VDG represented no information about interpretation, ordering or termination.

2.5.10 Value State Dependence Graph

The Value State Dependence Graph [72] builds upon the work of the VDG. In order to solve the termination problem with the VDG, the VSDG adds state dependency edges in order to model sequential execution of instructions. A VSDG is a labelled directed graph $G = (N, E_V, E_S, \ell, N_0, N_\infty)$ consisting of nodes N with unique entry node N_0 and exit node N_∞, value-dependency edges $E_V \in N \times N$, and state-dependency edges $E_S \in N \times N$. The labelling function ℓ associates each node with an operator. Value dependency edges E_V perform the same function as those in the VDG. State dependency edges E_S represent the essential sequential dependencies in the input program. An example of this would be enforcing a `store x` instruction before a `load x` instruction, ensuring in no circumstance that this ordering is violated. Like the VDG, the VSDG is explicitly in SSA form. It has two well-formedness conditions. The first is that ℓ and the E_V arity must be consistent. This ensures that a multiplication operator will always have exactly two inputs, and so on. The second is that the VSDG must be acyclic. Nodes in the VSDG represent either operators or constants. Each node has labelled ports in which edges emerge or connect to.

Like the VDG, conditional branches are represented by γ nodes, except these now also return a *state* as well as data values. Loops are represented

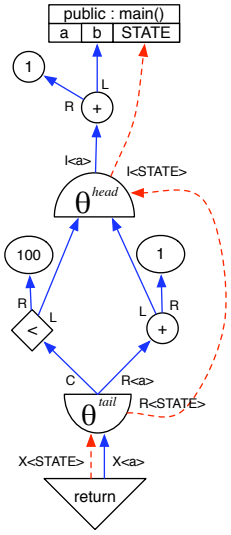


Figure 2.12: The VSDG representation of Figure 2.7.

differently to the VDG. Here, a θ node is used to model loops. A θ node $\theta(C, I, R, L, X)$ sets its internal value to initial value I. Then, while condition value C holds *true*, it sets L to the current internal value and updates the internal value with the repeat value R. When C evaluates to false computation ceases and the internal value is returned through the X port. By default, this node type is cyclic. Therefore, this does not match against one of the VSDG well-formedness conditions. As a result, during compilation, all θ nodes are replaced with two nodes θ^{head} and θ^{tail} which enclose the loop body. This transformation is defined as a VSDG G being translated into VSDG G^{noloop} form. Given a VSDG G, G^{noloop} is defined to be identical to G except that each θ node θ_i is replaced with two nodes, θ_i^{head} and θ_i^{tail}; edges to or from ports I and L of θ_i are redirected to θ_i^{head} and those to or from ports R, X and C are redirected to θ_i^{tail}. We have shown the VSDG for our example code with a loop in Figure 2.12.

Johnson [72] constructs the VSDG directly from the AST. However, this is limited to programs with reducible control flow: the occurrence of `goto` and `switch` statements causes this construction method to halt. Stanier [110] constructs the VSDG after performing an interval analysis technique called structural analysis [106], which allows irreducible control flow to be transformed into reducible control flow. Producing linear code from the VSDG has been explored in a number of ways. Johnson adds *serialising* edges to the graph in order to enforce an order of execution, and inserts split and merge nodes to enable γ nodes to be directly interpreted. However, optimal placement of split nodes was found to be NP-complete [122]. Lawrence [83] presents a framework

that involves translating the VSDG into a PDG by encoding a lazy evaluation strategy, similar to functional programming. This restores *enough* control flow information to continue with code generation.

In addition to being an efficient IR for many traditional optimisations [72], two traditionally antagonistic passes – register allocation and code motion – can be performed at the same time using the VSDG [70]. Also, algorithms to utilise multiple memory access instructions [105] for smaller code size have been developed [71]. A similar representation to the VSDG called the Gated Data Dependence Graph [123] has been described, which again uses γ nodes for conditional choice and the concept of state, but uses a μ and η loop representation similar to GSA. Firm[2] is a data dependence representation also using the concept of state, but the CFG is retained: the graph is built *within* the CFG basic blocks. More recently, Tate et al. [117, 116] used a similar graph, called the Program Evaluation Graph (PEG), in order to optimise by performing equality analysis. This work performs optimisations in different sequences in order to produce multiple versions of the same program, and then picks the best version according to heuristics.

2.5.11 Pegasus

Pegasus [32] is a data-flow oriented IR designed for use in hardware compilation. The IR is a directed graph in which nodes are operations and edges represent value flow. The results of an operation may be used as input to multiple other operations. Data is produced by an operation, transported by an edge and then consumed by another operation. Like the VDG and VSDG, primitive nodes represent constants and complex operations such as memory access and procedure calls have special nodes. Parameter nodes represent the arguments to procedures. Multiplexer nodes perform a similar function to γ nodes, and merge nodes perform the function of μ nodes. The notion of state or store dependence is represented by synchronisation tokens which are passed between operations with non-commuting side effects. This enforces the correct order of execution.

Construction of Pegasus begins from the CFG of each procedure in a program. The CFG is first transformed into hyperblocks and compiled into speculatively executed code in order to extract ILP, with branches transformed into multiplexers. Edges are then added between instructions that may depend on each other, and these are used to carry the synchronisation tokens. Then, data flow edges connect hyperblocks together, along with the insertion of loop back edges and `merge` nodes. The authors note that the construction time complexity is given by the complexity of the component phases; the most complicated of which is the alias analysis used to calculate points-to sets. The construction is then linear based on the size of these resulting sets. Pegasus

[2] `http://www.libfirm.org`

was implemented in the CASH compiler[3] which synthesises hardware circuits from the graph.

2.5.12 Click's IR

Click's IR [38] is a variation of the PDG based on Petri nets [94]. The Petri net model of execution involves control tokens being passed from node to node. Similar to the PDG, the set of edges contains two subgraphs: the control dependence subgraph and the data dependence subgraph. Nodes in the graph represent operations. REGION nodes perform the same function as those in the PDG. PHI nodes model SSA pseudo-assignment, and IF nodes model conditional branching. Loops are implemented with a REGION node at the head, and an IF node at the end of the loop body. A back edge links the TRUE projection from the IF node to the REGION node at the loop head. Construction and destruction of this IR are not discussed in detail. A modified version of this IR is used in the Java HotSpot server compiler [93].

2.5.13 Dependence Flow Graph

The Dependence Flow Graph [95] is an IR designed to be executable and for dependencies to be quickly traversed. Similar to the other data dependence IRs, nodes in the graph represent operations, and edges point from producers to consumers. Value-carrying tokens are passed along edges in the graph in a similar manner to the Petri net model of execution. An imperative updatable global store is used to enforce an order on operations, and load and store operators interact with this store. Operations that are store-dependent are said to have imperative dependence. Loops are represented explicitly with loop and until nodes.

Construction of the Dependence Flow Graph [74] proceeds by performing SESE analysis on the CFG. Then, the variables used in each region are discovered. Then, data dependence edges are inserted in parallel with the CFG control dependence edges to form a base level graph. A forward flow algorithm then traverses the graph and maintains the most recent source for each variable; when a region is bypassed, dependencies are cut. Then, any dead edges from this cutting process are cleaned up. The authors do not discuss the time complexity of this process.

The Dependence Flow Graph was implemented in the Pidgin compiler[4]. The graph is abstractly interpreted according to its operational semantics in order to produce code. A constant propagation algorithm is shown to be simpler to implement but just as effective as existing IRs.

[3] http://www.cs.cmu.edu/~phoenix/compiler.html

[4] http://iss.ices.utexas.edu/p.php

2.6 Classification and comparison

2.6.1 Classification and citations

We now apply the taxonomy of Figure 2.2 to the IRs, resulting in Table 2.1. We used Google Scholar to find the number of citations for the original paper describing each IR in Table 2.2. We did this to get an idea of the academic "importance" of each representation. We also present a timeline in Figure 2.13 which plots the publication years of the original papers shown in Table 2.2. Note that data here cannot be regarded as being precise. For example, although the seminal SSA publication is the 1991 journal article, the idea was in development for a long time at IBM beforehand [127].

Table 2.2 shows that, in academia, the PDG and SSA have been very influential. For such fundamental concepts, the CFG and DFG have a relatively low number of citations in comparison. However, the CFG and DFG are such commonplace IRs that authors often cite compiler textbooks rather than the original papers when referring to them. More recent (post-1990) IRs have a relatively low number of citations.

The timeline in Figure 2.13 shows that from the development of classical IRs through to the PDG, there were few radical new developments. IR literature in the 1970's was dominated by the CFG and the discovery of new analysis and transformations for it. Likewise, the DFG had a similar effect from 1980 onwards. There is a clear clustering of new IR publications from 1987–1995. We can only speculate the exact reason for the increase in new IR developments at that time, however much compiler literature, especially that at the more prestigious compiler conferences, was focusing on vectorisation and parallel computing during this period. This was possibly as a result of the installation of high-performance machines by Cray and other technology companies. The supercomputer era was short-lived, with most specialist supercomputer manufacturers apart from Cray filing for bankruptcy by the mid 1990's. This spike in data flow IRs (and the ILP specific superblocks) could have been as a result of academia trying to develop compilers that could utilise the parallelism of these machines. As noted by Bell [25] this approach to building massively parallel special purpose computers in order to tackle parallel computing wasn't a solution to the problem. It did, however, generate a great deal of academic interest, and most importantly, research money. Academic interest in IRs seems to have lessened over the last ten years, but Pegasus and VSDG-like IRs show that they are still a useful tool for specific compilation purposes.

2.6.2 IR technology in current compilers

We selected a range of widely used compilers and recorded the different IRs being constructed during compilation. The compilers we chose are as follows:

javac This is the principal Java compiler. It compiles Java source code into

IR	Structure	Dependence	Content
Polish notation	Linear	None	Partial
Extended Polish	Linear	None	Full
Triples/Quadruples	Linear	None	Partial
3AC	Linear	None	Full
SSA	Linear	Data	Partial
GSA	Linear	Data	Full
Tree	Graphical (acyclic)	None	Full
DAG	Graphical (acyclic)	Data	Full
CFG	Graphical (cyclic)	Control	Full
Superblocks	Graphical (cyclic)	Control	Full
SSA graph	Graphical (cyclic)	Data	Partial
DFG	Graphical (cyclic)	Data	Partial
PDG	Graphical (cyclic)	Hybrid	Full
PDW	Graphical (cyclic)	Hybrid	Full
VDG	Graphical (cyclic)	Data	Full
VSDG	Graphical (acyclic)	Data	Full
Pegasus	Graphical (cyclic)	Data	Full
Click's IR	Graphical (cyclic)	Hybrid	Full
Dependence Flow Graph	Graphical (cyclic)	Data	Full

Table 2.1: Classification of IRs according to the taxonomy of Figure 2.2.

IR	Paper	Citations
PDG	Ferrante et al. [57]	1659
SSA	Cytron et al. [47]	1576
Superblocks	Hwu et al. [68]	546
DFG	Dennis [50]	507
CFG	Allen [12]	262
PDW	Ottenstein et al. [90]	202
SSA graph	Wolfe [126]	140
VDG	Weise et al. [125]	107
Dependence Flow Graph	Johnson and Pingali [74]	107
Pegasus	Budiu and Goldstein [32]	34
Click's IR	Click and Paleczny [38]	22
VSDG (and similar)	Johnson and Mycroft [70]	21

Table 2.2: Number of citations on Google Scholar (accessed January 2011) for the paper originally describing the IR, ordered by total number.

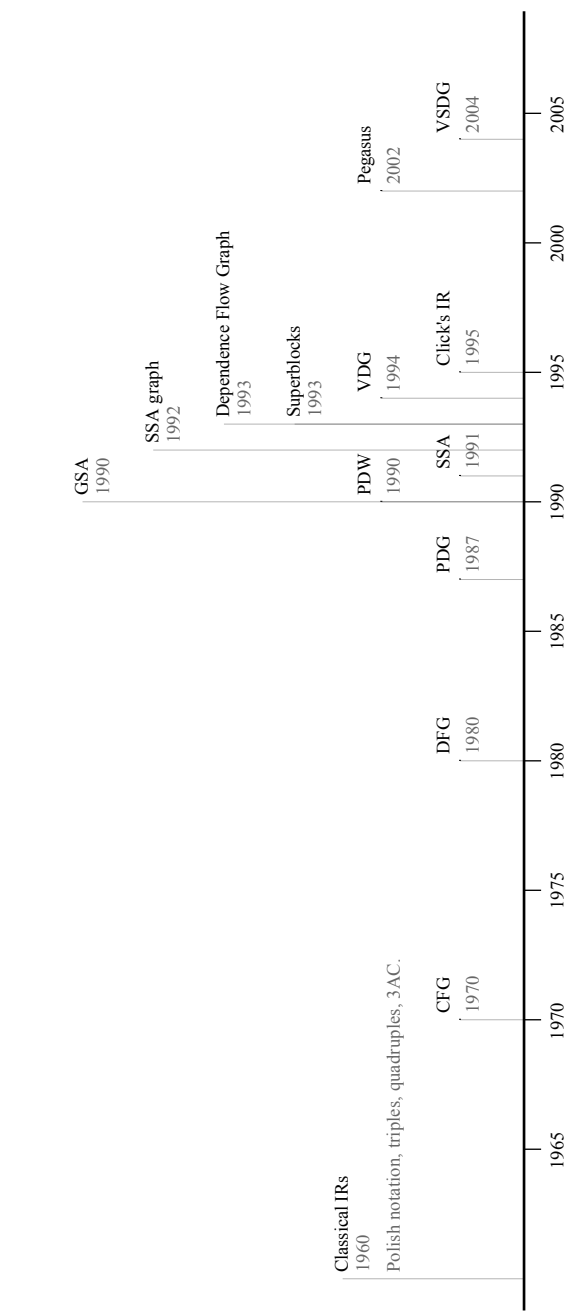


Figure 2.13: Timeline of IRs based on the publication date of the original paper describing it. The starting date of classical IRs has been estimated as no precise information is available.

bytecode, which is then later executed on the Java Virtual Machine (VM).

Java HotSpot server VM This is the principal server Java compiler. It optimises and executes Java bytecode, applying many more aggressive optimisations than the HotSpot client VM.

Jikes RVM This is a mature open source VM for Java, developed from the IBM Jalapeño project. Jikes RVM is a popular choice for implementing Java research projects.

GCC This is the GNU Compiler Collection, which is utilised as the standard compiler on a majority of Unix-based operating systems. It has front-ends for a wide variety of programming languages and targets a large number of processor architectures.

lcc This a simple C compiler written in order to document the process of compiler design [58, 59] and is highly cited.

tcc This is a very compact C compiler often used on embedded devices.

LLVM This is an open source compilation framework which is also very popular for research projects.

Mono This is an open source compiler for C#.

Open64 This is an open source compiler for the Itanium and x86-64 architectures.

These compilers were chosen as access is available to their internal structure, and apart from lcc, are still under frequent development. We show the IRs built by these compilers in Table 2.3. Other popular proprietary compilers such as those provided by Microsoft and Intel do not publish information about their compilation techniques in sufficient detail to be considered for this survey. In addition to the IRs mentioned earlier in the chapter, we record the presence of a multi-level IR system, and the number of levels. This is an explicit compiler design choice involving multiple levels of IR, from high to low. Jikes RVM features three levels of IR: high-level (HIR), low-level (LIR) and machine-specific (MIR). HotSpot and mono feature two levels: HIR and LIR. WHIRL, the IR used in open64, uses 5 levels of IR: very high (VH), high (H), mid (M), low (L) and very low (VL). Using different levels allows optimisations to be written to work on the level of IR that is most suitable for them.

Aside from lcc and tcc, which can be considered special-purpose compilers (i.e. for systems with storage and memory concerns), it can be seen that modern compilers share a common set of IRs:

	DAG	CFG	DFG	SSA	Tree	Multi-level	PDG	Linear	Stack	GSA/PDW	VDG/VSDG-like
javac + Jikes RVM		■	■	■	■	3		■			
javac + HotSpot server VM		■		■		2	■	■			
gcc		■		■	■			■			
lcc	■				■						
tcc									■		
clang/LLVM		■		■	■			■			
mono		■		■	■	2		■			
open64		■		■	■	5		■			

Table 2.3: IR technology used in current compilers.

- A syntax tree in the front-end;
- A linear representation, either for transforming or for outputting the program in a human-readable format;
- SSA for performing data flow optimisations;
- A CFG for control flow optimisations.

Syntax trees and linear representations can be regarded as foundational compiler technology. However, the CFG and SSA representations emerged from academia and have now become standard in mainstream compilers. SSA has generated both academic and real-world interest: it is the second most cited IR in Table 2.2. Interestingly, the most cited IR, the PDG, is only present in the Java HotSpot server compiler which uses an SSA variation of it. Through informal discussion with other compiler researchers, we have yet to note any compiler primarily using the PDG, despite it being the most cited. The reason for this could be similar to the spike in the timeline in Figure 2.13: the PDG was a seminal IR in the supercomputing era. It produced a high number of citations and interest and was most certainly influential in the development of other IRs, however, most mainstream computer users are only just getting affordable access to multicore processors for their home and office machines. Similarly, VDG and VSDG graphs have appeared in a number of influential conferences, yet the technology still remains unused outside of academia. In the short term future, these IRs may well be revisited in attempts to solve the multicore parallelization problems we are increasingly facing in both academic and mainstream compilation.

2.7 Summary

This chapter has explored the existing IR landscape in detail. We began with classical IRs that were implemented in some of the earliest compilers, such as Polish notation and triples, then moved to the early seminal IRs such as the CFG and SSA, and finally looked at the increasing trend towards whole program data flow graph IRs such as the VSDG. We saw that there is a divide between IR technology in academia and in mainstream compilers. The most highly cited IR – the PDG – is only present in one mainstream compiler, despite the technology being nearly 20 years old. Clearly it takes a long time for academic ideas to become fully realised and implemented. We speculate that not only is this because of the vast quantity of time it takes to implement a stable compiler, but academics may never have the required time to make notable open source compiler contributions, and compiler writers may not have the time to continually digest the latest literature. Pressures in academia and industry are very different. However, compiler researchers and programmers must continue to innovate if we are to keep up with ever evolving hardware.

Chapter 3

A Study of Irreducibility in C Programs

3.1 Introduction

In producing target code from a source program, most compilers spend a great deal of time and effort trying to optimise the generated code. This is performed in an attempt to improve the quality of the code according to prescribed criteria (most commonly, speed of execution). These optimisations can range from simple analyses such as deleting unreachable code, to more complicated techniques like code motion. When a program is running it is often the case that a large proportion of execution time is spent iterating in loops. Therefore, aggressive analysis and optimisation of loops is desirable in order to achieve large speed improvements in the generated target code. A wide range of techniques exist for optimising loops; however many can only be reliably used when the control flow graph (CFG) of a program is reducible. In a reducible CFG, all loops have a single entry point that dominates all of the basic blocks in its body. Conversely, an irreducible CFG contains one or more loops with multiple entry points.

Many loop optimisers give up once they have detected irreducible regions of a program, choosing to leave them unoptimised. Alternatively, irreducible CFGs can be made reducible by using a technique called node splitting; however this can lead to an exponential increase in the size of the graph. Many compiler writers who choose to restrict loop optimisations to reducible graphs cite surveys of FORTRAN programs which were undertaken during the 1970's as proof that they are a rare occurrence. We argue that since the landscape of programming has changed dramatically since the original surveys were performed, an up-to-date study of irreducibility is warranted.

Additionally, with respect to this thesis, the VSDG is restricted to representing reducible programs [72]. This is also true of a variety of other IRs, including the Value Dependence Graph [125], Gated Data Dependence Graph

[123] Program Dependence Web [90], and Thinned Gated Single Assignment form [62]. So not only do irreducible programs limit loop optimisations, they also limit the application of a number of compiler technologies in the literature.

This chapter presents the following:

- We outline the concept of irreducibility and give examples of how it can occur in programs.
- Existing methods for dealing with irreducibility are explained.
- We present the results for our study of a large number of human-written C functions to determine how common irreducibility is now compared to the time of the original surveys.
- We then study the output of a number of software tools that generate C code.
- We use this in order to gain an insight into how programmers and programming languages have changed, and posit that compiler designers should not have to worry about dealing with irreducible functions due to their rarity. This questions the need for any new node splitting research.

3.2 Background

In this chapter we are interested in the concept of **reducibility**. The original definition [39, 12] states that a reducible CFG is one on which a technique called interval analysis can be performed. Later work [65] described a notion of "collapsibility" which also determines whether a program is reducible. It can also be defined as whether a particular partitioning of edges can be performed on the graph [66]. We turn to collapsibility for the definition in this study as we feel it is the easiest to understand. A CFG is **reducible** if we can repeatedly perform transformations T_1 and T_2 until the graph has been transformed into a single node. The resulting graph is called the **limit graph**. Assuming we are analysing a CFG G, the transformations are as follows:

T_1 Let G be a CFG. Suppose n is a node in G with a *self-loop*, that is, an edge from n to itself. Transformation T_1 on node n is removal of this self-loop.

T_2 Let n_1 and n_2 be nodes in G such that n_2 has the unique direct ancestor n_1, and n_2 is not the initial node. Then transformation T_2 on node pair (n_1, n_2) is merging nodes n_1 and n_2 into one node, named n_1/n_2, and deleting the unique edge between them [65].

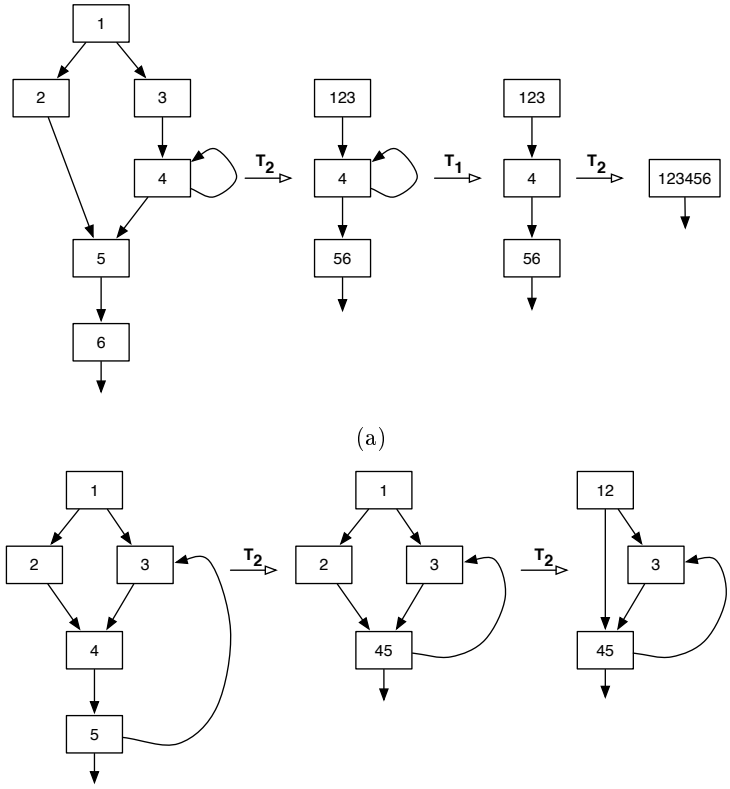


Figure 3.1: Performing T_1 and T_2 transformations on a reducible (a) and irreducible (b) CFG.

These transformations are confluent: the same limit graph will be reached regardless of the order of application. If there is more than one node in the limit graph, the CFG is said to be **irreducible**. Examples of performing T_1 and T_2 transformations can be seen in Figure 3.1. A canonical example of an irreducible flow graph is formed of three nodes and is shown in Figure 3.2. In this example we can see that we cannot perform T_1 as there are no nodes with a self-loop. We cannot perform T_2 as nodes b and c do not have a unique predecessor. Various algorithms exist for detecting whether a CFG is reducible. Hopcroft and Ullman [67] use the transformations above to detect reducibility in $O(E \log E)$ time, where E is the number of edges in the graph. Tarjan [114] presents an approach by performing a depth-first search over the graph and using a set-union function to test whether T_1 and T_2 can be performed, running in $O(E \log^* E)$ time, where $\log^* x = min\{i | log^i \leq 1\}$, which compares favourably.

```
bool main(bool x) {
   a:   if(x) goto b; else goto c;
   b:   if(x) goto end; else goto c;
   c:   if(!x) goto b; else goto end;
   end: return x;
}
```

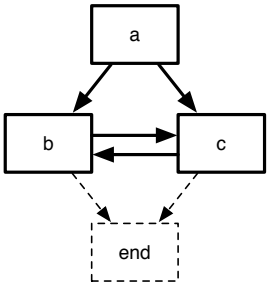


Figure 3.2: Some pseudocode which contains the canonical three-node irreducible CFG.

3.3 Causes and solutions of irreducibility

Generally, there are two culprits for irreducibility. The first is the programmer, and the second is the compiler. The programmer can create irreducible programs by writing code in an unstructured manner. The compiler can create irreducibility by aggressively optimising the program [43], causing difficulty for decompilation techniques [76, 44] which try to bring high-level structure back from assembly code.

One of the most famous debates in computer science stemmed from a letter [51] criticising the `goto` statement as being a cause for poor quality code. The original letter provoked some lively debate [17, 18]. This argument is seen as inspiring the move towards structured programming, where reliance on the `goto` statement is lowered or removed altogether, replaced by structured constructs such as `for` and `while` loops and `if...then...else` constructs. Much of the debate surrounding the `goto` statement centres around the fact that it makes programs difficult for programmers to understand. However, another important aspect of the `goto` argument is that unstructured use can produce irreducible loops in the CFG. This can be "harmful" to the compiler writer, who faces some important optimisation design decisions as a result. Hecht and Ullman [65] showed that all `goto`-less programs are reducible. However, this does not mean that all programs containing `goto` statements are *irreducible*. Aho et al. [10] state that many programs containing `goto` statements are often reducible, as programmers think about their code in terms of loops and branches when they are writing it. Since the CFG models control flow, producing an irreducible graph requires the input program to contain `goto` statements that create multiple-entry loops in the graph (Figure 3.2).

A compiler optimisation known to cause irreducibility is tail call elimination. This is a special case of tail call optimisation which transforms tail recursive functions into iterative ones [87]. By doing so, stack space is saved. This optimisation is important in functional languages where tail recursion

is extensively used, but it is also applicable in all languages where recursive functions can be written. If a function is recursive then tail call elimination replaces the recursive call with a branch to the entry of the function, creating a loop. When combined with inlining there is a possibility of creating multiple loop entry points, thus creating irreducibility.

Deciding not to perform optimisations on irreducible graphs can be costly. Programs usually spend the majority of their execution time iterating around loops. Many compiler optimisations – especially those that analyse and transform loops – fail to work on irreducible CFGs. If the compiler writer chooses to restrict loop optimisations to reducible graphs, then the occurrence of an irreducible loop prevents many optimisation techniques from being used. Many classical algorithms suffer this fate, such as global common subexpression elimination [39], computing dominators [9] and loop invariant code motion [10]. Havlak [63] showed that the presence of one irreducible loop can prevent the improvement of all loops in a procedure. Additionally, a number of intermediate representations in the literature cannot be built when the CFG is irreducible – such as the Value Dependence Graph [125] and variations thereof [72, 123], Program Dependence Graph [57] and Thinned Gated Single Assignment form [62] – which in turn means that any optimisation benefit through using them is impeded.

Two solutions have been proposed for dealing with irreducibility: improving algorithms so that they work on irreducible graphs at the cost of added complexity, or transforming the graph so that it becomes reducible at the cost of compilation time and a potentially exponential increase in code size [35]. These transformation techniques can be applied in the front-end of the compiler or on the intermediate representation. In the front-end, Erosa and Hendren [53] devise a method for detecting and eliminating `goto` statements on the abstract syntax tree. Ammarguellat [16] performs an aggressive normalisation technique on the input language which eliminates the need for optimisations on the CFG and removes any irreducibility. These front-end techniques are less common and less input language independent than CFG-based transformations which take the form of node splitting [64].

We can define node splitting as another transformation alongside T_1 and T_2. As described earlier, the continued application of T_1 and T_2 to an irreducible graph will result in a limit graph with a more than one node. When this point is reached, we perform a node splitting transformation T_3 of which our chosen definition is given by Unger and Mueller [121], citing Hecht [64]:

T_3 Choose any node with at least two predecessors. Duplicate this node so that there is one copy from each of them. Each of the predecessors is now connected to one of the copies, and all of the outgoing edges of the original node are duplicated for each copy.

Once T_3 has completed, T_1 and T_2 can be used again, and the process con-

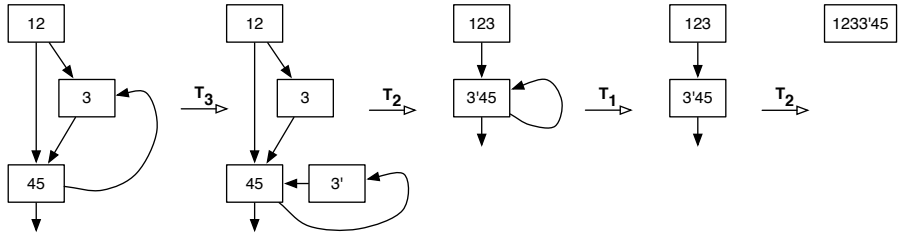


Figure 3.3: The application of node splitting to the irreducible CFG in Figure 3.1b.

tinues until the limit graph has only one node and the graph is reducible. Figure 3.3 shows the application of node splitting followed by the continuation of T_1 and T_2 transformations on the irreducible CFG from Figure 3.1b. The biggest challenge for node splitting algorithms is minimising the increase in code size as a result of duplicating nodes. Several algorithms for minimising size increase have been developed. Janssen and Corporaal [69] present an algorithm called Controlled Node Splitting (CNS), which is tested against the original T_3 method described above, and an optimal technique which is very computationally expensive. Benchmarks indicate that CNS is more effective at reducing increase in code size than the original T_3 method, with an average code size increase of 30.1% compared to 235.5%. Unger and Mueller [121] present an optimised node splitting algorithm using DJ-Graphs to detect irreducible loop structures. Optimised node splitting performed better than traditional T_3 node splitting, with 35% reduction in code size increase.

3.4 Modern languages, old languages

Programming language designers have the power to exclude `goto` statements from whichever language they create. The Java programming language includes `goto` as a reserved word, but it is left unimplemented. Java, along with other recently introduced languages have decided to not implement the `goto` statement, such as Python, JavaScript and Ruby. However, other popular languages include the `goto` statement. Java, C and C++ are the top three programming languages in the table of popularities in July 2009 as calculated by the TIOBE Software Programming Community Index [8] (Figure 3.4). Java has a popularity index of 20%, C 17% and C++ 10%. We can therefore infer that a large proportion of software is currently being written in C and C++. Both of these languages allow the use of `goto` statements, so programmers are potentially able to create irreducible programs. It is also possible for `switch` statements to cause irreducibility in C and C++, with the most famous example being Duff's Device, which was created in order to manually unroll loops.

Pos. Jul 2009	Pos. Jul 2008	Pos. δ	Language	Ratings Jul 2009	δ Jul 2008
1	1	–	Java	20.452%	-0.89%
2	2	–	C	17.319%	+1.37%
3	3	–	C++	10.419%	-0.27%

Figure 3.4: Top three languages in the TIOBE Programming Community Index for July 2009.

It interlaces a `switch` statement with a `while` loop and acts as a computed `goto`[1]. Java does not allow this interlacing of constructs. It is worth mentioning, however, that Java allows the use of labelled statements in combination with `break` and `continue`, allowing control to return to the enclosing labelled statement. These are more well-behaved than conventional `goto` statements, and will not cause irreducibility.

Two surveys are often cited in texts to prove that irreducibility is rare. Knuth [79] sampled a total of 33 FORTRAN 66 programs which spanned over 20,000 punch cards, which is roughly equivalent to the same number of lines of code. Static analysis was performed in order to count keywords and programming constructs. A form of run-time profiling was then performed to see how these were used when the programs executed. A random sample of 50 programs and subroutines were selected, and an interval analysis technique was applied to the CFG of each. It was discovered that every CFG was reduced to a single node limit graph, thus all were reducible. However, it is also stated that on average only 2.75 transformations were required per program, and the highest number of transformations required was 6. Cocke and Allen [13] performed interval analysis techniques on 72 randomly selected FORTRAN IV programs which were currently running in the T. J. Watson Research Center. The task being performed by these programs was not given. Five of these programs were irreducible. An average of 2.85 transformations were needed to reduce each program to its limit graph, a figure similar to that of Knuth's average. The maximum number of transformations required was 9. Even though these surveys constitute the primary citations for irreducibility studies in real-world programs, we argue that they no longer represent an accurate picture of programming. Current languages contain a richer set of features for the programmer to use and, most importantly, software projects are much bigger and more complicated than the FORTRAN programs tested.

Other large-scale empirical surveys of programming languages have been performed, including Java [40], COBOL [101, 36], APL [99, 100] and Pascal [42]. These surveys do not mention reducibility. Instead, they focus on discovering how programmers are using the language by measuring keyword and language construct frequencies.

[1] More information available from `http://foldoc.org/Duff's+device`.

3.5 Method

Since the surveys mentioned in the last section were carried out, the popularity of FORTRAN has dropped and so for this study we turned to C. According to the TIOBE index, C still ranks significantly higher in popularity than C++. Many core system utilities and tools are written in this language, as well as a lot of embedded code and a majority of large open source projects. In this section we present our methodology for performing our study of irreducibility.

The technique that we use to determine irreducibility in a program is called **structural analysis**. Structural analysis is a fine-grain form of interval analysis which is performed on the CFG. The technique was first presented by Sharir [106] as an extension of interval analysis. The aim of the algorithm is to analyse the CFG and recognise particular control flow patterns.An improper region is detected when the CFG is irreducible. It is characterised as a strongly-connected component with multiple entries. Structural analysis first constructs a depth-first spanning tree for the CFG. It then examines basic blocks in postorder, recognising regions and collapsing them into abstract nodes. This continues until the limit graph is reached. As with T_1 and T_2 analysis, a limit graph with more than one node is irreducible. In parallel, the algorithm constructs a control tree which represents the hierarchical structure of the regions within the original graph. In other work, structural analysis has been used to approximate the worst-case execution time of programs [46] and to perform thread partitioning [21]. We delay a detailed description of the structural analysis algorithm until Chapter 4, where we will be using it to construct the VSDG.

We have implemented the updated structural analysis algorithm given by Muchnick [87] as an optimisation pass inside the LLVM compiler framework [82]. If the algorithm completes and the limit graph is one node, then the CFG is reducible. If we detect an improper region at any stage of the algorithm the program is deemed irreducible.

We log the number of reductions needed to turn the CFG into a limit graph to give a rough measure of program complexity in the style of Knuth [79] and Cocke and Allen [13]. We also log the locations (via line number) of `goto` statements found in the source file for ease of reference, and to see how common their usage is. There is a slight discrepancy between our reductions and that of the previous surveys. T_1 and T_2 reductions result in the collapsing of 1 or 2 blocks respectively. In structural analysis, a reduction can collapse anywhere from 1 to 3 blocks in the fixed-size cases, and potentially many more in the catch-all proper region schema. This means that the number of reductions we measure in our framework would be higher in a T_1 and T_2 based framework as less blocks are being reduced each time a transformation is performed. Additionally, Allen and Cocke measure the number of iterations through their interval analyser as opposed to the number of T_1 and T_2 transformations performed. However, Knuth specifically measures the number of

T_1 and T_2 transformations. Clearly this means that the three surveys do not have directly comparable reduction data, however, we are only interested in a rough measure of program complexity since irreducibility is the focus of this survey.

3.6 Results

We first tested the irreducibility of the source of a set of open source projects written in C. We chose the source code of 15 GNU projects as our sample data. These represent non-trivial, regularly updated software with an active community of developers. They have also been in widespread use since their conception. We began by performing structural analysis on each C file in the source code. The only optimisation that was performed on the graph beforehand was LLVM's `-simplifycfg` pass, which performs dead code elimination and some simple basic block merging. The results are given in Figure 3.5. In total, we found 5 irreducible functions in a total of 10427, giving a total average irreducibility for this set of current programs of 0.048%. We see from the results that all irreducibility occurs only when `goto`s are used, agreeing with a finding from Hecht and Ullman [65]. It can be seen that a large number of `goto` statements are still used in most projects. After obtaining results for the current versions of the software, we ran the same tests on the oldest versions available to us. These results are shown in Figure 3.6. Here we found 20 irreducible functions in a total of 4772, giving an average irreducibility of 0.42% for older versions of the same software. This represents a statistically (chi-squared) highly significant difference ($P < 0.0001$). While irreducibility is still a rare occurrence, it seems to be nearly 10 times more likely in the software written between 9 to 15 years ago than in the C software of today. Importantly, the 5 irreducible functions in the latest versions are all present in the older versions. No new irreducible functions have been introduced.

In the survey carried out by Knuth [79] it is stated that the average number of transformations needed to produce a limit graph in 50 randomly selected functions is 2.75. In the Cocke and Allen [13] survey this average was 2.85 for a total of 72 functions. From our results, we see that for current versions of the software the average number of transformations is 4.250, and for old versions it is 5.286. By comparing with these original surveys, we can see that functions from 1993-2007 are more complicated than those from the 1970's, although this is unsurprising. Knuth found that the largest number of transformations needed to produce a limit graph was 6, and Allen and Cocke's maximum number of interval analyser iterations was 9. For current versions of software, the largest number of transformations required was 270. In older versions, the largest number of transformations was 193.

Interestingly, the current versions have a lower average number of transformations than the older versions of the same software. While this decrease

Name & version	Released	Number of C files	Number of functions	Reductions (average)	`gotos`	Irreducible functions
binutils-2.19.1	2009	42	828	4.127	216	0
bison-2.4.1	2008	31	538	5.043	63	0
coreutils-7.4	2009	108	1508	3.442	86	0
findutils-4.4.2	2009	8	321	2.059	8	0
grep-2.5.4	2007	9	232	5.263	110	0
guile-1.8.7	2009	104	4517	2.314	756	3
gzip-1.3.12	2007	32	141	6.645	5	0
idutils-4.2	2006	5	98	4.439	3	0
indent-2.2.10	2009	11	119	4.610	8	1
less-418	2009	35	445	3.591	3	1
m4-1.4.13	2009	11	201	4.846	0	0
make-3.81	2006	24	278	5.155	68	0
readline-6.0	2009	39	600	3.025	252	0
sed-4.2.1	2009	7	121	3.877	4	0
wget-1.11	2008	31	480	5.313	127	0
		497	**10427**	**4.250**	**1709**	**5**

Figure 3.5: Results for current (as of July 2009) versions of open source software.

Name & version	Released	Number of C files	Number of functions	Reductions (average)	`gotos`	Irreducible functions
binutils-2.7	1996	32	617	4.263	90	1
bison-1.25	1995	22	172	7.209	2	0
coreutils-5.0	1996	91	932	4.548	69	0
findutils-4.1	1994	7	147	1.925	0	0
grep-2.0	1996	9	133	6.707	148	0
guile-1.0	1997	72	1042	3.731	542	15
gzip-1.2.4	1993	20	110	6.755	2	0
idutils-3.2	1996	6	104	4.413	6	0
indent-1.9.1	1994	9	63	7.066	20	2
less-290	1995	32	335	3.042	22	1
m4-1.4	1994	20	227	6.053	72	0
make-3.75	1996	21	169	7.917	42	1
readline-2.0	1994	15	347	3.513	14	0
sed-1.18	1993	6	94	7.234	75	0
wget-1.5.3	1998	25	280	4.911	25	0
		387	**4772**	**5.286**	**1129**	**20**

Figure 3.6: Results for the oldest available versions of the software in Figure 3.5.

of 1.036 is not a large enough figure to conclude anything concretely, we can speculate that alongside the reduction in irreducibility through time, recent programming is becoming more structured and modular; i.e. large unwieldy functions are being broken down into several smaller ones. By visually inspecting the source code for both old and new programs, it can be seen that there has been a significant amount of restructuring on many of the projects. Usage of `goto` statements per function has also declined in the current versions. Here, the average number is 0.16 compared to 0.23 in the old versions. This is consistent with the claim that programming is becoming more structured.

Since tail call elimination is an optimisation that can cause irreducibility when replacing recursion with iteration, we tested all software after performing LLVM's tail call elimination pass with an inlining pass. We found that there was no irreducibility added by these optimisations to any of the graphs.

3.7 Patterns of irreducibility

We visually inspected the functions which were flagged as irreducible as we were curious as to whether there were any recurring patterns in the source code. In this section we will explore these and suggest how a similar functionality could be achieved by using more structured programming.

We discovered that the style of source code producing irreducibility is not far removed from the style of the canonical example given in Figure 3.2. Commonly, an irreducible function is partitioned into different sections by using labels. Each label contains code performing some specific task. `goto` statements transfer control between these sections in such a way that an irreducible CFG is produced. A good example of this behaviour can be seen in irreducible functions that are parsing text, such as `print_comment()` from indent-1.9.1 in the file `comments.c`. Here, the code is iterating over characters representing a comment in code, and then using `goto` to jump to different sections depending on the character that has been read. This allows a different action to be taken when reading the beginning and end of a line. Similar behaviour can be seen in another irreducible text processing function, `fch_get()` from less-290 in `ch.c`. We give a simplified example in Figure 3.7. Here we see three labelled sections representing different tasks, and control flow jumps between them. A possible solution is to translate each labelled task as a separate function, and then use function calls instead of `goto`s, as this resembles the original presentation of the code.

Further examples of this behaviour can be seen in four functions in guile-1.0, in the file `unif.c`. These functions are:

- `scm_vector_set_length_x()`
- `scm_array_prototype()`
- `scm_uniform_array_read_x()`

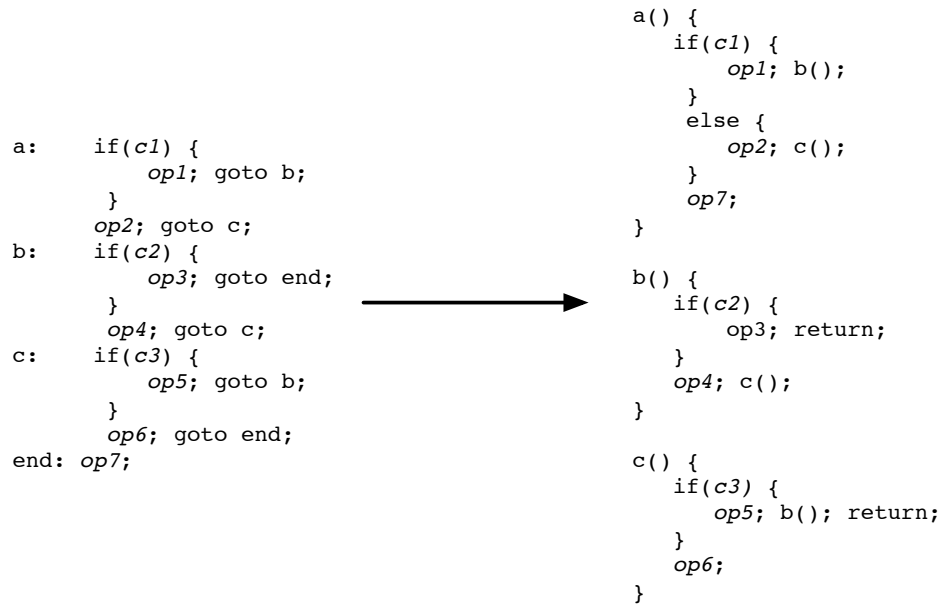


Figure 3.7: Unstructured labelled sections of code and the equivalent structured code. op_n represents non-branching code and c_n represents Boolean variables.

- `scm_uniform_array_write()`

Here, after a number of macros have been expanded by the compiler, an labelled loop performs processing that has been organised into labelled sections. These sections contain `goto`s which jump to different labelled sections representing different stages of this processing. We see no difficulty in a programmer refactoring this code in a structured manner.

It may be possible to automatically detect irreducible patterns in source code during compilation by pattern matching and then emit a diagnostic warning to the programmer. Alternatively, a diagnostic warning could be emitted during a CFG analysis phase. This would allow the programmer to reconsider the way they have programmed a particular piece of code and to adjust it accordingly, thus giving the compiler more chance to optimise it.

Most, if not all, of the irreducible functions were composed of the pattern above. Ideally, all iterative behaviour should be handled by structured looping constructs or by function calls.

3.8 Machine-generated irreducibility

C code is not always written by humans. Many software tools can generate C code as output. Machine-generated C code may have a very different style to that which is written by humans, especially as a software tool is not necessarily generating code with readability in mind. Often, machine-generated C exhibits heavy `goto` usage with the resulting output looking much more like low-level assembly code than structured C. In this section we test machine-generated C code for irreducibility.

3.8.1 Parser generators

Compiler writers and language designers often use parser generators. These tools simplify front-end compiler construction by reading a language specification and then generating a lexical analyser and parser for that language. Two commonly used lexical analyser generators are `lex`, and in a more recent implementation, `flex`. These tools transform a specification of the tokens in input language specification into a transition diagram and then generate a lexical analyser in C. These lexical analysis tools are often used in conjunction with two commonly used parser generators, `yacc` and more recently `bison`, which interface with the lexical tokens returned by `lex` or `flex` in order to perform syntax analysis. From a specification of the input language, an LALR parser is generated. Used together, these can form the front-end of a compiler for that language.

We obtained language specifications for a number of non-trivial programming languages via the `comp.compilers` newsgroup. Firstly, we generated a parser in C for these languages using `lex` and `yacc`. The results of running our analysis on these C files is shown in Figure 3.8. Then, we generated parsers using the more recent `flex` and `bison` tools and ran our analysis on these C files. The results for this are shown in Figure 3.9. The grammar for Fortran made use of a handwritten lexical analyser that interfaced with `yacc` or `bison` due to the complexities present in the lexical analysis of Fortran.

No irreducible functions were detected in the code produced by these tools.

3.8.2 Source-to-source compilation

It is also possible to generate C code by performing source-to-source compilation. Here, the compiler reads in C (or another high-level language) as the input language. However instead of generating code for a low-level target, it generates high-level source code. Source-to-source compilation can be achieved in LLVM by compiling to LLVM bitcode as usual, but then using the `llc` tool to generate C as the target language rather than a low-level target such as x86. As we are interested in observing differences between human-written C and machine generated C, we chose to source-to-source compile all

Language grammar	Number of C files	Number of functions	Reductions (average)	`gotos`	Irreducible functions
Ada 9x	2	40	4.300	15	0
C	2	42	4.024	15	0
C++	2	38	4.421	15	0
Fortran	2	10	11.100	1	0
Java 1.1	2	35	4.229	15	0
MATLAB	2	39	4.308	15	0
Modula-2	2	70	3.971	21	0
Pascal	2	40	4.200	6	0
	14	**314**	**5.069**	**103**	**0**

Figure 3.8: Results for parsers generated for 8 programming languages using `lex` and `yacc`.

Language grammar	Number of C files	Number of functions	Reductions (average)	`gotos`	Irreducible functions
Ada 9x	2	40	4.350	6	0
C	2	42	4.071	7	0
C++	2	42	5.381	6	0
Fortran	2	10	11.100	1	0
Java 1.1	2	35	4.286	6	0
MATLAB	2	39	4.359	22	0
Modula-2	2	70	3.986	28	0
Pascal	2	40	4.250	22	0
	14	**318**	**5.220**	**98**	**0**

Figure 3.9: Results for parsers generated for 8 programming languages using `flex` and `bison`.

of the human-written open source software tested earlier in Figure 3.5 into machine-generated C, and then analysed these files. The results can be seen in Figure 3.10. It is worth noting that the `llc` tool was unable to generate C code for 23 C files in guile-1.8.7, and exited with an error. We believe this to be a bug in the tool. However, the functions that exhibited irreducible control flow previously were unaffected by this bug and compiled without error.

Interestingly, no irreducibility was introduced or removed by source-to-source compiling the software in this way. There is a dramatic increase in the number of `gotos`. This is due to the fact that, on inspection, the `llc` tool generates C by making each basic block in the CFG a labelled section of C code, and then using `gotos` to transfer control to the next labelled section(s). Thus, since the structure of the machine-generated source code resembles the original CFG, when it is translated back into a CFG by our analysis pass, the structure of the graph is almost identical.

Name & version	Released	Number of C files	Number of functions	Reductions (average)	`gotos`	Irreducible functions
binutils-2.19.1	2009	42	828	4.605	19062	0
bison-2.4.1	2008	31	537	5.101	7816	0
coreutils-7.4	2009	108	1371	4.125	23668	0
findutils-4.4.2	2009	8	252	2.405	2970	0
grep-2.5.4	2007	9	185	7.281	5368	0
guile-1.8.7	2009	83	1484	3.348	14344	3
gzip-1.3.12	2007	32	135	6.970	2994	0
idutils-4.2	2006	5	89	4.944	1282	0
indent-2.2.10	2009	11	88	5.659	3187	1
less-418	2009	35	406	3.714	6969	1
m4-1.4.13	2009	11	89	4.944	3849	0
make-3.81	2006	24	253	6.055	4920	0
readline-6.0	2009	39	560	3.595	7695	0
sed-4.2.1	2009	7	106	4.292	1796	0
wget-1.11	2008	31	505	5.673	10302	0
		471	**6888**	**4.860**	**97160**	**5**

Figure 3.10: Results for source-to-source compiled versions of the software from Figure 3.5.

3.8.3 MATLAB Real-Time Workshop

Simulink is a tool for designing and simulating dynamic and embedded systems, developed by MathWorks and integrated within the MATLAB environment. The Real-Time Workshop package enables C code to be generated from Simulink models. We selected a number of non-trivial Simulink projects that were available to download from the MATLAB Central File Exchange [6] and then used Real-Time Workshop to generate C code for these simulations. The Simulink projects ranged from signal processing algorithms to neural network simulations, along with a PUMA robot arm controller and inverted pendulum, motor, and suspension simulations. The results from the analysis of these C files is shown in Figure 3.11.

Interestingly, the C code produced by Real-Time Workshop is highly structured and very readable. No `gotos` were present in any of the C files. Therefore, no irreducible functions were present in this code.

3.9 Concluding remarks

We conclude from this study that irreducibility is happening much less frequently than it was 9 to 15 years ago. It is difficult to say exactly *why*, as there are a great number of factors that contribute to why a programmer or software tool could be producing irreducible code. A programmer may naturally think about coding in an unstructured manner, or be adding a quick fix to some existing code by using `gotos` and labels rather than completely

Simulink model	Number of C files	Number of functions	Reductions (average)	`gotos`	Irreducible functions
DC Motor Model	5	22	0.363	0	0
Inverted Pendulum	30	13	0.920	0	0
PID Controlled DC Motor	6	25	1.080	0	0
Timing Recovery	16	55	5.909	0	0
Viterbi Decoding	6	21	3.524	0	0
Suspension System	5	22	0.366	0	0
PUMA Robot Controller	20	87	1.920	0	0
PAM Mod./Demod.	6	22	1.591	0	0
OFDM w/ QPSK Mod.	5	24	1.542	0	0
Cellular Neural Network	5	22	0.455	0	0
	14	**318**	**5.220**	**98**	**0**

Figure 3.11: Results for C code generated with Real-Time Workshop for MATLAB Simulink models.

restructuring a section into a structured loop, amongst many other reasons. However, there has been a clear move towards structured programming over the decades: many popular modern languages such as Java do not include the `goto` statement and rely on purely structured constructs to develop programs. Since this is the case, many newly educated programmers may be thinking differently about how they design their code than those who were taught to program when unstructured techniques were still widely used.

The Internet has greatly increased collaboration between programmers on open source projects, and the complexity and number of open source projects continues to grow. With this also comes an increase in the number of developers working on projects. Being able to write clear code that is readable by other humans is a key skill. Contributors are much more likely to accept code which is neat and readable, and this quality often goes hand-in-hand with structured programming and less irreducibility.

The results presented here should influence both compiler writers and compiler researchers. For those are writing compiler optimisations that may have worried about handling the chance of irreducibility, we say that it is an *extremely* rare occurrence, and it will probably be even rarer in the future. We feel that out of all the possible programs that a compiler can compile, the overall loss of potential optimisation from irreducibility is negligible. Taking into consideration the trend of less irreducibility over time, and the fact that no new irreducible functions were introduced since the old versions of the software we tested, we posit that *leaving irreducible code unoptimised is a feasible future-proof option for compiler writers.* Additionally, a corollary to this statement is that IRs restricted to reducible programs are feasible and future-proof also. This, of course, questions the need for more complicated node splitting techniques. Unless a drastically different approach to transforming any irreducible graph into a reducible one is discovered, we feel that any more node

splitting research will have little impact on real-world compilation, since many compilers do not even perform it in a basic T_3 form. Programmers wishing to write the "best" code for a problem should concentrate on structured algorithm design to give the compiler the best chance for aggressive optimisation.

Future work could analyse source code at more regular time intervals to try and see if a particular time or event coincided with a reduction in irreducible code. It would be interesting to see whether functions rewritten to be reducible were done so primarily for this purpose, or whether it was an unintended side effect of a programmer thinking in a more structured manner. Code written for some specialist purposes may rely heavily on unstructured control flow, but we are currently unaware of any such purpose. We would be interested to see if a compiler with a particularly aggressive inlining phase could repeatedly and predictably transform reducible CFGs into irreducible ones.

3.10 Summary

In this chapter we have performed an empirical study of irreducibility in current versions of open source software, and compared them with older versions of the same software. We also studied machine-generated C code from a number of software tools. We found that irreducibility is rare, and is becoming less common with time. We concluded that leaving irreducible functions unoptimised is a feasible future-proof option due to the rarity of its occurrence in non-trivial software, and as a result of this, IRs restricted to reducible programs are feasible and future-proof also.

Chapter 4

Construction

4.1 Introduction

In this chapter, we examine the problem of constructing the VSDG. There exist a number of different approaches to building the VSDG and related IRs. In the literature, construction proceeds from one of two data structures: the abstract syntax tree (AST) or CFG.

During syntax analysis it is common practice for the AST to be built. One existing approach uses the AST along with the symbol table to generate nodes and edges in the VSDG. Identifying the position of a conditional statement or a loop in the input program is simple: the keyword for the construct will be present in the AST. However, the major drawback to this approach is that it is entirely source language dependent, meaning that the algorithm would have to be rewritten for every input language that the compiler parses. Additionally, this approach is unable to detect irreducible control flow (since it lacks any kind of "bigger picture" of the source program), is restricted to 0-trip loop constructs, and cannot handle unstructured exit from loops.

The alternative—constructing from the CFG—presents different challenges. Whereas the position of conditional statements and loops is explicit in the AST, this must now be rediscovered from patterns in control flow. In the literature, this has been achieved by analysing the "shape" of the CFG to discover control flow patterns, and then using this information to guide the placement of branching and looping constructs in the IR. These techniques are all restricted to input programs with reducible control flow, and require separate node splitting passes to transform irreducible programs into reducible ones.

We posit that previous construction approaches lack a unifying technique that remains language independent; handles unstructured control flow, and allows the opportunity to detect and deal with irreducibility. This chapter presents an approach to constructing VSDGs that addresses these issues by:

1. Finding patterns in the CFG and handling irreducibility using an existing

technique called *structural analysis*;

2. Using a novel construction algorithm that *generates* a VSDG fragment for each basic block in the CFG, and then *merges* them together using the syntactic information discovered during the previous phase.

We show with a worked example how our approach can construct a VSDG from the CFG in SSA form of an input program. We believe that this framework provides the construction stage with more syntactic detail about the input program than previous approaches, enables a convenient way to deal with irreducibility, and neatly deals with unstructured control flow. Additionally, we believe that our framework is better documented than previous approaches in the literature.

4.2 Related work

We will begin by exploring the previous ways in which a number of related IRs have been constructed. The IRs that we consider for this section are the VDG, VSDG, DFG and GDDG, which we explored in Chapter 2, since they are all dataflow-based IRs that use γ-nodes for conditional choice and have related looping constructs.

4.2.1 Value State Dependence Graph

Johnson [72] uses a syntax-directed translation of C input programs. The abstract syntax tree (AST) for a program is often generated in the front-end of the compiler during syntax analysis. Top-down recursive parsers can easily generate the tree in the actions of their recognising methods. The AST along with the symbol table can be used to construct the VSDG. Lawrence's work with the VSDG did not present any new method for construction [83].

A stack-based approach is taken, similar to that of Brandis and Mössenböck [29]. To compile expressions, the AST is traversed whilst emitting VSDG nodes and edges. Nodes which assign to variables also update the symbol table information with the current VSDG node for that value. Side-effecting operations produce a new state.

Compiling `if` statements makes use of a private name stack for each value in the symbol table. VSDG node names are temporarily stored on this stack to model the lexical scoping within the source program. Upon entry to an `if` statement, the current node name of each register variable is pushed onto this stack. When exiting the `then` block and entering `else`, for each register variable, the name at the top of the stack is swapped with the current VSDG node name. When exiting the `else` block, the γ node has edges added to the respective nodes from the T and F ports to represent the values returned when the `if` statement evaluates to true or false.

Compiling loops makes use of θ^{head} and θ^{tail} nodes. When detecting the entry to a loop, for each register variable, edges are added from the θ^{head} I port to the source node. Then, all register variables are updated to refer to the θ^{head} L port. Before exiting out of the loop, edges are added from the θ^{tail} R port to the current nodes for those values. On exit, all register variables are updated to refer to the θ^{tail} node X port.

There are two benefits to this method of construction. Firstly, access to keywords in the input program such as `if` and `while` identify the precise position of conditional statements and loops. Secondly, use of simple AST traversal methods guarantees an efficient implementation of the algorithm.

However, the disadvantages far outweigh the advantages. The first is concerned with modularity. Many modern compilers are split into modular units which are as independent from the input and target languages as possible. This allows the development of compilers for new programming languages and architectures to take less time, as just a new front-end or back-end can be written. Writing VSDG construction into the syntax analysis stage ties it to a particular input language; in this case C. Additionally, there exists no way of detecting irreducible control flow. Due to these reasons, Johnson's algorithm quits with an error upon seeing the `goto` or `switch` keyword in the AST [72]. This makes a syntax-directed construction approach unsuitable.

4.2.2 Dependence Flow Graph

Construction of the DFG begins by computing Single-Entry Single-Exit (SESE) regions in the CFG. A **SESE** region in a CFG is an ordered pair (a, b) of distinct edges a and b where a dominates b, b postdominates a, and every cycle containing a also contains b and vice versa. Once SESE regions have been discovered [73], the authors state they use four steps to build the DFG:

1. Determine the variables used within each SESE region by means of an "inside-out" traversal.

2. Create a base-level DFG with no region bypassing by inserting dependence edges in parallel with control flow edges.

3. Perform region bypassing using a forward flow algorithm using the information in Step 1.

4. Remove dead flow edges created during bypassing using a backward propagation algorithm.

Exact details of how to implement this construction algorithm are waived for space concerns [95, 74], and little extra detail on DFG construction is available in the author's thesis [75]. We therefore look elsewhere for examples of construction from the CFG.

4.2.3 Value Dependence Graph

The VDG can be seen as the predecessor of the VSDG, and represents loops by translating them into tail-recursive function calls with λ and `call` nodes. The authors tackle construction of the VDG in four stages:

1. SESE analysis is performed to identify conditional branches, loops and unstructured control flow.

2. A Store Dependence Graph (SDG) is constructed by translating basic blocks into *block* nodes, branching control flow into γ-nodes and looping and unstructured control flow becomes λ and `call` nodes. The authors note that this step is "expository fiction", in that the implementation doesn't actually build the SDG, but merely uses the information to drive the next stage.

3. The SDG is converted into a VDG by performing symbolic execution, where each *block* node is translated into a DDG node and then linked together with subsequent DDG nodes with γ-nodes.

Like the DFG, the authors waive any exact details of the construction algorithm due to space [125]. No further publications were produced on the VDG.

4.2.4 Thinned Gated Single Assignment

TGSA construction proceeds from the CFG. To begin with, an analysis pass discovers control flow patterns that represent loops. Then, the entry and exit of loops are augmented with preheader and postbody nodes. These augmentations are then used to guide the placement of μ- and η-nodes which represent loops in TGSA form.

After this stage has completed, the CFG is then translated into SSA form. The ϕ-functions that are placed at control flow merges serve as a marker for where to place DAGs of γ-nodes. An algorithm traverses the CFG backwards from each ϕ-function along immediate dominators and places a γ-node at each conditional branch that is encountered, using a process the authors call γ-conversion.

Upton [123] states that this construction approach was attempted for the building of the GDDG, but it failed to work for loops with multiple exits.

4.2.5 Gated Data Dependence Graph

As a result of the aforementioned flaws of TGSA construction, building the GDDG incorporates Tarjan's algorithm [114] for building the loop nesting tree, which identifies a unique header node for each loop in a reducible CFG. Once

the loop nesting tree has been built, loop construction proceeds in the manner of TGSA form, augmenting loop entries and exists and placing μ- and η-nodes.

Placement of γ-nodes is achieved without translating the program into SSA form; instead path expressions [115] are used to guide insertion. These path expressions can be used to place γ-nodes in the manner of Tu and Padua [120]. Upton uses this technique to place γ-nodes for every virtual register defined on some path to control flow merges from its immediate dominator. Loop predicates are also built in this way. This is an effective construction approach, however it requires a previous analysis pass to remove irreducibility from the input program.

4.2.6 Our requirements

We therefore state the following requirements of our VSDG construction algorithm:

Correctness Fundamentally, the VSDG should correctly represent the input program by performing the same operations.

Language-independence Johnson's syntax-directed approach with the AST is not independent of the input language. We should operate on the program in a (mostly) machine-independent form, namely the CFG.

Dealing with irreducibility We need to be able to detect irreducible regions in the control flow graph. Ideally, we would like to have the ability to apply node splitting as construction proceeds.

Dealing with unstructured control flow We also need to be able to handle unstructured control flow. As seen in the previous chapter, most real-world C programs feature frequent `goto` usage.

The first part of our construction algorithm aims to detect control flow patterns. We choose to begin construction from the CFG as most modern compilers use it as the primary IR, and it is simple to work with. We take an existing approach to detecting control flow patterns called *structural analysis*.

4.3 Structural analysis

Structural analysis is a fine-grain form of interval analysis on the CFG. It was first presented by Sharir [106] as an extension of existing interval analysis techniques [14]. A detailed pseudocode algorithm is given by Muchnick [87]. We use Muchnick's pseudocode to implement it. The purpose of structural analysis is to make the syntax-directed method of dataflow analysis [97] applicable to a lower-level IR. Thus, this approach unites the different benefits of constructing from the AST and CFG. It also identifies more patterns than

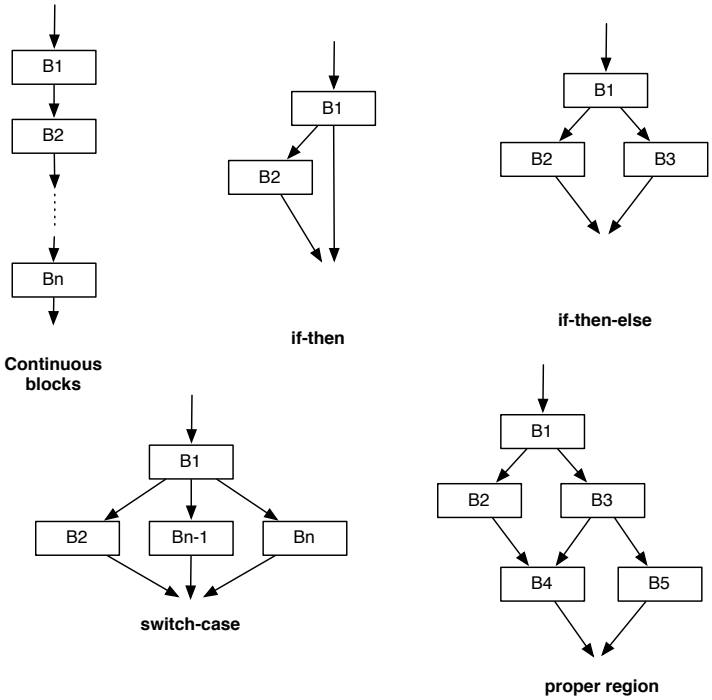


Figure 4.1: Acyclic structures that can be recognised by structural analysis. A proper region is of arbitrary size; the pictured example is the smallest possible proper region.

standard interval analysis techniques. Structural analysis works by analysing the CFG and matching regions against pre-defined patterns, called *schema*. The acyclic schema recognised by structural analysis are shown in Figure 4.1. The cyclic schema are shown in Figure 4.2.

First, we define the *acyclic schema*. These describe various conditional constructs, straight-line code and an acyclic "catch-all" schema.

- A sequence of basic blocks $S = \{b_1, b_k, ..., b_{n-1}, b_n\}$ where each b_k has exactly one successor b_{succ} and predecessor b_{pred} where $b_{succ}, b_{pred} \in S$ and $succ(b_n) \leq 1$ are identified as a schema of **continuous blocks**.

- Two basic blocks b_1 and b_2 are identified as the **if-then** schema iff: b_1 has exactly one predecessor and two successors, where $b_2 \in succ(b_1)$, $pred(b_2) = b_1$ and $succ(b_1) - b_2 = succ(b_2)$.

- Three basic blocks b_1, b_2 and b_3 are identified as an **if-then-else** schema iff: b_1 has exactly one predecessor, $succ(b_1) = \{b_2, b_3\}$, b_2 has exactly one successor and $succ(b_2) = succ(b_3)$.

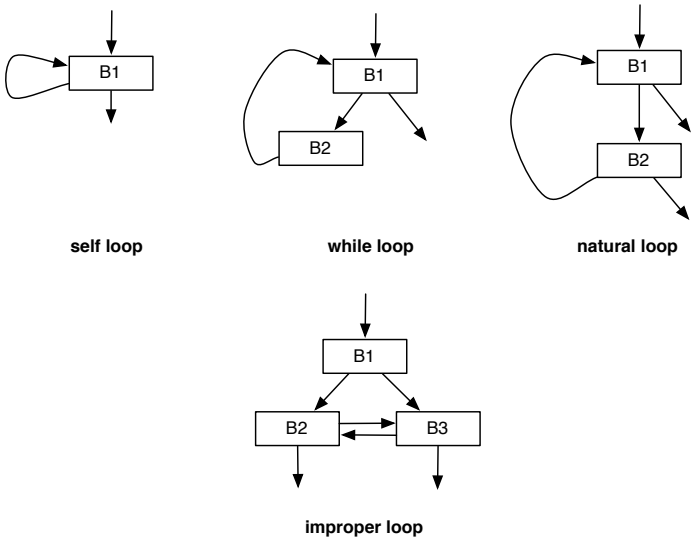


Figure 4.2: Cyclic structures that can be recognised by structural analysis. Like the proper region, the improper loop is schematic.

- A set of basic blocks $S = \{b_1, b_k, ..., b_{n-1}, b_n\}$ is identified as a **switch-case** schema iff: b_1 has exactly one predecessor, b_n has exactly one successor, and all blocks in the subsequence $\{b_k, ..., b_{n-1}\}$ have only b_1 as a predecessor and b_n as a successor[1].
- Any acyclic region of arbitrary size that does not match any of the above cases is a **proper region**.

Similarly, we can describe the *cyclic schema*. These represent a variety of loop constructs, and also a method for detecting irreducible regions.

- A basic block b_1 is a **self loop** schema iff it has exactly one distinct successor and predecessor, and an edge (b_1, b_1).
- Two basic blocks b_1 and b_2 form a **while loop** schema iff: b_1 has exactly one predecessor, b_1 has exactly two successors of which one is b_2, and $pred(b_2) = succ(b_2) = b_1$.
- A **natural loop** is any reducible loop that does not match the above schema.
- The **improper loop** schema applies to any *irreducible* regions of the code.

[1]The exact implementation of the switch-case schema can be adjusted to allow fall-through behaviour in case statements if desired.

When the structural analysis algorithm executes, it examines basic blocks in postorder, recognising regions and collapsing (reducing) them into abstract nodes. These abstract nodes are annotated with the type of reduction that has taken place. For example, the *if-then-else schema* from Figure 4.1 would become an *if-then-else abstract node* upon reduction. This continues until the limit graph is reached. In parallel, a structure called the *control tree* is constructed. This represents the hierarchical structure of the regions within the original graph. In Figure 4.3 we see an example of a CFG being reduced step-by-step by structural analysis. The original basic blocks are unshaded, whereas abstract nodes are shaded grey. Each basic block in the CFG has been annotated with its numbering in the postorder traversal. The structural analysis algorithm starts at the lowest number in the postorder and works upwards. Block `1` alone does not match any of the schema. However, blocks `1` and `2` match the *continuous blocks* schema and are reduced into an abstract node which we label `12a`. Block `3` alone does not match a schema, but block `4` matches the *self loop* schema and is reduced to `4a`. Now this allows `4a` and `5` to match the *continuous blocks* schema, being reduced to `45a`. Now blocks `6`, `3` and `45a` match the *if-then-else* schema, and are reduced to `3456a`. This leaves two blocks which are reduced using the *continuous blocks* schema into the limit graph `123456a`. Since the limit graph consists of only one node, this graph is reducible, and we have finished the structural analysis stage.

The control tree is generated during structural analysis is shown in Figure 4.4. We follow the same shading convention for abstract nodes and original basic blocks. It can be seen that all of the original basic blocks form leaf nodes, and abstract nodes are always interior nodes. We have now completed the CFG analysis phase and produced a control tree. This is the structure that we will work with to begin the VSDG construction phase.

One problem with Johnson's construction during syntax analysis is that unstructured exit from loops is not completely supported: recall that the algorithm quits upon seeing `switch` or `goto` keywords in the AST [72]. However, the benefit of using structural analysis is that a loop consisting of conditionals and `goto`s may take the same control flow structure as an existing loop schema even though its syntax would not necessarily show this.

An additional benefit of structural analysis is the opportunity to apply node splitting to irreducible regions. Since every region that the algorithm identifies has exactly one entry point, the improper schema will always include the lowest common denominator of the set of entries to the strongly connected component that is the multiple-entry cycle within the improper region [87].

Analysing the CFG using structural analysis has offered the following benefits which aid VSDG construction:

- An opportunity to recognise more patterns in the CFG than other methods of interval analysis, reclaiming much of the syntactic information lost after syntax analysis.

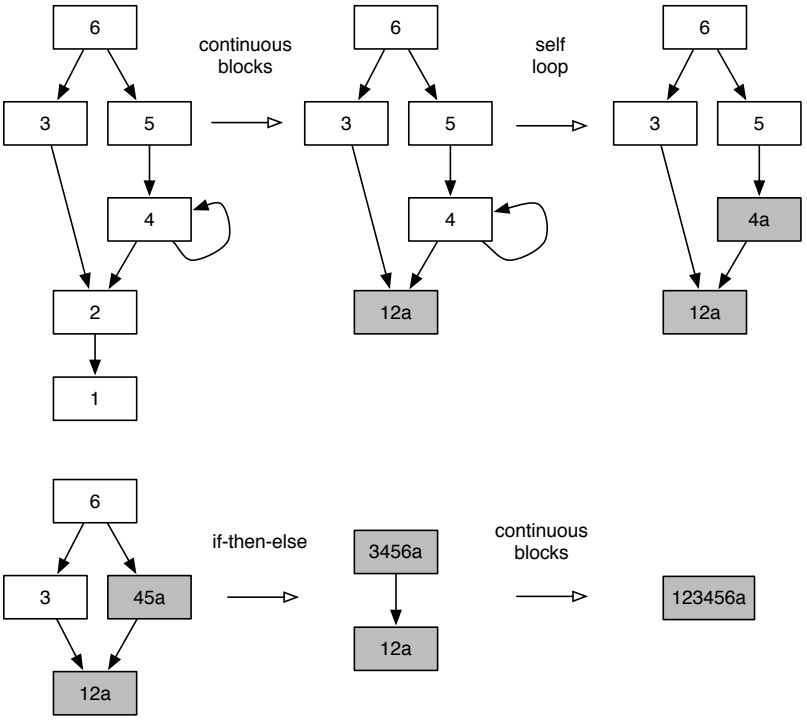


Figure 4.3: A CFG is reduced step-by-step by structural analysis into a one-node limit graph.

- Detection of irreducibility in the program, which can then be dealt with by whichever preferred method (e.g. node splitting).
- A better way of dealing with unstructured control flow, as unstructured loops will often match against schema their syntax may not immediately suggest.

We used our implementation of structural analysis to see the commonality of different region types in real world programs. This acts as an indicator as to which merge algorithms would be called the most often when performing this merging stage. We analysed the source code of the open source projects in Figure 3.5. The table in Figure 4.5 shows the occurrences of each region type found in all of the software.

4.4 Sketching the algorithm

Constructing a VSDG from the CFG consists of two stages:

1. Traversing the CFG to find regions.

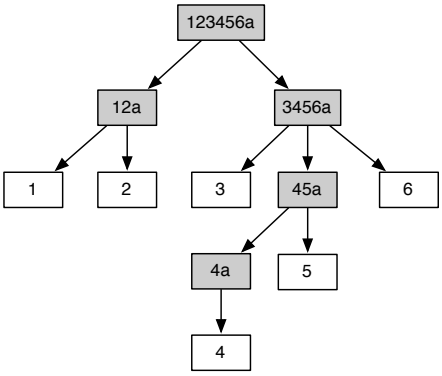


Figure 4.4: The resulting control tree from applying structural analysis to the CFG in Figure 4.3.

Region name	**Number of occurrences**
Continuous blocks	14715
If-then	8347
If-then-else	5739
Switch-case	252
Proper	3756
Self loop	191
While loop	2477
Natural loop	3263
Improper	5

Figure 4.5: Region occurrences for 15 open source projects from Figure 3.5 as of July 2009.

2. Using the discovered syntactic information about these regions to drive construction of the graph.

We have used structural analysis to achieve the first stage. In our construction approach, we split the second stage into two sub-stages:

2a. Creating a VSDG fragment for each basic block. These fragments represent straight-line code, and hence do not have any γ- or θ- nodes.

2b. Traversing the control tree and combining fragments by using the type of the abstract node to identify the γ- or θ- nodes (or none in the case of continuous blocks) to add.

4.4.1 Generating VSDG fragments

The first stage examines the instructions in each basic block in turn and creates a VSDG fragment. No gating functions are added until the merge stage, since this information is encoded in the control tree abstract nodes during structural analysis. We limit the scope of variables to the basic block that we are looking at. We generate nodes with value edges for instructions and their operands. Memory operations on values such as `load` and `store` are linked in the correct order with state edges, and we keep track of the most recent state operation for each value. We also keep track of the first and last usage of a value in each fragment. If a variable is being used that has not been defined in the current block (i.e. it was defined in a previous one), we create a *top-link* node. This is used in the merging stages to link our basic block fragments together. We do not generate nodes for terminating branch or jump instructions, as the destination to transfer control flow is explicitly shown in the edges of the CFG. Once we have performed this for all basic blocks, we traverse the control tree.

4.4.2 Control tree traversal

A postorder traversal of the control tree is required to construct the graph in the correct order. When traversing, we check if we are visiting a leaf node or an interior node. Leaf nodes are always the original basic blocks, so they have had a VSDG fragment generated in the previous stage. Interior nodes are abstract nodes, and identify the type of merge required. When we visit an abstract node, we perform the indicated merge. A merge at an abstract node creates a new fragment containing the children of that node. These merges continue until we reach the root of the control tree, where we will have constructed the complete VSDG by merging all of the fragments.

4.4.3 Merging

As we work through the control tree, we perform different merges on the child fragments of a given abstract node. The type of merge is identified by the type of abstract node we are visiting. For example, the first three nodes visited in postorder in Figure 4.4 would be `1`, `2` and `12a`. The abstract node `12a` was created during a *continuous blocks* reduction as per its annotation, specifying that the nodes `1` and `2` can be merged together as straight-line code without requiring any γ- or θ-nodes. Below we list the different types of abstract node and the type of merge we will perform upon seeing them. We begin with the abstract nodes representing acyclic schema. For every merge operation we seek to *resolve* top-link nodes if possible by looking for the previous use of a top-link node's value.

Continuous blocks This schema represents straight-line code. Each child fragment $G_k \in \{G_1, ..., G_n\}$ is merged in order into G_{Blocks}.

If-then This has two child fragments: G_C and G_T. The fragment that returns the γ-node condition is G_C. The fragment evaluated when this is true is G_T. We create a γ node and link the C port to G_C, and the T port to G_T. The F port will be linked later on, so a top-link is created and linked to the F port.

If-then-else The procedure is the same as above, except we have an additional fragment G_F executed when G_C is false. We link this to the F port of the γ node.

Switch-case Each child fragment represents an individual `case` statement in the input program. We translate these into a chain of γ nodes.

Proper A proper region is of arbitrary size, and again we translate these into a chain of γ nodes.

We also describe the merges for the cyclic abstract nodes.

Self loop This abstract node has one child. We generate a θ^{head} node and for the first usage of all values in the child fragment we create a link to the L port. For these same values we create a top-link which is linked to from the I port. We also generate a θ^{tail} node, with the last usage of all values being linked from the R port. The C port is linked to the conditional test that terminates the fragment.

While loop This abstract node has two children, and we treat it similarly to the SelfLoop merge. We link the loop body between the θ^{head} and θ^{tail} nodes as before, and link the condition to the C port of the θ^{tail} node.

Natural loop Like the *switch-case* and *proper region* schema, these are of arbitrary size. We create a μ node at the loop header, and at each loop exit we place a η node. The condition for each η node links to the terminating condition of each loop exiting block in the schema.

Improper regions can be transformed during structural analysis by node splitting so they should no longer be present at this stage. Alternatively, a procedure containing an improper region can be left unoptimised, a behaviour seen in many real-world compilers.

4.5 Generating and merging

The structural analysis algorithm was implemented in LLVM for the study of irreducible programs performed in Chapter 3. We therefore wrote the merging algorithm as an LLVM pass that ran after structural analysis completed, using the generated control tree. The LLVM IR is a CFG containing instructions that are implicitly in SSA form.

4.5.1 Generating VSDG fragments

Before we traverse the control tree, we generate a VSDG fragment for each basic block in the CFG. The goal of this stage is to populate a map data structure called $Fragments$, which will contain a reference to each basic block in the CFG mapped to the fragment that has been generated for it. We will begin with some definitions that will be used in the presentation of the algorithm.

Basic block A basic block $BB(Ins, GV, T)$ consists of a set of instructions Ins, a set of global variables GV and a terminating instruction T. The set of all basic blocks is called $Blocks$.

Instruction An instruction $I(Op, V, L, R, SE, BB)$ consists of the operation of the instruction Op, value stored in a virtual register V, operand L, operand R, a Boolean flag indicating whether it is a side-effecting instruction SE, and a reference to the basic block it is contained in BB.

We use a dot notation to access data contained within a tuple. For example, in order to access the type of the operation of instruction I, we would write $I.Op$. We use the $\leftarrow$ operator to indicate the storing of a value to a tuple. The operand(s) of an instruction may be a constant or a reference to another instruction, as is the case in LLVM.

Each fragment has a function header node. When all fragments have been merged into the final VSDG these function header nodes will have been merged together.

Function header A function header node $FH(\ell, Vals,$`__STATE__`$)$ has a unique label ℓ, a set of entry variables $Vals$ and the variable `__STATE__` [72].

We use the $\uplus$ operator to describe the addition of an element or set to an existing set. We also use this same operator to describe a value being added to the function header node, e.g. $S \uplus$ `x`.

The VSDG we wish to construct can consist of 4 types of node. These are value nodes, state nodes, conditional (γ) nodes and loop (θ^{head} and θ^{tail}) nodes. In this stage of the algorithm we are only required to generate value nodes and state nodes. The conditional and looping constructs are introduced at the merging stage. For the construction algorithm we also use a type of node called a *top-link*, which literally means "this variable has been declared, but not in this block". Top-link nodes are used to discover the correct places to join fragments together during the merging stage.

VSDG node A node $Node(T, Value, BB)$ consists of a type T (`value`, `state`, `gamma`, `mu`, `eta`, `top-link`), virtual register value $Value$, and reference to the originating basic block BB.

VSDG edge A directed edge $Edge(From, To, FPort, TPort)$ consists of a source node $From$, destination node To, and port labels from the source node $FPort$ and destination node $TPort$.

As mentioned previously, the global data structure for this stage of the algorithm which is accessible to later stages is a map

$$Fragments : (BB \rightarrow (N, E, FHs, CurrentNode, CurrentState, ThetaMap))$$

which maps each basic block BB with the generated VSDG fragment. This information is the nodes (N) and edges (E), the function header node (FHs) along with a map

$$CurrentNode : (V \rightarrow N)$$

mapping a value with the latest node for it in the fragment, and the map

$$CurrentState : (V \rightarrow N)$$

mapping a value with the latest state-changing node for it in the fragment. The map

$$ThetaMap : (BB \rightarrow N)$$

is used for merging loop regions with multiple exits, and will be explained in more detail later.

For brevity, we assume the existence of a function `ValueLink(X,Y)` which creates a directed value edge from node `X` to node `Y` and adds this edge to the set E. We also assume a similar function `StateLink(X,Y)` exists which creates a directed state edge from node `X` to node `Y`, and also adds it to the set E. We assume the correct port names are assigned to edges during this process in order to save space.

We now consider Algorithm 2, which handles the construction of a fragment from a basic block. If this is the first basic block in the CFG, all global variables declared are added to the function header. Then, for each instruction a VSDG node is constructed. If the instruction is side-effecting, then a state edge is created between the newly created instruction node and the previous side-effecting usage of that value.

Algorithm 3 creates a node for each instruction. If the instruction is side-effecting, it is labelled as a `state` node, and the L operand is constructed. Both `load` and `store` instructions have a L operand, but `store` nodes also have a R operand. This new node is now linked to the previous state usage of the computed value, whether this be in the function header or a previous node. If the node does not have side-effects, then it is created and labelled as a `value` node, updated as the current node for that value, and its operands are constructed.

Algorithm 2: Construct a VSDG fragment for each basic block in the CFG.

Input : The set of all basic blocks in the CFG: *Blocks*.
Output: The populated data structure *Fragments*, containing a fragment for each basic block.

```
forall the BB ∈ Blocks do
    if first then
        first ←false;
        forall the gv ∈ GV do
            FH ⊎ gv;
    forall the I ∈ Ins do
        ConstructInstruction(I);
        if I.SE then
            StateLink(CurrentState(I.V),I);
    FHs ⊎ FH;
    Fragment ⊎ (N,E,FHs,CurrentNode,CurrentState);
    Fragments(BB) ← Fragment;
```

Algorithm 3: Construct a VSDG node for each instruction.

Input : An instruction I.
Output: A constructed node for this instruction added to N.

```
if I.SE then
    n.T ← state;
    ConstructOperand(n,I.L);
    if I.Op is store then
        ConstructOperand(n,I.R);
    if CurrentState(I) is null then
        StateLink(n,FH,__STATE__ )
    else
        StateLink(n,CurrentState(I),I);
else
    n.BB ← I.BB;
    n.V ← I.V;
    n.T ← value;
    CurrentNode(I.V) ← n;
    ConstructOperand(n,I.L);
    ConstructOperand(n,I.R);
N ⊎ n;
```

Each instruction will have some number of operands. Algorithm 4 handles this. An operand can be several things. If it is a constant, we simply join it to the parent constant node. Otherwise, we check for the previous use of this value. If it is not present, we create a top-link node. Else, we link to the previous value.

Algorithm 4: Construct a VSDG node for each operand.

Input : The parent instruction node n, and the operand to construct O.
Output: A constructed node for this operand added to Nodes.

```
Opr.T ← value;
Opr.V ← O.V;
if O is a constant then
    ValueLink(n,Opr);
else
    if CurrentNode(O.V) is null then
        Opr.BB ← O.BB;
        Opr.T ← top-link;
        ValueLink(n,Opr);
        CurrentNode(O.V) ← Opr;
    else
        ValueLink(n,CurrentNode(O.V));
N ⊎ Opr;
```

4.5.2 Control tree traversal

We now have generated a fragment for each basic block in the CFG, stored in the map *Fragments*. These fragments, when merged together in the correct way, will form the completed VSDG. The order in which they are joined together is dictated by the syntactic structure of the program encoded in the abstract nodes of the control tree. We perform a postorder traversal of the control tree. Each leaf node of the control tree is a basic block. Each interior node is an abstract node created when a reduction occurred during structural analysis. The postorder numbering is stored in a map

$$PostNums : (int \rightarrow BB)$$

generated by Algorithm 5.

4.5.3 Merging

Using this postorder numbering, we can then process the control tree and perform merges when we reach interior nodes. We visit each node in the control tree in postorder. If the node is a leaf node it is added to a set called RegionNodes. When we reach an interior node, the function performing the relevant

Algorithm 5: Number the control tree in postorder.

Input : The root node of the control tree.
Output: The postorder numbering map *PostNums*.

```
TreeNode.Visited ← true;
forall the c ∈ TreeNode.Children do
    if c.Visited is false then
        TraverseControlTree(c);
    PostMax ← PostMax +1;
    PostNums(PostMax) ← TreeNode;
```

merge is called with this set of nodes as the parameter. These functions merge these child fragments into one fragment, using the syntactic information encoded in the abstract node.

Algorithm 6: Merge together the nodes, edges and function headers of fragments from a particular schema.

Input : RegionNodes, the set of all basic blocks in a schema.
Output: A single fragment containing all of the elements from the fragments generated from RegionNodes.

```
forall the r ∈ RegionNodes do
    CurrentFragment ← Fragments(r);
    MergedN ⊎ CurrentFragment.N;
    MergedE ⊎ CurrentFragment.E;
    forall the x ∈ CurrentFragment.FHs do
        MergedFH ⊎ x;
return (MergedN,MergedE,MergedFH,null,null,null);
```

Some similar processes occur in each abstract node merge. To begin with we must merge the nodes and edges of each fragment together. We also have to merge together the function headers. We do this in Algorithm 6. The other common process is resolving top-links. This is presented in Algorithm 7. Here, we check to see if there is a node in the current fragment that is a use of the same value (or state, depending on the edge type) that the top-link refers to. If so, we can update the edge to point to this node. We also update the current node and state usage of a value whilst doing so. We check whether the ThetaMap has been passed as an argument if we need to resolve the top-links with multiple loop exits.

Now we can use processes to merge two fragments. We begin by showing the process taken when merging together two fragments that are part of a ContinuousBlocks schema; that is, they form straight-line code in the input program. This does not introduce any γ- or θ-nodes and is shown in Algorithm 8. Here, the nodes, edges and function headers are merged together into one fragment using the `MergeElements` algorithm. Then, any edges pointing

Algorithm 7: Resolve the top-link nodes in a fragment, where possible.

Input : Fragment with unresolved top-link nodes.
Output: Fragment with resolved top-link nodes, where possible.

```
forall the e ∈ MergedEdges do
    if ThetaMap is not null then
        BB ← e.From.BB;
        ThetaTail ← ThetaMap(Pred(BB));
        if e is a value edge then
            e.To ← ThetaTail;
            MergedCurrentNode(TopLink.V) ← e.From;
        else if e is a state edge then
            e.To ← ThetaTail;
            MergedCurrentState(TopLink.V) ← e.From;
    else
        if e.To is a top-link then
            TopLink ← e.To;
            if e is a value edge then
                if MergedCurrentNode(TopLink.Value) is not null then
                    e.To ← MergedCurrentNode(TopLink.V);
                    MergedCurrentNode(TopLink.V) ← e.From;
            else if e is a state edge then
                if MergedCurrentState(TopLink.V) is not null then
                    e.To ← MergedCurrentState(TopLink.V);
                    MergedCurrentState(TopLink.V) ← e.From;
            if TopLink.Indegree is 0 then
                MergedEdges − TopLink;
```

at the old function headers are updated, followed by the `ResolveTopLinks` algorithm.

In an IfThen merge, we use a γ-node in order to represent conditional choice. We show the algorithm for creating an IfThen region in Algorithm 9. Since the IfThenElse algorithm is so similar, it has been included in the appendix (Algorithm 16). For each register variable being used in the `then` part of the `if` statement, a γ node is created. Then, the C port of each γ node is linked to the terminating conditional instruction in the `if` statement. Since an IfThen region does not have an `else` block, a top-link is created so the F port can be connected later in the merging process.

Next we consider the SwitchCase region. This is an arbitrary size schema that matches against a typical `switch...case` construct in code. Since the γ node cannot handle multiple predicates and selection, nor do we have a "switch" node to use, we must use a chain of γ-nodes in order to replicate

Algorithm 8: Merge for the ContinuousBlocks schema.

Input : RegionNodes, which contains the basic blocks identified as the continuous blocks schema.
Output: The merged ContinuousBlocks schema fragment.

```
CurrentFragment ← MergeElements(RegionNodes);
forall the e ∈ MergedE do
    if e is a value edge then
        if e.To is a state other than MergedFH then
            if MergedCurrentNode(e.To) is null then
                e.To ← MergedFH;
                MergedCurrentNode(e.To) ← MergedFH;
            else
                e.To ← MergedCurrentNode(e.To);
    else if e is a state edge then
        if e.To is a state other than MergedFH then
            if MergedCurrentState(e.To) is null then
                e.To ← MergedFH;
                MergedCurrentState(e.To) ← MergedFH;
            else
                e.To ← MergedCurrentState(e.To);
ResolveTopLinks(CurrentFragment);
Fragments(CurrentBlock) ← CurrentFragment;
```

the behaviour. We show this in Algorithm 10. We iterate over all blocks representing `case` statements. For each register variable, we create a γ node of which the T port is linked to the current node for the value in this block. For the F port, we use a map to link to the previous γ node for this value, if it exists, or we create a top link to be resolved later. Since the Proper region is of arbitrary size and is acyclic, we can utilise the approach in Algorithm 10, and we do so in Algorithm 11. Each basic block in a Proper schema will have some arbitrary number of children greater than 1. Thus, we can just treat each block examined and its successors as a case statement, and repeatedly call Algorithm 10 on all nodes in the region.

For the WhileLoop region, we must insert the θ^{head} and θ^{tail} nodes during the merge, and Algorithm 12 does this. Prior to merging together the elements of each fragment, we create θ^{head} and θ^{tail} nodes. Then, for each register variable in the loop body, the first usage is linked to the L port of the θ^{head} node. A top-link is created for each L port variable from the I port. For the θ^{tail} node, the last usage of each register variable is linked to from the R port. The C port links to the terminating conditional instruction of the loop. The SelfLoop merge is very similar, except it only consists of one block. Due to this, the SelfLoop algorithm (Algorithm 17) has been placed in the appendix.

Algorithm 9: Merge for the IfThen schema.

Input : RegionNodes, which contains two fragments to merge.
Output: The merged IfThen schema fragment.

```
CondBlock ← RegionNodes(0);
TrueBlock ← RegionNodes(1);
CondFragment ← Fragments(CondBlock);
TrueFragment ← Fragments(TrueBlock);
forall the register variables v ∈ TrueBlock do
    create new GammaNode;
    create new TrueEdge;
    if TrueFragment.CurrentNode(v) is not null then
        TrueEdge.To ← TrueFragment.CurrentNode(v);
        TrueEdge.From ← GammaNode.TPort;
    else
        CreateTopLink(v,GammaNode.TPort);
    CreateTopLink(v,GammaNode.FPort);
    create new CondEdge;
    CondEdge.From ← GammaNode.CPort;
    CondEdge.To ← CondFragment.CurrentNode(CondBlock.T);
    MergedCurrentNode(v) ← GammaNode;
    MergedCurrentState(v) ← GammaNode;
    MergedN ⊎ GammaNode;
    MergedE ⊎ CondEdge, TrueEdge;
CurrentFragment ← MergeElements(RegionNodes);
ResolveTopLinks(CurrentFragment);
Fragments(CurrentBlock) ← CurrentFragment;
```

The remaining merge algorithm is for the NaturalLoop schema. This represents any reducible loop that is not one of the other looping constructs as it has two or more exit edges. Algorithm 18 shows this merge in the appendix.

Algorithm 10: Merge used in acyclic schema of arbitrary size (SwitchCase and Proper).

Input : RegionNodes, which contains basic blocks, and a flag of either `CASE` or `PROPER` depending on the type of schema being merged.
Output: The merged ArbitrarySize fragment.

```
CondBlock ← RegionNodes(0);
CondFragment ← Fragments(CondBlock);
ctr ← 0;
while ctr < RegionNodes.Size do
    CurrentBlock ← RegionNodes(ctr);
    CurrentFragment ← Fragments(CurrentBlock);
    forall the register variables v ∈ CurrentBlock do
        create new GammaNode;
        create new TrueEdge;
        if CurrentFragment.CurrentNode(v) is not null then
            TrueEdge.To ← CurrentFragment.CurrentNode(v);
            TrueEdge.From ← GammaNode.TPort;
        else
            CreateTopLink(v,GammaNode.TPort);
        create new FalseEdge;
        if PrevGamma(v) is not null then
            FalseEdge.To ← PrevGamma(v);
            FalseEdge.From ← GammaNode.FPort;
        else
            CreateTopLink(v,GammaNode.FPort);
        PrevGamma(v) ← GammaNode;
        create new CEdge;
        CEdge.From ← GammaNode.CPort;
        CEdge.To ← CondFragment.CurrentNode(CondBlock.T.Case(ctr));
        CurrentFragment.CurrentNode(v) ← GammaNode;
        CurrentFragment.CurrentState(v) ← GammaNode;
        MergedN ⊎ GammaNode;
        MergedE ⊎ CEdge, TrueEdge, FalseEdge;
    ctr ← ctr + 1;
if PROPER then
    RegionNodes ← RegionNodes − RegionNodes(0);
CurrentFragment ← MergeElements(RegionNodes);
ResolveTopLinks(CurrentFragment);
Fragments(CurrentBlock) ← CurrentFragment;
```

Algorithm 11: Merge used for Proper regions.

Input : RegionNodes, which contains the basic blocks in a Proper schema..
Output: The merged ProperRegion fragment.

```
ctr ← 0;
while ctr < RegionNodes.Size do
    SubRegion ← ∅;
    CurrentBlock ← RegionNodes(ctr);
    SubRegion ⊎ CurrentBlock;
    SubRegion ⊎ Succ(CurrentBlock);
    CurrentFragment ⊎ ArbitrarySize(SubRegion,PROPER);
Fragments(CurrentBlock) ← CurrentFragment;
```

4.6 Worked example

We will show how the algorithm works by demonstrating it on an example program. In Figure 4.6 we give a C program alongside its translation into a linear IR. We obtained this by compiling the program with the LLVM compiler, and then editing it by hand to make it more readable.

4.6.1 Structural analysis

Figure 4.7 shows the CFG for the code in Figure 4.6 and the reduction steps that occur during structural analysis. Traversing the blocks in the CFG in postorder, the structural analysis algorithm first detects an *if-then-else* schema made up of the blocks `w.b`, `i.t` and `i.el`. These are reduced into the abstract node `a1`. This allows for `a1` and `i.en` to match the *continuous blocks* schema, so they are reduced into `a2`. Now `w.c` and `a2` match the *while loop* schema and are reduced into `a3`. The next three reductions repeatedly apply the *continuous blocks* schema until the limit graph is reached. The resulting control tree is shown in Figure 4.8.

4.6.2 Generating VSDG fragments

Before traversing the control tree, we generate a fragment for each basic block in the CFG. We begin with the basic block named `entry`. Since this is the first basic block, we create a function header, which includes the `__STATE__` pseudo-register that state-dependent nodes will interact with. The function takes a parameter `x` so this is added to the function header. The first two instructions allocate memory for `x` and `y` so we add these as well. We generate nodes and edges for the multiplication of `x` by `2`. This is now the most recent value for `y`. For basic block `w.c` our only non-terminating instruction is `sgt`. It is operating on `x`, but since it was defined elsewhere we create a top-link node which is represented by a cloud in the diagram. This means "`x` has been allocated, but not in this block". We follow the same principle for other blocks.

Algorithm 12: Merge for the WhileLoop schema.

Input : RegionNodes, which contains the basic blocks from the WhileLoop schema.
Output: The merged WhileLoop fragment.

```
CondBlock ← RegionNodes(0);
BodyBlock ← RegionNodes(1);
CondFragment ← Fragments(CondBlock);
BodyFragment ← Fragments(BodyBlock);
create new ThetaHead;
forall the register variables v ∈ CondBlock do
    create new LEdge;
    LEdge.To ← CondFragment.FirstUse(v);
    LEdge.From ← ThetaHead.LPort;
    CreateTopLink(v,ThetaHead.IPort);
    CondFragment ⊎ LEdge;
create new ThetaTail;
forall the register variables v ∈ BodyBlock do
    if v ∉ CondBlock then
        create new LEdge;
        LEdge.To ← BodyFragment.FirstUse(v);
        LEdge.From ← ThetaHead.LPort;
        CreateTopLink(v,ThetaHead.IPort);
        BodyFragment ⊎ LEdge;
    create new REdge;
    REdge.From ← ThetaTail.RPort;
    REdge.To ← BodyFragment.CurrentNode(v);
    MergedCurrentNode(v) ← ThetaTail;
    MergedCurrentState(v) ← ThetaTail;
    MergedE ⊎ REdge;
create new CEdge, SEdgeUp, SEdgeDown;
CEdge.From ← ThetaTail.CPort;
CEdge.To ← CondFragment.CurrentNode(CondBlock.T);
SEdgeUp.From ← BodyFragment.FirstSideEffect;
SEdgeUp.To ← ThetaHead.STATE;
SEdgeDown.From ← ThetaTail.STATE;
SEdgeDown.To ← BodyFragment.LastSideEffect;
MergedN ⊎ ThetaHead, ThetaTail;
MergedE ⊎ CEdge, SEdgeUp, SEdgeDown;
CurrentFragment ← MergeElements(RegionNodes);
ResolveTopLinks(CurrentFragment);
Fragments(CurrentBlock) ← CurrentFragment;
```

```
int foo(int x) {
    int y = x * 2;
    int z;
    while(x > 100) {
       x++;
       if(x % 2 == 0)
            z += x;
       else
            z += y;
    }
    return z;
}
```

```
int foo(i32 x) {
    entry:
          y = alloc i32
          z = alloc i32
          y = x mul 2
          jmp @w.c
    w.c:
          %cmp = x sgt 100
          br %cmp, @w.b, @w.e
    w.b:
          x = x add 1
          %tmp = x mod 2
          %cmp2 = %tmp eq 0
          br %cmp2, @i.t, @i.el
    i.t:
          z = z + x
          jmp @i.en
    i.el:
          z = z + y
          jmp @i.en
    i.en:
          jmp @w.c
    w.e:
          ret z
}
```

Figure 4.6: Some C code which has been translated into a linear IR. Identifiers with a `%` indicate local variables, and those with a `@` indicate labels.

The return node (`ret` instruction in basic block `w.e`) is state-dependent, as all other operations must finish before we can return the value of `z`. It is dependent on the previous usage of `z` in the completed VSDG. We have now generated our fragments for each basic block in the CFG. We keep a reference between each basic block and its fragment. The next stage is traversing the control tree and performing merges at abstract nodes.

4.6.3 Traversing and merging

The control tree in Figure 4.8 has been annotated with the numbering of each block after a postorder traversal. We consider the nodes of the tree in this order. Once we reach an abstract node, we take the fragments for each of the child nodes and merge them together. The first merge required is `w.b`, `i.t` and `i.el` into `a1`. This abstract node represents an *if-then-else* reduction, so we need to perform an *if-then-else* merge.

- Block `w.b` forms the G_C fragment. It contains the condition controlling

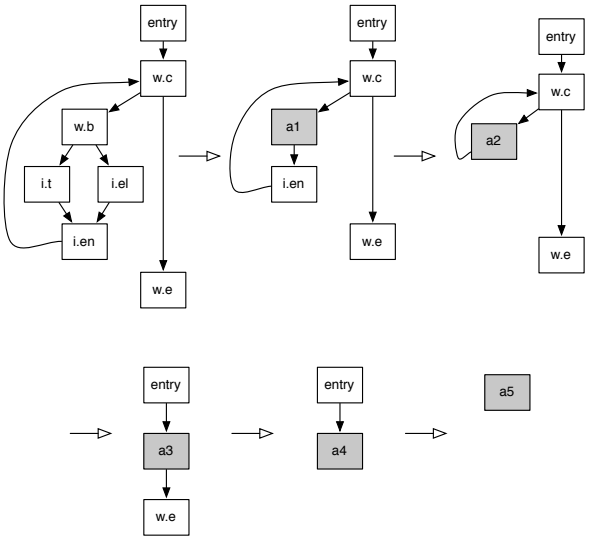


Figure 4.7: Performing structural analysis on the CFG from Figure 4.6.

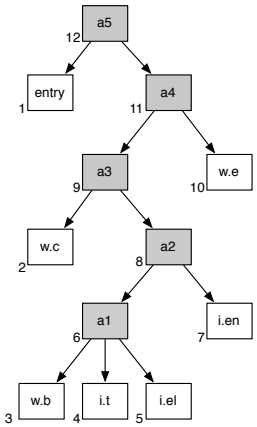


Figure 4.8: The control tree generated after structural analysis in Figure 4.7. The blocks have been annotated with their postorder numbering.

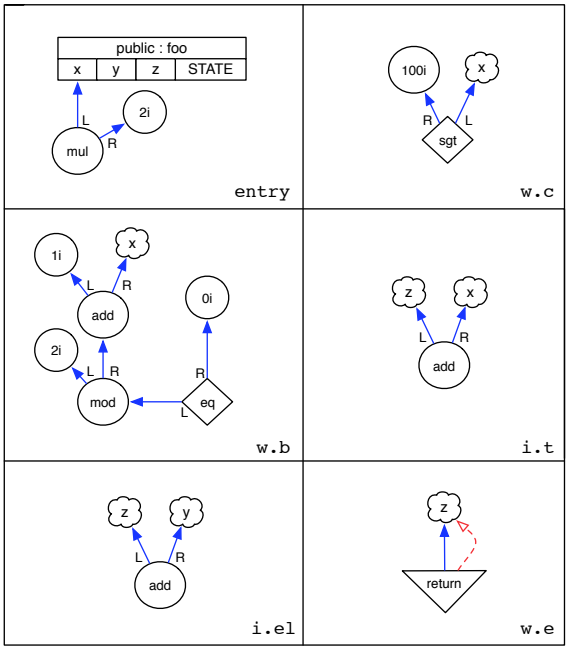


Figure 4.9: VSDG fragments generated for each basic block in the CFG. Note that since the basic block `i.en` does not contain any non-terminating instructions, we do need to generate any nodes for it.

the `if` statement: the `eq` instruction in Figure 4.9.

- Block `i.t` forms the G_T fragment.
- Block `i.el` forms the G_F fragment.

We create a new fragment, G_γ. As per the previous description, we generate a γ-node and add it to this fragment. The C port of the γ-node is linked to the conditional instruction in G_C, which is the `eq` node. We merge fragment G_C with G_γ, and add this edge. We then see if we can resolve any top-links. There is a top-link to `x`, but we have no previous usages of `x` in G_γ.

We then merge G_T with G_γ, and the T port is linked to the topmost instruction in G_T, which is `add`. We have two new top-links for `x` and `z`. The `add` instruction in G_C was a previous usage of `x`, so we can remove the top-link and add an edge to here. We have no previous usages of `z`. There are no state-changing nodes in G_T, so we add a top-link to `__STATE__`. We then merge G_F with G_γ. The F port is linked to the topmost instruction in G_F, which is another `add` instruction. We merge the `z` top-link with the existing one as they operate on the same value. There is no previous usage of `y`. There are no state-changing nodes in G_F either, so we add a top-link to `__STATE__`.

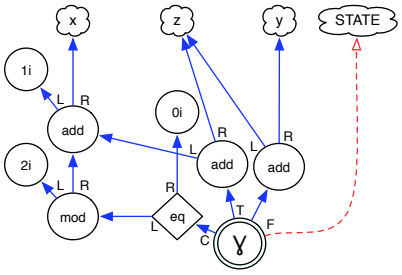


Figure 4.10: The fragment G_γ created after performing a merge at abstract node `a1`.

A γ-node is a state node itself. It now is the topmost instruction and state usage in G_γ (i.e. it must be evaluated first). We can see the G_γ fragment in Figure 4.10.

The other interesting merge occurs at `a3`. This abstract node was created from a *while loop* schema reduction during structural analysis. This merge involves generating θ^{head} and θ^{tail} nodes for the loop header and footer. The loop header θ^{head} receives its initial values through the I port. We haven't merged with that yet, so we generate top-link nodes. G_γ forms the body of the loop. The θ^{tail} node implements the loop guard condition through the C port, dependent on the initial value of `x`.

We continue performing merges when we visit an abstract node in the control tree until we have obtained the finished VSDG. This is shown in Figure 4.11. Here we can see the addition of the return node (`ret`) which demands the value of `z` and is state-dependent on the loop. The state for the beginning of the function is now present, along with the original values for `x`, `y`, `z` and `__STATE__`. The algorithm has now finished.

4.7 Summary

In this chapter, we have presented a method for constructing VSDGs that unites the benefits of previous methods without replicating their problems. In summary, the advantages of our approach are:

Language-independence Johnson's construction during syntax analysis using the AST tied that particular implementation to a single language. By operating on the CFG, our approach keeps graph construction language-independent.

Dealing with irreducibility Structural analysis is able to detect irreducible regions in the CFG. Node splitting can be applied to these regions as they are detected. Another benefit of using structural analysis is that

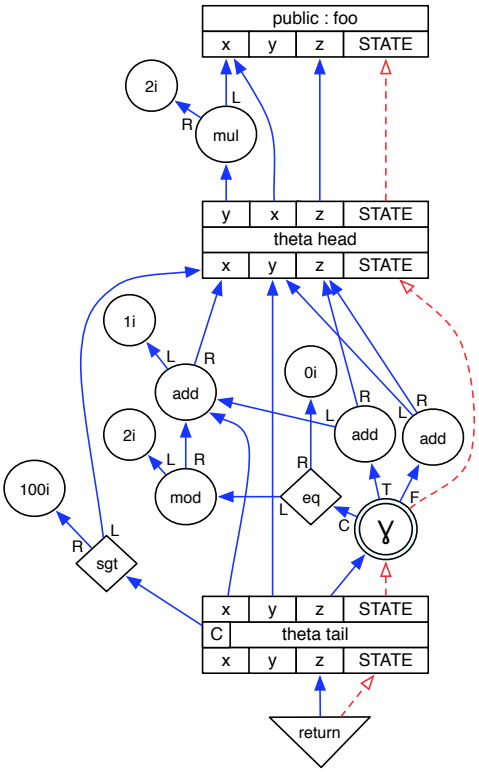


Figure 4.11: The complete VSDG constructed from Figure 4.6.

an irreducible region will always contain the lowest common dominator of the set of entries to the strongly connected component, thus node splitting will always be performed on the smallest possible irreducible region.

Dealing with unstructured control flow Unstructured portions of code that contain loops consisting of conditional statements and `gotos` can match against schema that their syntax would not necessarily suggest.

Modularity Keeping CFG structural analysis and VSDG construction decoupled and modular makes it easier to program and debug. Additionally, the independence of each merge algorithm pulls apart graph construction into further modular pieces, which allows for easier unit testing.

Simplicity Each stage of the graph construction algorithm requires only basic data structures, mostly in the form of maps, in order to store the first and current usage of values and states. This keeps both computational and *mental* overhead down when programming each merge stage, as each has a uniform input (n fragments) and output (1 fragment).

Running time Although our construction algorithm is detailed, the run time is not excessive. Computing dominators is done via Lengauer and Tarjan's $O(m\ log\ n)$ method [84] where m is the number of edges and n is the number of nodes in the CFG. Structural analysis itself has no formal run-time proof, however, when we used it to analyse the open source software in the previous chapter, performance was always acceptable. Once the control tree has been constructed by structural analysis, the generating phase is linear, requiring just one pass over each basic block and its instructions, and the merging stage also visits each basic block exactly once.

We believe that these factors that benefit both the compiler and the *programmer* make our algorithm for VSDG construction more advantageous than existing approaches in the literature.

Chapter 5

Proceduralisation

5.1 Introduction

The Value State Dependence Graph (VSDG) discards control flow entirely and represents programs purely as data dependencies, retaining only essential sequential dependencies, e.g. for I/O operations. This allows a wide variety of optimisations to be performed easily, but complicates the generation of sequential code as control flow must be *restored* by the back-end of the compiler.

We explore previous incomplete attempts to restore control flow to the VSDG and examine their differences and similarities. Then, we present two implementations based on the proceduralisation work of Lawrence [83]. We show how one of these implementations is unable to deal with a specific type of VSDG, and how the other implementation solves this, thus arguing that the longstanding problem of sequentialising the VSDG is now solved – provably so in practice as well as in theory.

This chapter presents the following:

- We outline the problem of generating sequential code from the VSDG, and explore previous attempts made in the literature to solve this problem.
- We present our software tool, `proc`, which translates VSDGs from Johnson's VECC compiler into PDGs with virtual registers allocated using Lawrence's proceduralisation algorithm.
- We use `proc` to show output of Lawrence's naïve and effective algorithms, showing how the effective algorithm deals with VSDGs exhibiting *independent redundancy*.
- We then show how to compile θ-node loops into PDGs by modifying Lawrence's existing proceduralisation framework.

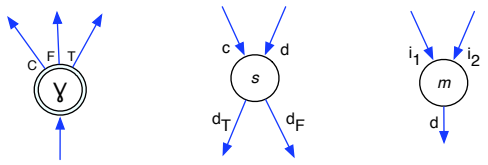
5.2 Previous work

Although the VSDG makes many optimisations easier or indeed seemingly automatic [72, 123], generating sequential code from it has proved problematic. The power and simplicity of the VSDG is achieved by discarding all specification of instruction ordering and control flow bar the minimum necessary to specify the input program. This gives great opportunity for performing optimisations. However, in order to generate sequential code, a total ordering of operations, as in the CFG, must be restored to the VSDG. This process is called *sequentialisation*; despite previous attempts, it remains a – Upton suggests *the* – major obstacle to building a compiler based on the VSDG [123].

The idea of restoring control flow by translation into a CFG predates the VSDG – Weise et al. [125] state that the VDG is translated back into a CFG via a demand-based PDG (dPDG). A dPDG is defined as a PDG where the CDG is replaced by a demand dependence graph. This demand dependence is "characterised by the γ-nodes encountered on paths from a return node", in which "any path from a return node yields a sequence of γ selector ports and a corresponding sequence of selector/predicate values along that path". However, the authors give no formal algorithms or references for the analysis of these demand conditions or the translation from VDG into dPDG. An interesting point is that the mapping from the VDG to the CFG is many-to-one, so a highly important factor of code generation from the VSDG is the encoding of an *evaluation strategy* that maps to exactly *one* CFG.

Johnson et al. [70] originally approached the problem by defining a *sequential* VSDG as one which has a *split node* (Figure 5.1) matching every γ-node and enough serialising edges to make it correspond to a single CFG. "Enough" means that each VSDG operation node has a unique immediate dominator which can be seen as a predecessor in the CFG. A split node is the push semantics [72] equivalent of a γ-node: it takes a value, d, and a condition, c, and outputs the value to (only) one of its two outputs d_T or d_F depending on the condition. Thus, the split node parallels branching in the CFG, identifying which nodes will be evaluated only according to runtime conditions. A matching merge node recombines the values: two inputs are received (i_1 and i_2) and merged to one output d. This is similar to the γ-node, but in the push semantics its condition input can be omitted (by construction, in any execution a value will only be pushed into one input). In Johnson's algorithm, split/merge pairs were strictly nested. Given a γ-node, the T and F port will postdominate two subregions of the graph respectively. The split node is inserted as the immediate E_S dominator of both of these subregions. For θ-loops, the unique immediate postdominator property is a constraint on the I port. Johnson provides no algorithm for the placement of these nodes.

However, deciding where to place split nodes is a key part of the sequentialisation process, as it selects the *evaluation strategy* for the output program. Johnson, optimising for size rather than speed, and thus permitting speculative



(a) (b)

Figure 5.1: A γ-node (a) and corresponding split and merge nodes (b).

evaluations to minimise duplication of expressions, implicitly chose an eager strategy. This placed split nodes according to dominance by true/false ports of γ-nodes. We observe that his technique effectively decides upon a PDG, in that the nodes *between* a split and its matching merge are precisely those for which the corresponding PDG nodes are children of the PDG predicate node via its true or false edges. Johnson's algorithm for combined register allocation and code motion can then be seen as operating on a PDG, mutating and refining it in order to limit register pressure, whilst totally ordering the children of each region node.

The question of optimising for speed was addressed by Upton, who showed that for that purpose, the optimal placement of split nodes[1] is NP-complete [122]. Upton's algorithm sequentialises a Gated Data Dependence Graph, a similar IR to the VSDG, by constructing a demand-based PDG. He first computes demand conditions, which are Boolean expressions over the various γ-node predicates, describing the control conditions under which the result of a node is demanded, by propagation through the graph. Then, Upton's algorithm traverses the graph and produces a hierarchical demand-based PDG, which must be iteratively refined in order to remove redundant predicate tests before generating code from it.

Lawrence [83] proposed a new compiler architecture by breaking sequentialisation into several stages: the first of these being the translation of the VSDG into a Ferrante et al. PDG [57] – a process he called proceduralisation. In his thesis, Lawrence presents two algorithms for the proceduralisation, one being a naïve approach that produces similar results to Johnson, and the other being a more effective approach which succeeds by encoding an evaluation strategy into the PDG. Since these algorithms were not implemented, in this chapter we present our software tool `proc` which takes VSDG files from Johnson's VECC compiler as input and produces PDGs.

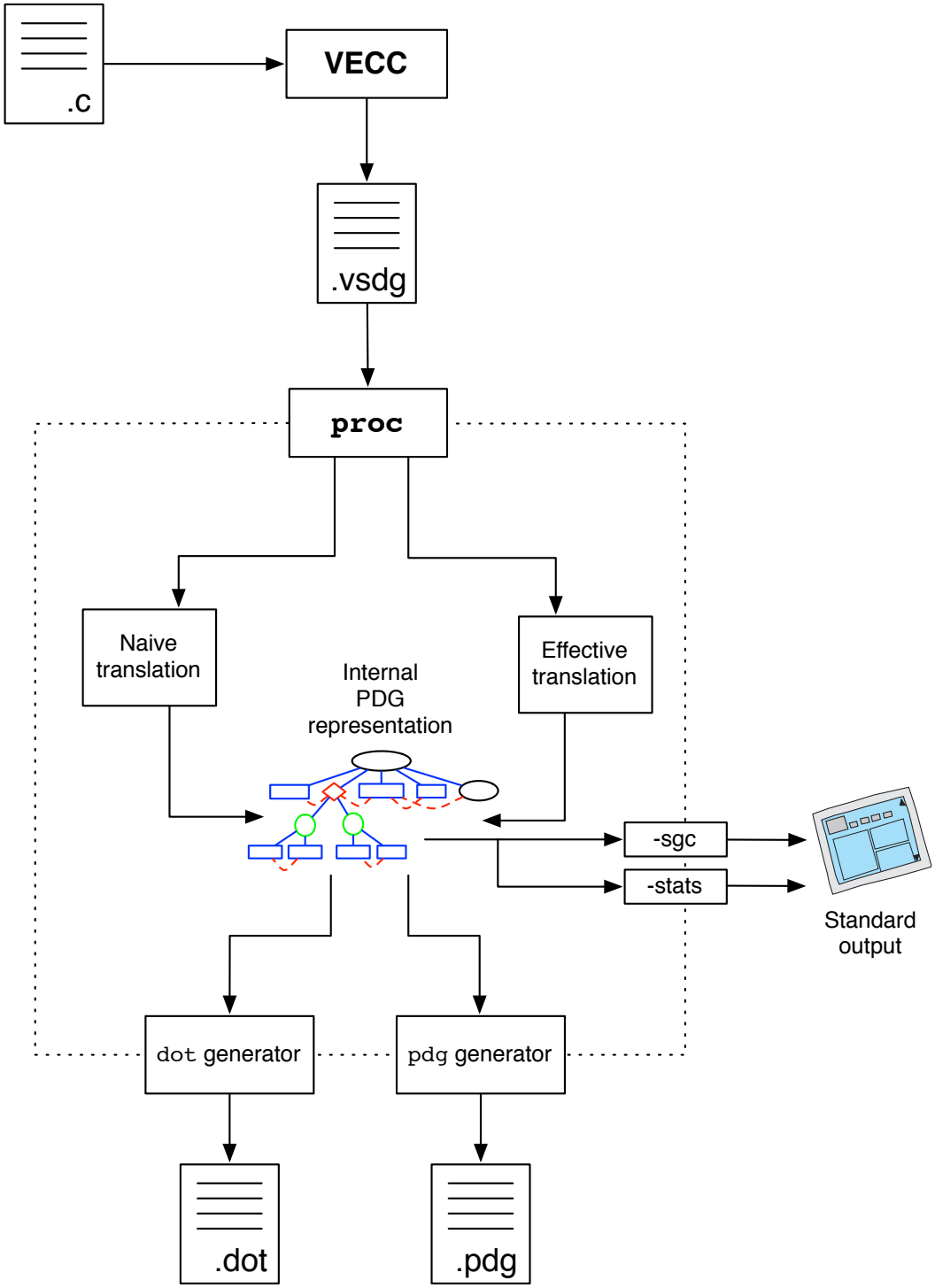


Figure 5.2: An outline of the `proc` tool.

5.3 The `proc` tool

This chapter describes a software tool called `proc` for applying proceduralisation to the VSDG. We utilised the VECC compiler, written by Johnson [72] to obtain VSDG input. The VECC compiler reads in a C source file and converts it into a VSDG, emitting a VSDG file. A VSDG file is a human-readable description of the graph. We used the grammar for these VSDG files to produce a handwritten lexical analyser and parser which forms one of the front-ends of the tool. With an input VSDG, the tool can apply either of the two algorithmic approaches outlined by Lawrence for proceduralisation, resulting in a PDG. This PDG can be output as a `dot` [3] file in order to visually display it, or as a human-readable `pdg` file. The syntax of the `pdg` files is given in Appendix C, along with an example. These `pdg` files are used as input to the code generation tool described in the next chapter. The tool is written in C++ using only standard library functions.

The tool is invoked as follows:

```
proc [-help] [-naive|-full] [-dot|-sgc|-stats|-pdg] inputfile
```

We provide the following functionality in the tool:

`-help` Display the help dialogue.

`-naive` Perform a naïve translation of an input VSDG into a PDG.

`-full` Perform an effective translation of the above.

`-dot` Produce a `dot` file output for the resulting PDG.

`-pdg` Produce a `pdg` file output for the resulting PDG.

`-sgc` Check the resulting PDG for well-formedness conditions and notify the user of any problems. These well-formedness conditions are described in this chapter.

`-stats` Display PDG statistics. These include the number of nodes, number of edges in E_C and E_D, and so on.

The naïve translation does not support VSDGs with loops (it quits with a message if looping constructs are detected), but the effective translation does provide this support. An outline of the `proc` tool is given in Figure 5.2. All PDG diagrams in this chapter are produced by the `dot` output of the `proc` tool.

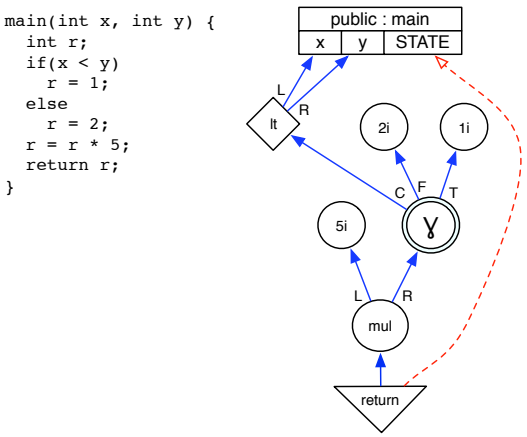


Figure 5.3: An example program and the VSDG produced by the VECC compiler.

5.4 Lawrence's naïve approach

Lawrence's naïve implementation consists of several steps. The first of these steps translates each node in the VSDG into a PDG statement node. This begins by assigning a virtual register to every operation node in the VSDG. For example, a VSDG `sub` node is translated into a PDG statement node labelled with, for example, $\mathtt{r_1} = \mathtt{r_2} - \mathtt{r_3}$ where $\mathtt{r_1}$ is the virtual register assigned to the `sub` node and $\mathtt{r_2}$ and $\mathtt{r_3}$ are the virtual registers assigned to the operands of `sub`. Next, predicate nodes are generated. These correspond to the γ-nodes in the VSDG. In our implementation we split any tupled γ-nodes into individual ones before proceeding. Each γ-node g with a predicate operand in virtual register r_p is translated into a PDG predicate node testing r_p and two group nodes for true and false children respectively. Given the notation that $\overrightarrow{rn}$ indicates the virtual registers assigned to the results of a node n, PDG statement nodes $\overrightarrow{rg} = \overrightarrow{r_t}$ and $\overrightarrow{rg} = \overrightarrow{r_f}$ are generated which assign virtual registers to the result of the γ-node for true and false paths.

With the PDG statement and predicate nodes generated and assigned virtual registers, the next step is to create the CDG that places them in the correct place in the control hierarchy. Lawrence's naïve construction works on the principle that if some statement S is control dependent on some group node G, then all children of S should also be dependent on G. The strategy chosen by the naïve construction is to traverse the VSDG from the return node, using γ-nodes as markers to indicate the next group node to link statements to. Thus, all VSDG nodes that are reachable from the return node are

[1] i.e. that first minimises the number of runtime evaluations, *then* program size – parallelling definitions of Optimal Code Motion due to Knoop et al. [78]

made control dependent on the entry node (root) of the PDG. When a γ-node is reached, the corresponding $\overrightarrow{rg} = \overrightarrow{r_t}$ and $\overrightarrow{rg} = \overrightarrow{r_f}$ PDG statement nodes are made control dependent on the corresponding PDG predicate node true and false group node children.

Then, all dependency edges from the VSDG are copied across to the corresponding PDG nodes, and both the heads and tails of these edges are moved up the CDG hierarchy until they are between siblings (i.e. they share the same immediate parent node). In Figure 5.4 we show the result of performing our implementation of the naïve translation on the VSDG in Figure 5.3. Statement nodes are represented by rectangular boxes. Predicate nodes are represented by diamonds. True and false edges from predicate nodes are labelled T and F respectively. Region nodes are represented by circles containing the letter R. In this example we can see how the γ-node in the VSDG is translated into a PDG predicate node. It can be seen that this approach works sufficiently well on simple VSDGs.

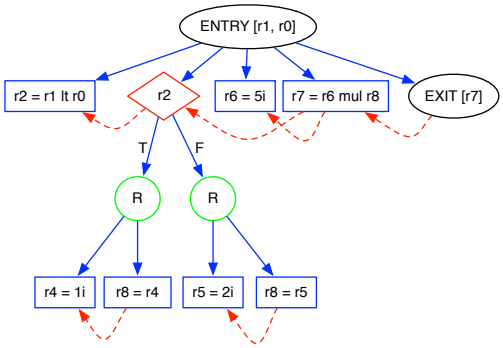


Figure 5.4: The PDG produced by applying the naïve algorithm to the VSDG in Figure 5.3.

However, this algorithm does not perform adequately in general. The problems with this approach are specified below:

Code size The resulting PDG sequentialises to an unnecessarily large CFG due to duplication of nodes.

Execution speed Duplication of nodes results in the CFG containing redundant operations which slows execution time of the resulting program.

Correctness The resulting PDG may not be well-formed – specifically, it may not be sequentialisable.

It is this last problem which is most serious. It can allow the formation of an illegal PDG subgraph, of which the canonical form is shown in Figure 5.6.

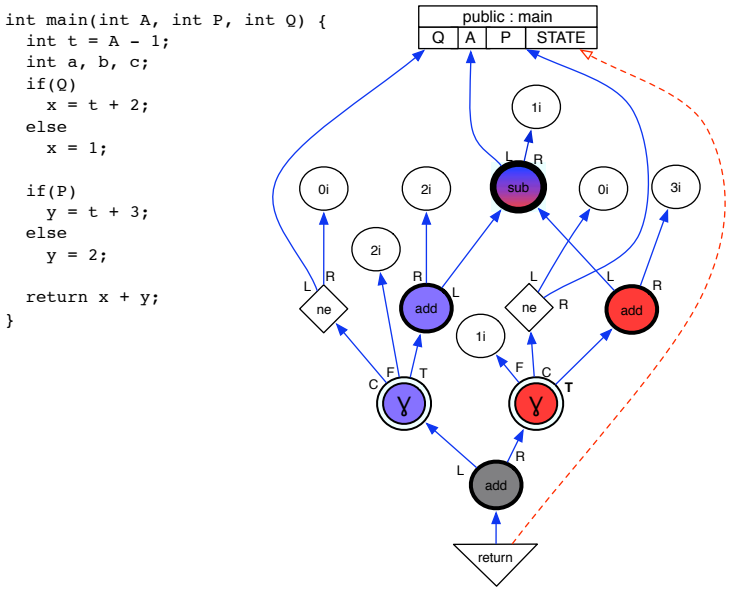


Figure 5.5: A VSDG for a program exhibiting independent redundancy. The grey `add` node demands the result of two separate γ nodes on its L and R port. Two different paths can use the result of the shaded `sub` node. One path is via the blue coloured nodes, and the other is via the red coloured nodes.

This makes the resulting PDG unsequentialisable, as an evaluation strategy has *not* been encoded.

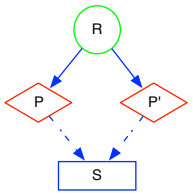


Figure 5.6: The canonical illegal PDG subgraph.

Figure 5.5 shows such a (legal) VSDG: here, in any execution the `sub` node may be demanded by both γ-nodes, by either one, or by none. Lawrence calls this situation *independent redundancy*. Applying the naïve implementation produces the unsequentialisable PDG of Figure 5.7. Thus, Lawrence states that an effective algorithm must *choose* an evaluation strategy in order to generate a sequentialisable PDG.

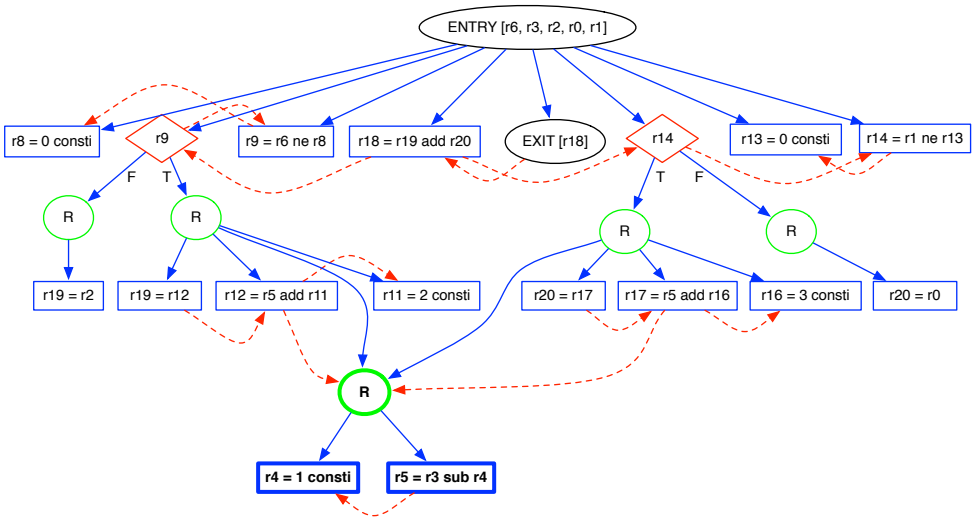


Figure 5.7: The unsequentialisable PDG produced by the naïve approach on the VSDG of Figure 5.5: the bold computation is shared. The illegal subgraph exists between the two predicate nodes testing `r9` and `r14`, and their respective children when the predicate evaluates to true.

5.5 Lawrence's effective algorithm

In order to avoid the problems with VSDGs exhibiting independent redundancy, Lawrence presents an algorithm that analyses the VSDG and uses a system of *gating conditions* in order to compute the demand conditions between nodes. While this approach can be seen as similar to that of Weise and Upton, the gating conditions ensure that, when independent redundancy exists, VSDG operation nodes that are shared between different γ-paths are identified. We provide the full algorithm from Lawrence's thesis in Appendix B for reference, edited to be applicable to standard VSDGs rather than Petri nets.

Figure 5.8 shows the application of γ-ordering to the independent redundancy example in Figure 5.5. Here, the input program returns the sum of two conditionals, each of which might (or might not) make use of the shared computation `int t = A - 1`. Thus, in the VSDG the returned value is the sum of the result of two γ-nodes, and there is no specification of when to evaluate the `sub`. Therefore, the algorithm applies γ-ordering, (arbitrarily) choosing the predicate test on `r4` (`P` in the source code) to be dominant and cloning the subsidiary test on `r9` (`Q`). This ensures a valid PDG is produced, and that the shared computation (highlighted in bold) is only computed *once* on any path.

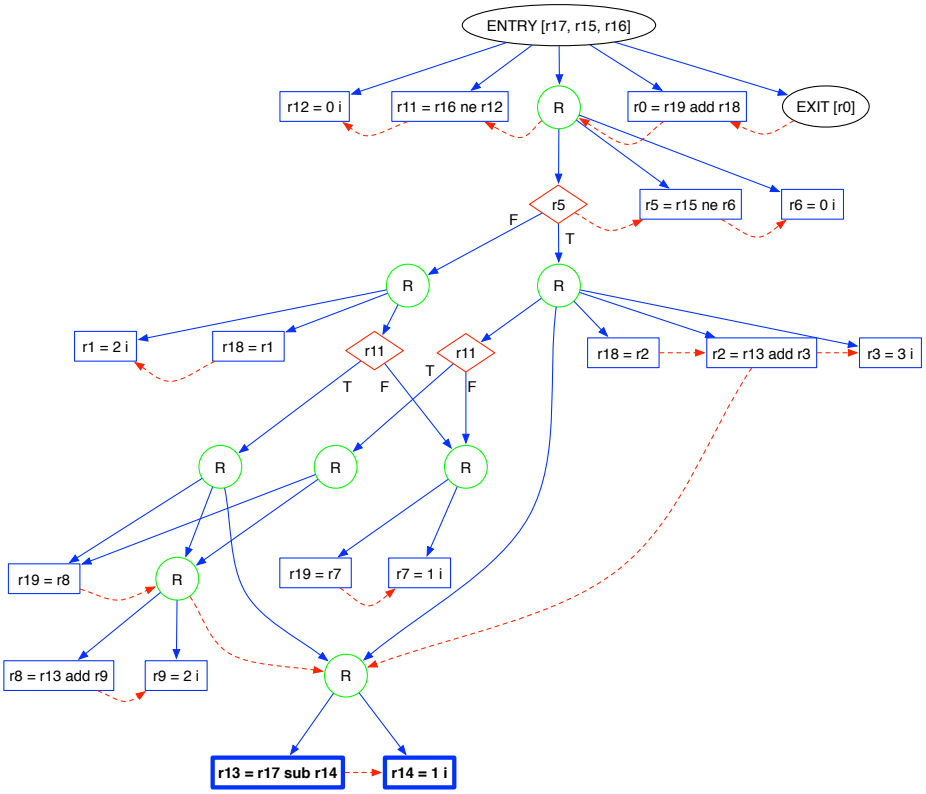


Figure 5.8: Application of γ-ordering to the independent redundancy example in Figure 5.5. The shared computation is in bold. Performing γ-ordering means that the shared computation occurs only *once* on each possible control flow path from `r4`, thus preventing redundant computations.

5.6 Compiling loops

The algorithms presented in Lawrence's thesis [83] operated on an acyclic VSDG. This is due to loops being represented as infinite regular chains of γ-nodes (Chapter 2). However, we are using Johnson's VECC compiler to produce VSDG input files which use a θ-node loop representation. This section outlines how we integrated this style of loop into the VSDG to PDG translation above.

5.6.1 θ-nodes

Recall from Chapter 2 the θ-node, which models the iterative behaviour of loops:

θ-node A θ-node $\theta(C, I, R, L, X)$ sets its internal value to initial value I.

> Then, while condition value C holds *true*, sets L to the current internal value and updates the internal value with the repeat value R. When C evaluates to $false$ computation ceases and the internal value is returned through the X port. [72]

Johnson uses θ-nodes to implement 0-trip loops (`while`, `for`) in his VSDGs. 1-trip loops (`do...while`, `repeat...until`) are left outside the scope of his thesis with the claim that these can be synthesised by code duplication, addition of Boolean flags, or augmentation of the loop semantics. The θ-node is cyclic. In order to work with the θ-node in Lawrence's algorithmic framework, we wish to translate it into an acyclic form. Johnson defines the G^{noloop} form, which is acyclic, as follows:

VSDG G^{noloop} form Given a VSDG, G, we define G^{noloop} to be identical to G except that each θ-node θ_i is replaced with two nodes, θ_i^{head} and θ_i^{tail}; edges to or from ports I and L of θ_i are redirected to θ_i^{head} and those to or from ports R, X, and C are redirected to θ_i^{tail}.

We implemented the translation to G^{noloop} form as a required pass in `proc`. This acyclic VSDG is suitable for input into a modified version of Lawrence's algorithm.

5.6.2 Loops in the PDG

PDG loops are defined by a back edge in the CDG. The simplest definition of a PDG loop is given by Ballance and Maccabe [20], where a conversion from abstract syntax to the structure of the CDG is given diagrammatically. We reproduce this diagram in Figure 5.9. Here, a loop is represented by a predicate node testing the loop condition P, with the body of the loop positioned under the true group node of the predicate, with a back edge in the CDG that transfers control to the beginning of the loop. The predicate does not need a false group node; if the predicate evaluates to $false$ then the loop is not executed.

This hierarchical structure of loops is very similar to the structure of conditional statements produced by Lawrence's framework, where a γ-node is translated into a PDG predicate node. The only difference is that no false group node is required, and a back edge is added to the parent group node of the predicate node from the true group child. We will firstly show a translation of loops based on this principle, and highlight why this doesn't work with particular loops with side-effecting guard conditions. Then, we will show how to solve this problem with a further transformation on the PDG.

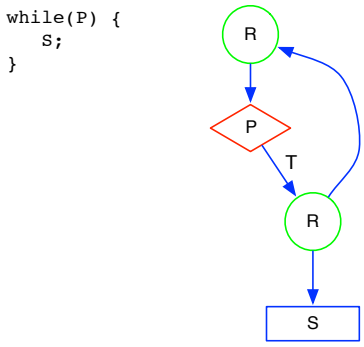


Figure 5.9: A `while` loop in abstract syntax translated into a loop in the CDG.

5.7 Adding loops to Lawrence's framework

To begin with, each θ^{head} and θ^{tail} node must be converted into corresponding PDG nodes. A $\theta^{tail}(C, R, X)$ node translates naturally into a PDG predicate node testing the virtual register of the C condition, with one group node for true children, which will represent the loop body. Since a back edge encloses a loop region in the PDG, we translate the θ^{head} node into a single group node. This is removed later by the normalisation pass during construction. We mark predicate nodes corresponding to θ^{tail} nodes so that after the algorithm completes we can locate them in the PDG and add in back edges from the true group node child to the parent group node of the predicate. Additionally, during the virtual register assignment phase, any operations with operands coming from the L port of the θ^{head} node get their virtual register from the corresponding operation node connected to the θ^{head} I port. Similarly, any operations with operands coming from the X port of the θ^{tail} node get their virtual register from the corresponding operation node connected to the R port. This look-up works within nested loops also. In addition, we tag each predicate node corresponding to a loop with the θ-node I and R port registers of each value using this look-up mechanism. This information is used during code generation.

We do not need to add any extra gating conditions to Lawrence's framework; we only have to modify the behaviour of the $cond(e)$ function that gives a gating condition for any $n \to n'$ edge:

$$cond(e) = \begin{cases} \langle ? \rangle(g, \Lambda, \emptyset), & \text{if } e \text{ is a true edge from a } \gamma\text{-node } g \\ \langle ? \rangle(g, \emptyset, \Lambda), & \text{if } e \text{ is a false edge from a } \gamma\text{-node } g \\ \langle ? \rangle(t, \Lambda, \emptyset), & \text{if } e \text{ is a } R \text{ edge from a } \theta^{tail}\text{-node } t \\ \Lambda, & \text{otherwise} \end{cases}$$

Thus, the $\langle ? \rangle$ gating condition now represents both γ-nodes and θ^{tail}-nodes.

In the context of loops, the Λ gating condition represents the execution of the loop body along the R port, and enforces the ordering of the loop body statements under the `true` group node of the PDG predicate node.

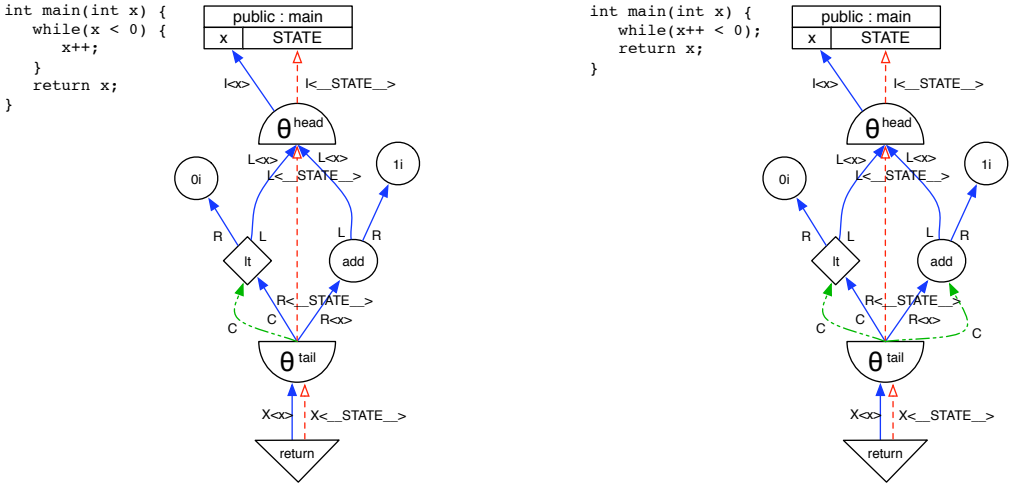
However, loops are different from conditional statements in that the guard condition is evaluated multiple times due to the back edge. This means that the registers that the guard conditions test must be kept updated with the most recent value of the corresponding variable in the loop. This is implemented as a pass once the PDG has been built. For each loop in the PDG, the statement node computing the guard condition is identified. Then, for all the statements in the loop body (i.e. those that are children of the `true` group node), we check the virtual register that is the left-hand operand. If this matches either of the virtual registers on the right-hand side of the guard computation, then a new statement node is inserted into the body that assigns the result of this computation into the matching virtual register in the guard.

This technique works effectively for loops that do not have side-effecting guard conditions. However, when a side-effecting guard condition is present, it runs into problems. In Figure 5.10, we show two C programs containing a loop, and their corresponding VSDGs. The first program has a non-side-effecting guard condition, but the second has. Each of these two VSDGs produced by the VECC compiler have identical value and state edges, but *different* serial edges (drawn as green dashed arrows). The serial edges ensure that the `add` is evaluated at the same time as the guard in the second VSDG. However, up to this point, with Lawrence's framework and our naïve loop translation, we have ignored serial edges. Thus, this means that the two semantically *different* VSDGs translate to the *same* PDG 5.10b, which is incorrect.

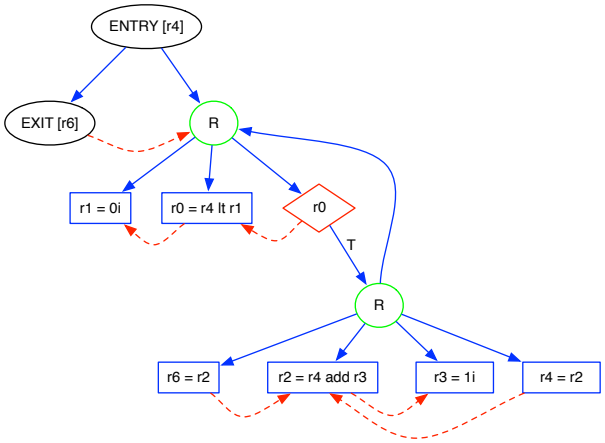
We note the difference between the two VSDGs in Figure 5.10. In the non-side-effecting example, one serial edge with the label C is present between the θ^{tail} node and and the `lt` node. In the side-effecting example, an additional serial edge, also labelled C, is present between the θ^{tail} node and the `add` node. In the context of the VECC compiler, these enforce a particular evaluation order of the graph; that is that in the second example we must also evaluate the `add` during evaluation of the θ_{tail} C condition, thus enforcing the side-effecting behaviour of the guard. In the next section, we will show how we can use these serial edges to transform the PDG to enforce the same behaviour, avoiding the problems with the translation in Figure 5.10.

5.7.1 Using serial edges to transform the PDG

With the translation in Figure 5.10, the side-effecting guard statements are incorrectly placed under the T group node of the loop predicate in the PDG. This is incorrect as they need to be computed at the same time as the guard condition. Therefore, we need to identify the required statement nodes representing the side-effect of the guard and hoist them out of the loop body, making them a sibling of the computation of the condition.



(a)



(b)

Figure 5.10: Two semantically different VSDGs and (a) which translate to the same PDG (b) when serial edges are ignored.

To begin with, we keep record of the C serial edges that come from the θ^{tail} node. Then, for each of these edges pointing at VSDG nodes, we find the corresponding PDG nodes, and copy the edges across to the PDG; we call these *hoist edges*. In Figure 5.11a we show the PDG with annotated hoist edges (also drawn as green dashed arrows) for the second VSDG in Figure 5.10a. Each hoist edge points at the statement node that needs to be hoisted, however that statement node may be data dependent on other nodes in order to compute. Therefore, for each node that needs to be hoisted, we do a depth-first search along the DDG to identify the nodes it is dependent on. Then, if not already, this group of nodes are then made control dependent on the parent node of the loop predicate (the *source* of the hoist edge). This results in the correct PDG of Figure 5.11b. This pass is run on the PDG after it has been built by the modified version of Lawrence's algorithm in the `proc` tool.

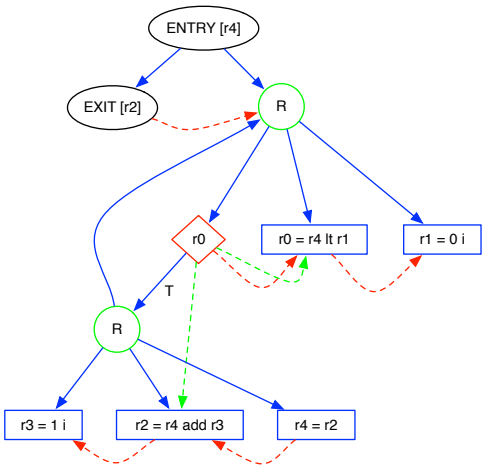
5.7.2 Handling `break` and `continue`

The keywords `break` and `continue` allow early exit from loops in C: immediate early termination in the case of `break` and the restart of the loop with `continue`. In Johnson's VSDG, `continue` and `break` are implemented as "special" nodes that modify the runtime execution of loops. These nodes "have the exact same value and state dependencies as the θ^{tail} node, and produce a new state". These special nodes are then used during code generation in order to generate jumps to the loop exit and loop entry labels respectively. In Figure 5.12 we show a sample C program containing a `break` node. Broadly speaking, the `break` and `continue` nodes specify which nodes to evaluate before they execute. In Figure 5.12, the `add` operation will be performed before the loop terminates early with `break`.

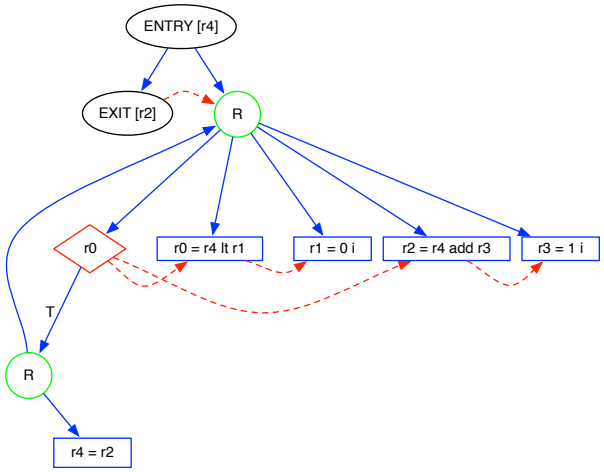
The VSDG implementation of `continue` is done in a similar manner. Since `break` and `continue` have specific meaning in code generation, rather than in the PDG itself, we translate them across as `break` and `continue` statement nodes in the PDG. We make the statement nodes data dependent on the value predecessors in the VSDG to ensure that, from a scheduling perspective, the `break` and `continue` are performed in the correct place. We show a PDG generated by `proc` for Figure 5.12 in Figure 5.13.

5.8 Worked example

To show the workings of the modified algorithm, we provide a worked example. We show the generation of a PDG containing nested loops by worked example on the VSDG in Figure 5.14. Since there are two loops, we have labelled the θ^{head} and θ^{tail} nodes with unique numbers for reference. The first stage of the algorithm involves assigning virtual registers for operation nodes in the VSDG and then generating PDG statement nodes for each (Section 5.4). The result of this phase is shown in Figure 5.15. Here, we can see the generated entry and



(a)



(b)

Figure 5.11: A PDG with (a) hoist edges annotated, and (b) after application of hoisting.

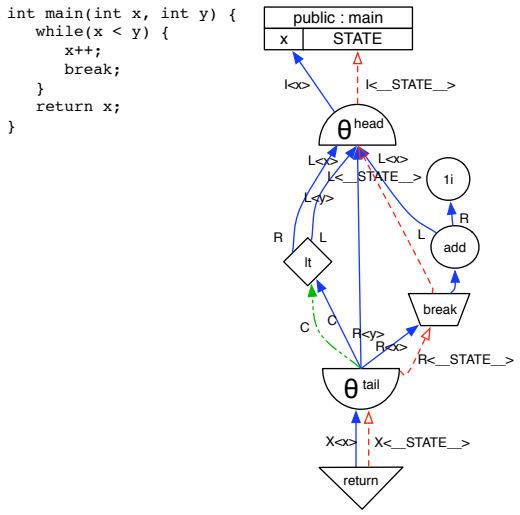


Figure 5.12: A C program containing a `break` node and the corresponding VSDG.

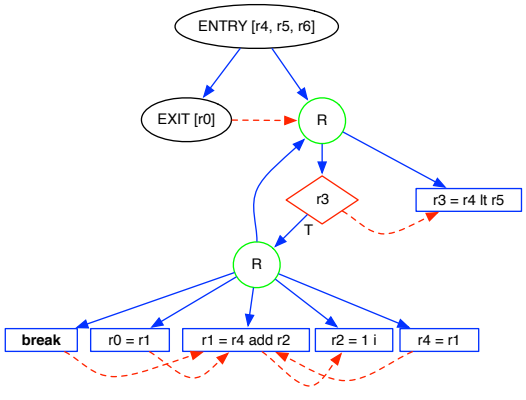


Figure 5.13: The PDG produced by `proc` for the VSDG in Figure 5.12.

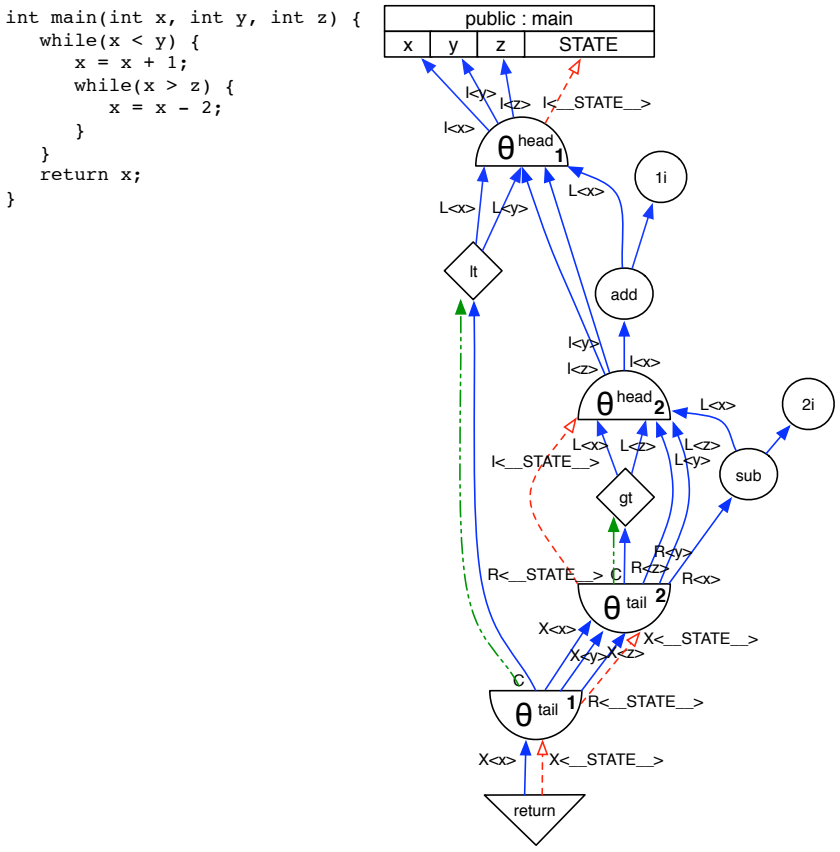


Figure 5.14: A VSDG containing a nested loop.

exit nodes, along with two predicate nodes for each θ^{tail}, two group nodes for each θ^{head}, and a collection of statement nodes corresponding to the operation nodes in the VSDG.

After the postdominator tree has been calculated for the VSDG, the `buildPDG` function (Appendix B) is called on the root node, which is the `return` node in the VSDG. We use $P(A)$ to represent the PDG fragment generated for A.

1. The edge `return` $\rightarrow \theta_1^{tail}$ is processed, yielding $C^{\texttt{return}}(\theta_1^{tail}) = \Lambda$.
2. `buildPDG` recurses on θ_1^{tail}.
   a) All edges $\theta_1^{tail} \rightarrow \theta_2^{tail}$ yield $C^{\theta_1^{tail}}(\theta_2^{tail}) = \langle ? \rangle(\theta_1^{tail}, \Lambda, \emptyset)$.
   b) The edge $\theta_1^{tail} \rightarrow$ `lt` yields $C^{\theta_1^{tail}}(\texttt{lt}) = \Lambda$.
3. `buildPDG` recurses on `lt`.
   a) Both edges `lt` $\rightarrow \theta_1^{head}$ yield $C^{\texttt{lt}}(\theta_1^{head}) = \Lambda$.

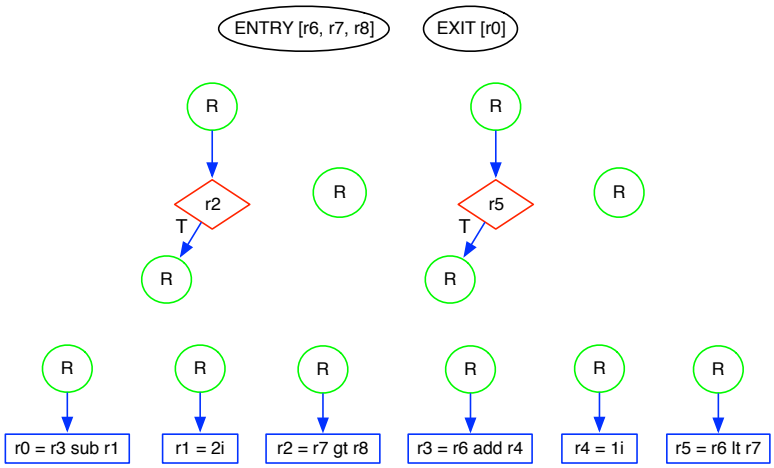


Figure 5.15: Unconnected PDG nodes with virtual registers assigned from the VSDG in Figure 5.14.

4. `buildPDG` recurses on θ_2^{tail}.
   a) The edges $\theta_2^{tail} \to \theta_2^{head}$ yield $C^{\theta_2^{tail}}(\theta_2^{head}) = \langle ? \rangle(\theta_2^{tail}, \Lambda, \emptyset)$.
   b) The edge $\theta_2^{tail} \to$ `sub` yields $C^{\theta_2^{tail}}(\texttt{sub}) = \langle ? \rangle(\theta_2^{tail}, \Lambda, \emptyset)$.
   c) The edge $\theta_2^{tail} \to$ `gt` yields $C^{\theta_2^{tail}}(\texttt{gt}) = \Lambda$.
5. `buildPDG` recurses on `sub`.
   a) The edge `sub` $\to$ `2i` yields $C^{\texttt{sub}}(\texttt{2i}) = \Lambda$.
   b) The edge `sub` $\to \theta_2^{head}$ yields $C^{\texttt{sub}}(\theta_2^{head}) = \Lambda$.
6. `buildPDG` recurses on `2i`, but this has no predecessors. Thus `buildPDG`(`2i`) returns the fragment containing `r1` = `2i` and the set of external producers $^{\bigstar}D(\texttt{2i}) = \emptyset$.
7. `link`$(P(\texttt{2i}), \Lambda, P(\texttt{sub}))$ is called, resulting in a new $P(\texttt{sub})$ as shown in Figure 5.16a.
8. `buildPDG` recurses on `gt`.
   a) Both edges `gt` $\to \theta_2^{head}$ yield $C^{\texttt{gt}}(\theta_2^{head}) = \Lambda$.
9. `buildPDG` recurses on θ_2^{head}.
   a) The edge $\theta_2^{head} \to$ `add` yields $C^{\theta_2^{head}}(\texttt{add}) = \Lambda$.
   b) The edges $\theta_2^{head} \to \theta_1^{head}$ yield $C^{\theta_2^{head}}(\theta_1^{head}) = \Lambda$.

10. `buildPDG` recurses on `add`.
    a) The edge `add` $\rightarrow$ `1i` yields $C^{\texttt{add}}(\texttt{1i}) = \Lambda$.
    b) The edge `add` $\rightarrow \theta_1^{head}$ yields $C^{\texttt{add}}(\theta_1^{head}) = \Lambda$.
11. `buildPDG` recurses on `1i`, but this has no predecessors. Thus `buildPDG`(`1i`) returns the fragment containing `r4` = `1i` and the set of external producers $\bigstar D(\texttt{1i}) = \emptyset$.
12. `link`$(P(\texttt{1i}, \Lambda, P(\texttt{add}))$ is called, resulting in a new $P(\texttt{add})$ as shown in Figure 5.16b.
13. `link`$(P(\theta_2^{\texttt{head}}), \Lambda, \texttt{add})$ is called, linking a single group node with no children to the fragment $P(\texttt{add})$.
14. Edges in $\bigstar D(\texttt{add})$ are updated, remaining $C^{\texttt{add}}(\texttt{1i}) = C^{\texttt{add}}(\theta_1^{head}) = \Lambda$.
15. `link`$(P(\theta_2^{tail}), \langle ? \rangle(\theta_2^{tail}, \Lambda, \emptyset), P(\texttt{sub}))$ is called. Recursively, this calls `link`$(P(\theta_2^{tail}).\mathsf{true}, \Lambda, P(\texttt{sub}))$, where $P(\theta_2^{tail}).\mathsf{true}$ is the `true` child of the predicate node representing the θ_2^{tail} loop, thus linking the loop body underneath the loop header. `link`$(P(\theta_2^{tail}).\mathsf{false}, \emptyset, -)$ does nothing; there is no `false` child required in loops.
16. Edges in $\bigstar D(\texttt{sub})$ are updated. $C^{\texttt{sub}}(\texttt{2i}) = C^{\texttt{sub}}(\theta_2^{head}) = \langle ? \rangle(\theta_2^{tail}, \Lambda, \emptyset)$.
17. `link`$(P(\theta_2^{tail}), \Lambda, P(\texttt{gt}))$ is called, connecting the loop condition PDG fragment to the parent group node of the loop predicate in the fragment $P(\theta_2^{tail})$.
18. The edges in $\bigstar D(\texttt{gt})$ are updated, remaining as $C^{\texttt{gt}}(\theta_2^{head}) = \Lambda$.
19. `link`$(P(\theta_2^{tail}), \Lambda, P(\theta_2^{head}))$ is called, resulting in a new $P(\theta_2^{head})$.
20. The edges in $\bigstar D(\theta_2^{head})$ are updated, remaining $C^{\theta_2^{head}}(\theta_1^{head}) = C^{\theta head_2}(\texttt{1i}) = \Lambda$.
21. `buildPDG` recurses on θ_1^{head}.
    a) All edges $\theta_1^{head} \rightarrow$ `main` yield $C^{\theta_1^{head}}(\texttt{main}) = \Lambda$.
22. `link`$(P(\theta_1^{head}), \Lambda, P(\texttt{main}))$ is called, connecting the entry node to the graph.
23. `link`$(P(\theta_1^{tail}), \Lambda, P(\texttt{lt}))$ is called, connecting the outer loop condition to the parent of the loop predicate.
24. The edges in $\bigstar D(\texttt{lt})$ are updated, remaining as $C^{\texttt{lt}}(\theta_1^{head}) = \Lambda$.

25. $\texttt{link}(P(\theta_1^{tail}), \langle\!?\!\rangle(\theta_1^{tail}, \Lambda, \emptyset), P(\theta_2^{tail}))$ is called. Recursively, this calls $\texttt{link}(P(\theta_1^{tail}).\mathsf{true}, \Lambda, P(\theta_2^{tail}))$ where $P(\theta_1^{tail}).\mathsf{true}$ is the true child of the predicate node representing the θ_1^{tail} loop, thus linking the loop body underneath the loop header. $\texttt{link}(P(\theta_1^{tail})).\mathsf{false}, \emptyset, -)$ does nothing; there is no false child required in loops. The result of this link is shown in Figure 5.16c.

26. Edges in ${}^{\bigstar}D(\theta_2^{tail})$ are updated, with $C^{\theta_2^{tail}}(\texttt{2i}) = \langle\!?\!\rangle(\theta_2^{tail}, \Lambda, \emptyset)$, $C^{\theta_2^{tail}}(\theta_1^{head}) = \Lambda$ and $C^{\theta_2^{tail}}(\texttt{1i}) = \langle\!?\!\rangle(\theta_1^{tail}, \Lambda, \emptyset)$.

27. $\texttt{link}(P(\theta_1^{tail}), \Lambda, P(\theta_1^{head}))$ is called, linking a single group node with no children to the fragment $P(\theta_1^{tail})$.

28. Edges in ${}^{\bigstar}D(\theta_1^{head})$ are updated, remaining $C^{\theta_1^{head}}(\texttt{main}) = \Lambda$.

29. $\texttt{link}(P(\texttt{return}), \Lambda, P(\theta_1^{tail}))$ is called, linking the exit node to the parent node of the outer loop predicate.

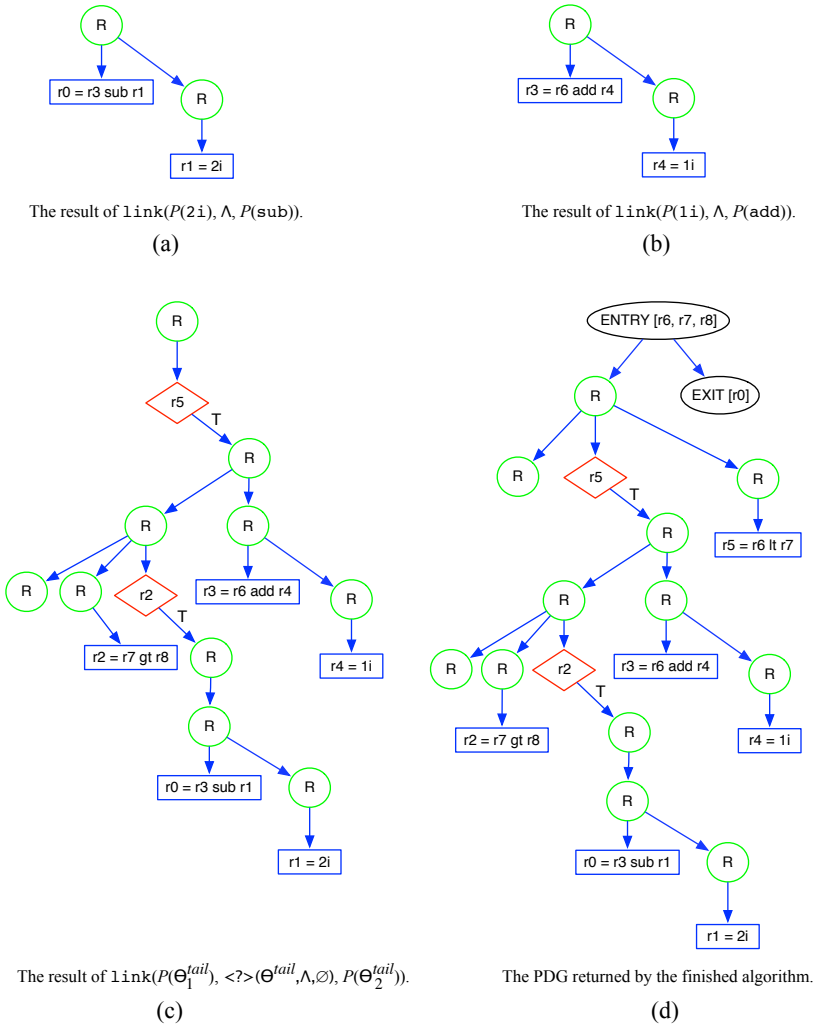
30. Edges in ${}^{\bigstar}D(\theta_1^{tail})$ are updated, remaining $C^{\theta_1^{tail}}(\texttt{2i}) = \langle\!?\!\rangle(\theta_2^{tail}, \Lambda, \emptyset)$, $C^{\theta_1^{tail}}(\texttt{1i}) = \langle\!?\!\rangle(\theta_1^{tail}, \Lambda, \emptyset)$ and $C^{\theta_1^{tail}}(\texttt{main}) = \Lambda$.

31. The algorithm finishes, returning the completed PDG for the function `main`, shown in Figure 5.16d.

Some additional passes are then run on the PDG. Firstly, a normalisation pass runs that merges any group nodes with only one parent into that parent to ensure well-formedness. Back edges are added between loop predicate true group nodes and the parent group node of that predicate, as described in Subsection 5.6.2. Following this, a pass inserts the additional required assignments into loop bodies to ensure that guard conditions are correct. Then, the DDG is built using the following principle: for a statement node `r3 = r1 add r2`, data dependence edges are added to the statement nodes producing `r1` and `r2`. This methodology is applied to the whole PDG. Then, the endpoints of these edges are moved up between siblings in the CDG. We implemented this edge moving algorithm by extending the Lowest Common Ancestor [9, 60] algorithm, using it to find the group node parent of the two subtrees containing each data dependence edge endpoint. Then, we run the loop guard hoisting algorithm described in Subsection 5.7.1. However, in this example, no loops have side-effecting conditions, so no hoisting is performed. The finished PDG for the example is shown in Figure 5.17.

5.9 Summary

This chapter has explored the existing problems with generating sequential code from the VSDG and similar graphs. While previous attempts to restore



The result of `link`(P(`2i`), Λ, P(`sub`)).

(a)

The result of `link`(P(`1i`), Λ, P(`add`)).

(b)

The result of `link`($P(\Theta_1^{tail})$, <?>(Θ^{tail},Λ,∅), $P(\Theta_2^{tail})$).

(c)

The PDG returned by the finished algorithm.

(d)

Figure 5.16: The results of various link operations during the proceduralisation algorithm.

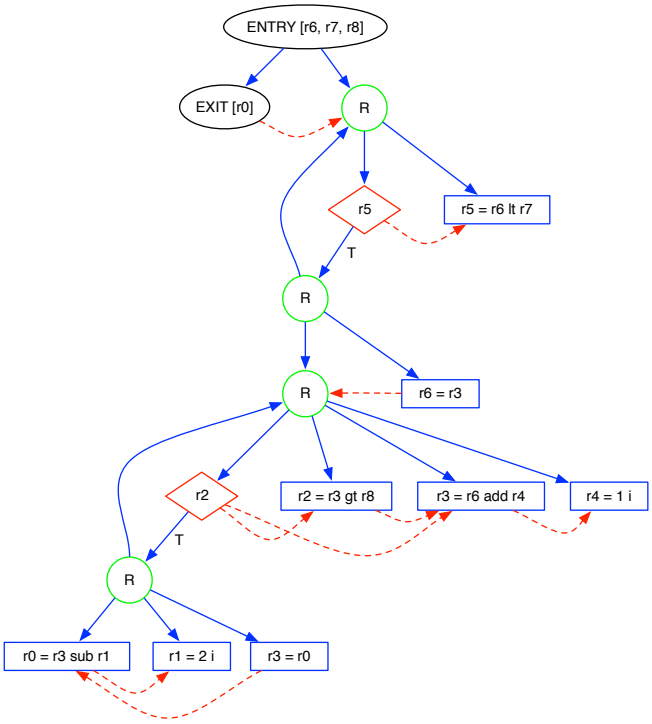


Figure 5.17: The finished PDG after proceduralisation, normalisation, and building of the DDG for the VSDG in Figure 5.14.

control flow have fallen short, Lawrence's algorithm, which translates a VSDG into a PDG whilst encoding a lazy strategy in the manner of Upton, showed promise. We proved, by means of an implementation, that Lawrence's strategy works for loopless VSDGs, and showed how γ-ordering occurs in the implementation. We then modified the algorithm to allow for θ-node loops, thus bridging the gap between Johnson and Lawrence's work. We also showed how serial edges are used to ensure that side-effecting guard conditions are ordered correctly in the PDG. We posit that the longstanding problem of sequentialising the VSDG is now solved.

Chapter 6

Sequentialisation

6.1 Introduction

After applying the proceduralisation algorithm in Chapter 5 to the Value State Dependence Graph (VSDG), a Program Dependence Graph (PDG) is generated. The PDG is a parallel IR that has been mentioned frequently in the literature, but has had little usage in mainstream compilers (see Chapter 2).

In this chapter we explore scheduling and code generation from the PDG. Since the PDG is a parallel representation, an ordering on instructions must be chosen for the generated code to execute on a single program counter. This process has commonly been called *sequentialisation* in the literature. We examine the literature for previous attempts to perform sequentialisation on a PDG, highlighting how it has been problematic. We show how the problem becomes more straightforward as a result of the proceduralisation algorithm in Chapter 5, and then outline our approach and implementation of sequentialisation, resulting in sequential code.

This chapter presents the following:

- We outline the problem of generating sequential code from the PDG, and examine the literature for previous attempts to solve this problem. We explain how Lawrence's proceduralisation algorithm makes sequentialisation more straightforward.
- We present our software tool, `seq`, which translates PDGs from the `proc` tool into sequential program code.
- We outline our scheduling algorithm for the PDG.
- We demonstrate output from the `seq` tool via a human-readable pseudocode format and also LLVM bitcode.

6.2 Previous work

Much of the early work with the PDG focused on construction and performing analyses and transformations. To begin with, little attention was paid to the problem of translating *out* of the PDG. This was because the PDG was often built as an auxiliary IR—e.g. to perform program slicing [91]—rather than the primary IR, so the problem was less important. In the original paper by Ferrante et al. [57], neither scheduling nor code generation are mentioned. The PDG was developed for the parallelisation opportunities it offered, and in subsequent literature, the target code was not for sequential processors. For example, the IBM PTRAN project [11] used the PDG to translate sequential FORTRAN programs into code that ran on parallel architectures.

The PDG is a natural representation for parallel programming. However, in this thesis, we have seen that an effective approach to restore control flow to the VSDG involves building a PDG to represent the program in parallel. Therefore, this leaves us with the need to transform this PDG into sequential code in order to finish a VSDG compiler framework for a sequential processor.

The initial work on generating sequential code from a PDG was done by Ferrante and Mace [55]. Here, the authors consider a FORTRAN-like language that contains statements, predicates and `goto` operators. In this paper, the authors acknowledge that there are particular programs, when represented as PDGs, that require duplication of nodes and edges before they can be translated into a sequential form. An algorithm is presented that can generate sequential code without duplication of statements for PDGs where a single corresponding CFG is *guaranteed to exist*. No detail is given on how to handle PDGs that do not have a guaranteed single corresponding CFG, nor are the conditions for detecting a sequentialisable PDG given.

A follow up paper [56] presents a method for determining whether extra guard variables or duplicate code must be inserted in order to guarantee a single corresponding CFG. This paper contains the first explicit mention of the illegal subgraph in Figure 5.6. Then, a more detailed algorithm is given for generating sequential code when a guaranteed corresponding CFG exists. When an illegal subgraph is detected, the authors conjecture that generating the *minimal size* CFG is an NP-complete problem, showing a constrained example that reduces to 3-SAT. Two years later, an oversight was spotted in the previous two papers, and a corrected algorithm for sequentialising PDGs with acyclic CDGs was given [107] (i.e. loop-free PDGs). A sketch is presented showing how to sequentialise reducible loops. An alternative approach [108] was also taken by the same authors, which uses preprocessing to make information about external edges entering subgraphs available prior to sequentialisation. Again, the handling of loops is sketched and a method for detecting PDGs requiring duplication is outlined, but no concrete algorithm is given for sequentialising either; this was left to a technical report which never surfaced. Steensgaard [112] extends the algorithm to handle irreducible PDGs.

The most prominent observation is that all of these attempts acknowledge that sequentialisation becomes more difficult in the presence of illegal PDG subgraphs. So far, in our compiler framework, we have begun by using the VSDGs produced by Johnson's VECC compiler, and then implemented Lawrence's proceduralisation algorithm. As seen in Chapter 5, the VSDG is analysed using gating conditions, which are in turn used to guide the construction of the PDG. The disjunction ($\oplus$) gating condition identifies nodes that may be demanded via paths from more than one γ node: either by both, one, or none. Recall that Lawrence called this situation *independent redundancy*. When the PDG is being built and independent redundancy is detected, one predicate node in the PDG is selected as dominant and the others are selected as subsidiary, and an arbitrary ordering is selected, duplicating the subsidiary tests. This enforces an evaluation strategy, and more importantly, prevents any illegal subgraphs from occurring in the corresponding PDG. Since this is the case, we do not have to deal with illegal subgraphs when scheduling the PDG, making the process more straightforward as a result.

6.3 The `seq` tool

This chapter describes a software tool called `seq` for sequentialising a PDG. The `pdg` files that are generated by the `proc` tool are used as input to `seq`. A handwritten parser for the `pdg` files, using the grammar in Appendix C, forms the front-end of the tool. Once the `pdg` file has been read in, a scheduling algorithm decides on an ordering of the instructions. Then, code is generated into either human-readable pseudocode or LLVM bitcode. The tool is written in C++ using only standard library functions.

The tool is invoked as follows:

```
seq [-help] [-s|-p|-l] [optional flags] inputfile
```

where `inputfile` is a valid `pdg` file. An outline of the `seq` tool is given in Figure 6.1. The options are as follows:

`-help` Display the help dialogue.

`-s` Output the code in a human-readable pseudocode to the standard output.

`-p` Output the code in a human-readable pseudocode to a `.pse` file.

`-l` Output the code in human-readable `llvm` bitcode to a `.ll` file.

`-dot` Print out a `.dot` file representation of the input PDG.

`-stats` Display PDG statistics.

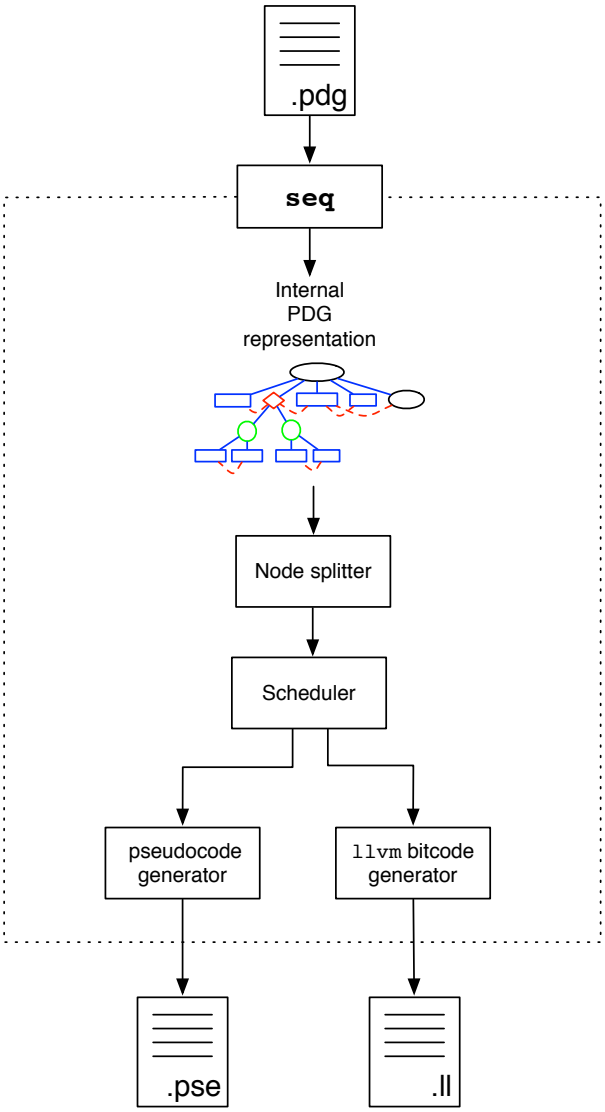


Figure 6.1: An outline of the `seq` tool.

6.4 Node splitting

As seen in Chapter 5, analysing the VSDG with gating conditions can detect instructions that are shared down multiple control flow paths. When building the PDG for shared instructions, the subsidiary predicate nodes are duplicated and ordered in order to prevent redundant computations on any of these paths, and also to prevent illegal subgraphs from occurring. This duplicating and ordering process results in PDG statement nodes having multiple control parents, which causes difficulty for existing scheduling algorithms. Notably, Simons et al. [108] state that, for their scheduling algorithm to work, region nodes must be the only type of node in the control dependence graph that two control paths can merge. Intuitively, this causes a problem for any kind of graph walking algorithm intended to generate code. A statement node with n control dependence parents signifies that it is executed n times in the generated code. Graph walking algorithms, such as depth-first search, traditionally use a list of visited nodes in order to prevent repeated visits. To allow for repeated visits when walking the PDG, and also to conform to the PDG well-formedness conditions of Simons et al. [108], we must apply a node splitting transformation. We outline this transformation in Algorithm 13.

Algorithm 13: Node splitting algorithm to restore well-formedness to the PDG.

Input : The root node of the PDG: n
Output: A well-formed PDG.

```
Mark(n);
forall the control dependence children in BFS order c do
    if controlParents(c) > 1 then
        max ← controlParents(c);
        i ← 1;
        while i < max do
            p ← controlParents(c).at(i);
            clone ← Clone(c);
            SplitControlEdges(p, c, clone);
            max ← controlParents(c);
SplitDataEdges();
```

This algorithm calls three helper functions:

`Clone(c)` Given a node c, this function returns a duplicate of that node. It also tags the duplicated node with a reference to the original node it was cloned from.

`SplitControlEdges(p, c, clone)` Whenever a node is split, this function updates the control edge (p, c) to become $(p, clone)$.

`SplitDataEdges()` After all nodes have been split, the data dependence edges must also be updated. This function iterates over all cloned nodes and checks, for each, if any data dependence edges point to or from the tagged original. If so, then these data dependence edges are updated to point to or from the clone.

Splitting increases the size of the graph in memory. For each node with $n > 1$ control dependence parents, n duplicates are created. However, this penalty is only incurred at compile time. Additionally, the scheduling algorithm given by Simons et al. requires duplication of all of the nodes in the graph before assigning edges to create a schedule [108], so duplication of nodes while working with the PDG is not uncommon, so we feel that it is an acceptable solution.

In Figure 6.2 we show the PDG for the independent redundancy example, seen previously in Figure 5.5, before and after the node splitting algorithm is applied. The split PDG can now be used as input into a simple recursive scheduling algorithm.

6.5 Scheduling

In this section we will show our approach to scheduling the PDG. In the next section, we will show what modifications need to be made to the schedule in order to generate sequential code. Given a PDG, we need to decide on a sequential ordering of the instructions. Recall the two different types of edges in the PDG:

Control dependence A control dependence edge (a, b) means that once a has been executed, b is the next node to be considered for execution. a may have multiple outgoing edges.

Data dependence A data dependence edge (a, b) means that node a requires the result of b in order to be executed. Therefore, b should be executed before a. Recall from Chapter 5 that Lawrence's algorithm specifies that data dependence edges should be moved up the PDG hierarchy so that they are between siblings. As a result of this, a data dependence edge in our PDG means that a requires the subgraph, of which b is the root node, to have been executed before it can be executed itself.

Scheduling of the PDG begins at the entry node (root). From here, control dependence edges specify a number of instructions to be scheduled next. The order of these instructions depends on their data dependencies. Our scheduling algorithm is recursive, and begins at the root. We present the basic scheduling algorithm in Algorithm 14, which is a modified depth-first search. The exact

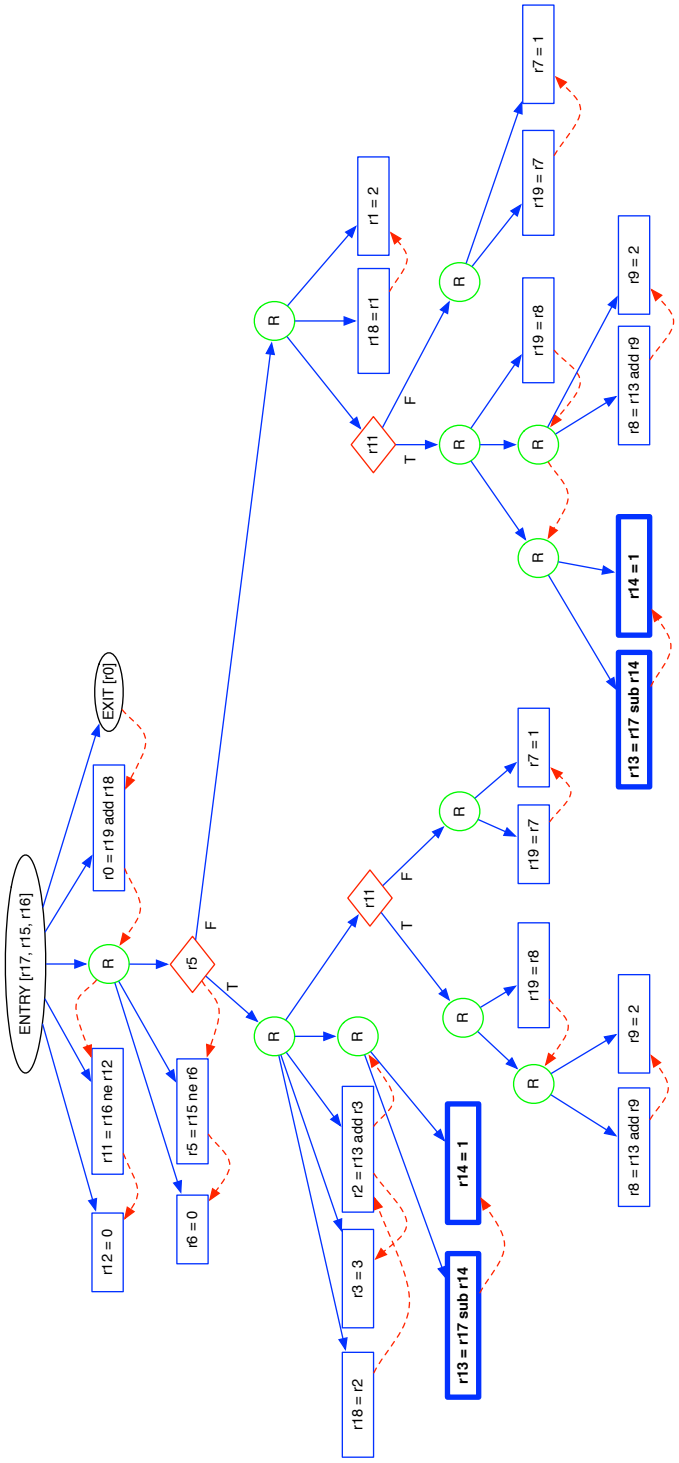


Figure 6.2: Application of node splitting to the independent redundancy example PDG. The (previously) shared `sub` computation is in bold. In contrast to Figure 5.8, the computation now exists on two separate control dependence regions.

ordering of control dependence children and data dependence children is arbitrary. The `Mark` function adds a node to the marked list. The `Number` function adds a node to the schedule list.

Algorithm 14: Recursive scheduling algorithm (`Schedule`) for a PDG with no illegal subgraphs.

Input : The root node of the PDG: n
Output: A numbered schedule for the PDG.

```
Mark(n);
forall the Data dependence children d do
    if d is not marked then
        Schedule(d);
Number(n);
if n is a predicate node then
    Schedule(true);
    Schedule(false);
forall the Control dependence children c do
    Schedule(c);
```

We demonstrate the above algorithm on a PDG in Figure 6.3. The C program given has been translated into a PDG by the `proc` tool, and the ordering is given by our `seq` tool. Region nodes in the PDG are added to the schedule, but do not generate instructions, as they do not have an associated operation; they merely group instructions together with the same control dependencies. Now that we have a schedule decided for the PDG instructions, we can look at how the scheduled PDG nodes map to sequential code.

6.6 Towards sequential code

Broadly speaking, our PDGs represent three different types of programming construct. The first is simple statements, the second is conditional branches (predicate nodes), and the third is loops (predicate nodes with back edges). We will look at the structure of sequential code that needs to be generated for each of these. The `seq` tool provides a human-readable pseudocode output for our PDGs, and we make use of that in this section. For now, we continue to use virtual registers in the code. Also, at this point, the scheduling algorithm will have completed, so we have access to a list of PDG nodes, where the position in this list is the numbering in the schedule.

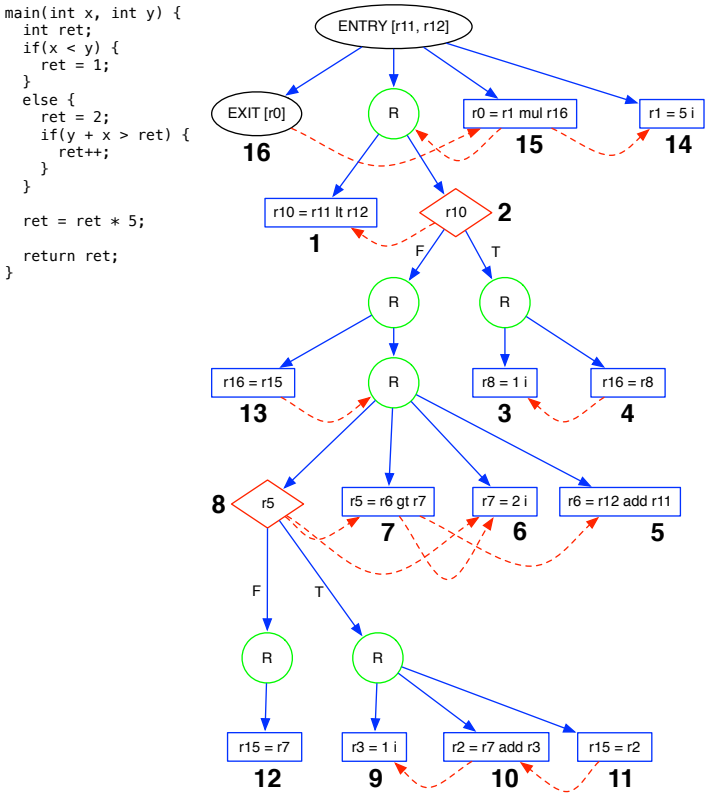


Figure 6.3: Application of the scheduling algorithm to a PDG. The order in the schedule is annotated next to each node.

6.6.1 Statements

Given some number of statements that are contiguous in the schedule, they can simply be output one after the other with respect to their order. For example, in Figure 6.3, statements 5, 6 and 7 become:

```
5: r6 = r12 add r11
6: r7 = 2i
7: r5 = r6 gt r7
```

where each instruction is labelled with its number in the schedule. No special action is required when a statement node is encountered.

6.6.2 Conditional branches

In the previous chapter, we annotated predicate nodes with whether they represent a conditional branch or loop in the input program. When a predicate node represents a conditional branch, there are instructions to be executed

when it evaluates to true, and likewise for false. Therefore, an instruction testing the contents of the predicate's register must be generated that jumps to the first instruction scheduled for the false branch of the PDG predicate node if it evaluates to false. If it is true, then control continues, executing the associated true instructions. At the end of the true instructions, a jump must be made to the first instruction after the false instructions.

So, when a predicate node is encountered, we must insert the correct jump statements. This is achieved by modifying the schedule. Iterating through the schedule, if we encounter a predicate node, we must perform the following steps:

1. Get the immediate children from under the false branch region node. Record the position of the first of these with respect to the schedule (F_{first}), and also the last (F_{last}).
2. Get the immediate children from the under the true branch region node. Record the position of the last of these with respect to the schedule (T_{last}).
3. Insert a conditional instruction `if (reg != 0) jmp` $\texttt{F}_{\texttt{first}}$ where `reg` is the associated predicate register.
4. Insert a `jmp` $\texttt{F}_{\texttt{last+1}}$ statement after T_{last}.

For example, given the instructions in Figure 6.3 numbered 7–12, we generate the following:

```
7:  r5 = r6 gt r7
8:  if (r5 == 0) jmp 13
9:  r3 = 1i
10: r2 = r7 add r3
11: r15 = r2
12: jmp 14
13: r15 = r7
14: ...
```

6.6.3 Loops

Loops are the most difficult of the constructs to handle. In the PDG, a loop is a predicate node with a back edge to its parent region node. As with conditional branch predicate nodes, we have previously annotated loop predicate nodes so that we can identify them when they are encountered. The reason why loops are most difficult can be traced back to the original VSDG representation of a loop. In the VSDG, a θ node is split into a θ^{head} and θ^{tail} node, with the loop body contained inside of these. The handling of the iteration of the loop is left entirely up to Johnson's semantics. Recall the definition of the θ node:

> A θ node $\theta(C, I, R, L, X)$ sets its internal value to initial value I. Then, while condition value C holds *true*, sets L to the current internal value and updates the internal value with the repeat value R. When C evaluates to *false* computation ceases and the internal value is returned through the X port.

When this is split into θ^{head} and θ^{tail}, the edges to or from ports I and L are redirected to θ^{head} and those to or from ports R, X and C are redirected to θ^{tail}.

When translating loops in Chapter 5, we translated the θ^{tail} into a predicate node testing the value computed through the C port, and placed the instructions inside the body of the loop underneath the true region node of the predicate, and added a control back edge from this region node to the parent region node of the predicate. In terms of generating code, this is fine for when the loop guard evaluates to true. However, we lack any kind of necessary assignments for when the body evaluates to false before any iteration occurs; that is, assignments which align with Johnson's original semantic definition.

Values which are demanded through loops in the VSDG, and therefore PDG, need treatment for when the loop never executes at all. This is summarised as follows:

1. In Lawrence's register allocation scheme, a VSDG node demanding a value through the X port of a loop will operate on the virtual register of that assignment. However, when the loop does not execute, that assignment *never happens*.

2. Therefore, we need to generate code that assigns the *original* value from before the loop execution when this is the case.

3. However, we only want this assignment to occur if the loop *never* executes.

This, therefore, involves generating an additional Boolean assignment and an additional predicate test in generated code. We leave the optimisation of this to later stages; namely inside the LLVM compiler. Adding additional complexity to this problem is that a value may be assigned to in a nested loop, of which different registers mark the original, and post-loop registers for each value. We tackle this problem by a phase after scheduling in which we identify these registers. We show this in Algorithm 15. We make use of the I port and X port information annotated in the previous chapter.

For each PDG statement node that has a data dependence on a loop, we run Algorithm 15. The data dependence occurs via either the left or right operand registers of this node, so these are passed to the algorithm. For each loop encountered on a depth-first search (therefore including nested loops), both registers are checked to see if they match the registers tagged earlier from the

VSDG loop R port. If either does, then a PDG statement node is created which assigns to this register the tagged I port information (i.e. the initial loop value). This assignment is temporarily tagged to the associated group node for use when generating code.

Algorithm 15: Generate the assignment statements required for when a loop does not execute. The ordering assigned by the scheduling phase is also annotated.

Input : Operand registers *lReg* and *rReg* from a statement node with a data dependence edge pointing to a loop, and the predicate node of that loop, p.

Output: Loop predicate nodes tagged with assignments which handle non-execution of loops.

```
S.push(p.TrueGroup);
visited ← ∅;
while S is not empty do
    if u has not been visited then
        u ← S.pop;
        if u is a loop predicate then
            forall the r in p.RPorts do
                if r == lReg ∨ r == rReg then
                    i ← p.IPorts.find(r);
                    asmt.ResultReg ← r;
                    asmt.LOperandReg ← i;
                    p.addLoopAsmt(asmt );
    visited ⊎ u;
    forall the control dependence children w of u do
        if w has not been visited then
            S.push(w);
```

As an example, we show a C program with a nested loop which we translate into a PDG with our `proc` tool in Figure 6.4. The I and X port registers of each loop value are indicated next to each loop predicate node. In this example, the exit node returns the value in register `r0`; from the source code, we can see is the value `x`. The outer loop, which is guarded by the predicate node testing `r5`, has an I port register of `r6` for `x` (this is the initial input register for `x` in the function) and an R port register of `r0`, which is the updated value returned through the loop body. For the nested loop, the R port register is the same (the result of the nested loop body is assigned to `r0`), however the I port register is `r3`; the result of the `x++` computation in the outer loop. By running Algorithm 15, we generate an assignment of `r0` = `r6` for the outer loop when it is never executed, and an assignment of `r0` = `r3` for the nested loop when it is never executed. We can then use these assignments when generating code.

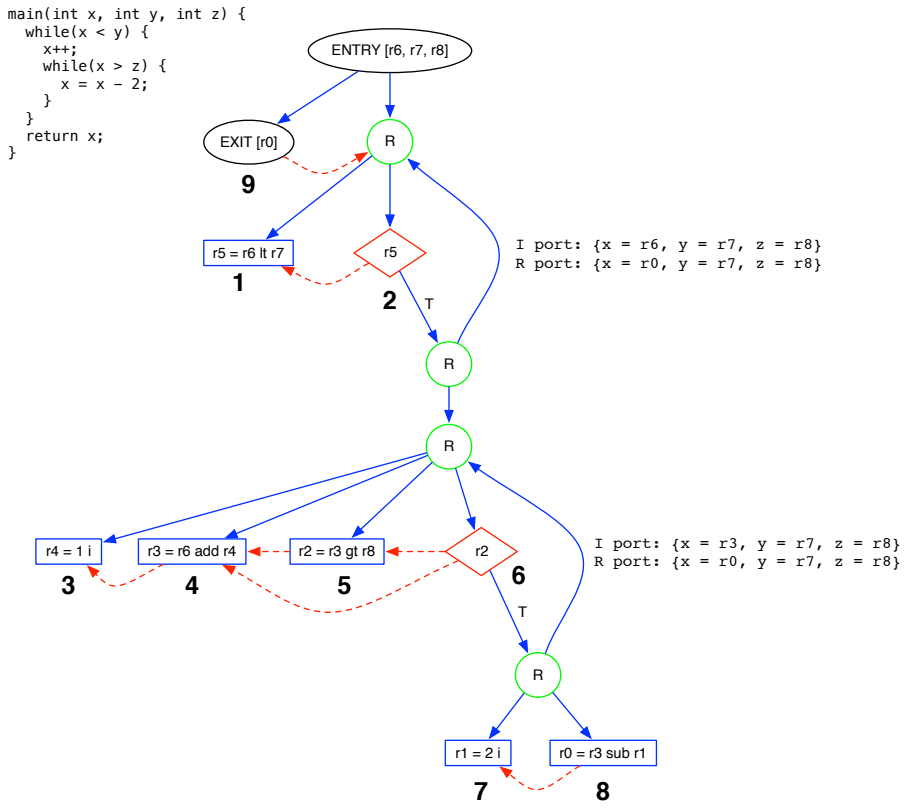


Figure 6.4: A C program with a nested loop translated into a PDG by `proc`. The I and X port registers of each loop value are indicated next to each loop predicate node.

In Figure 6.4 we have annotated the PDG with the ordering assigned by the scheduling phase. When iterating through the schedule and encountering a loop predicate node, we approach generating sequential code in the following manner:

1. Insert a Boolean assignment before the loop predicate in the schedule which is set to false.

2. Insert an instruction setting this Boolean assignment to true after the loop predicate in the schedule.

3. Get the immediate children from under the true branch region node. Record the position in the schedule of the last of these (T_{last}).

4. Insert a conditional instruction `if (reg != 0) jmp` $\texttt{T}_{\texttt{last+2}}$ where `reg` is the register tested by the loop predicate node.

5. Insert a `jmp predLabel` instruction at T_{last} to return to the top of the loop, where `predLabel` is the schedule number of the loop predicate node.

6. Insert a conditional instruction `if (reg != 0) jmp current + size(loopAssignments) + 1` where `reg` is the register of the Boolean assignment, `current` is the current position in the schedule, and `size(loopAssignments)` is the number of additional assignments generated by Algorithm 15 for this loop predicate node.

Following this approach, the pseudocode for the nested loop in Figure 6.4 is as follows:

```
1: r5 = r6 lt r7
2: b0 = 0 consti
3: if (r5 != 0) jmp 17
4: b0 = 1 consti
5: r4 = 1 consti
6: r3 = r6 add r4
7: r2 = r3 gt r8
8: b1 = 0 consti
9: if (r2 != 0) jmp 14
10: b1 = 1 consti
11: r1 = 2 consti
12: r0 = r3 sub r1
13: jmp 9
14: if (b1 != 0) jmp 16
15: r0 = r3 asmt
16: jmp 3
17: if (b0 != 0) jmp 19
18: r0 = r6 asmt
19: return r0
```

6.7 Generating LLVM

In order to generate executable code for a sequential processor, we chose LLVM bitcode as our target. There are a number of advantageous reasons for using LLVM, especially in relation to our compiler architecture:

- LLVM is a mature, regularly updated open source project that competes strongly with GCC, and is often faster than it [80].

- LLVM provides a pseudo-assembly IR called LLVM bitcode that is easy to generate [81].

- LLVM bitcode also uses virtual registers like our compiler architecture, meaning that we can leave target register allocation to LLVM.

However, LLVM bitcode has well-formedness conditions that must be adhered to:

1. LLVM treats labels in bitcode as the start of a basic block. It therefore requires the end of basic blocks to be terminated with a conditional or unconditional jump (i.e. no fall-through control flow).

2. SSA form must be strictly adhered to, that is, variables can only be assigned to once. Even though the VSDG is implicitly in SSA form, the virtual register allocation phase in Lawrence's algorithm creates assignment statements on both true and false sides of predicate nodes which assign to the same register. Additionally, γ-ordering when building the PDG creates duplicate predicate nodes, and our splitting phase duplicates nodes also.

Condition 1 is satisfied by trivially inserting extra jump instructions in the schedule, achieved by a linear walk. Condition 2 requires some extra work. Since LLVM is in SSA form, it uses `phi` instructions to "choose" between definitions of a variable when control flow merges. LLVM's `phi` instructions are of the following form

```
<result> = phi <ty> [ <val0>, <label0>], ...
```

where `<ty>` is the type of the assignment, `<val0>` is one possible definition reaching that point, and `<label0>` is the label of the basic block that definition was created in. The `...` indicates that `phi` functions can have an arbitrary number of reaching definitions. Generating `phi` functions when traversing the PDG is difficult as the recursive scheduling algorithm has no visibility of other parts of the graph other than the node being currently visited and its immediate children. However, we choose to take the same approach that the `llvm-gcc` front end takes when generating LLVM IR. For each variable that is assigned to more than once, we allocate memory with an `alloca` instruction at the beginning of the function. Then, for each use of this variable, we insert a `load` instruction beforehand, reading from this location, and update the use to the load location. Likewise, each assignment to this variable is assigned to a new virtual register, and is then followed by a `store` instruction. Since LLVM IR is not a real target architecture, these loads and stores can be easily optimised out. In fact, LLVM always does this by using the `mem2reg` pass. This promotes `alloca` instructions into SSA registers, inserting `phi` instructions where necessary. In fact, the LLVM documentation highly recommends this approach to generating `phi` instructions in LLVM IR unless there is an extremely good reason not to [81].

```
define i32 @ir(i32 %r17, i32 %r15, i32 %r16) nounwind readnone {
  %r11 = icmp ne i32 %r16, 0                      ; <i1> [#uses=2]
  %r5 = icmp eq i32 %r15, 0                       ; <i1> [#uses=1]
  br i1 %r5, label %l30, label %l7

l7:                                               ; preds = %0
  %r2 = add i32 %r17, 2                           ; <i32> [#uses=1]
  %r8s2 = add i32 %r17, 1                         ; <i32> [#uses=1]
  %r8s2. = select i1 %r11, i32 %r8s2, i32 1       ; <i32> [#uses=1]
  br label %l41

l30:                                              ; preds = %0
  %r8a = add i32 %r17, 1                          ; <i32> [#uses=1]
  %r8a. = select i1 %r11, i32 %r8a, i32 1         ; <i32> [#uses=1]
  br label %l41

l41:                                              ; preds = %l30, %l7
  %r19.0 = phi i32 [ %r8s2., %l7 ], [ %r8a., %l30 ] ; <i32> [#uses=1]
  %r18.0 = phi i32 [ %r2, %l7 ], [ 2, %l30 ]      ; <i32> [#uses=1]
  %r0 = add i32 %r18.0, %r19.0                    ; <i32> [#uses=1]
  ret i32 %r0
}
```

(a)

```
define i32 @ir(i32 %A, i32 %P, i32 %Q) nounwind readnone {
entry:
  %0 = icmp eq i32 %Q, 0                          ; <i1> [#uses=1]
  %1 = add nsw i32 %A, 1                          ; <i32> [#uses=1]
  %r1.0 = select i1 %0, i32 1, i32 %1             ; <i32> [#uses=1]
  %2 = icmp eq i32 %P, 0                          ; <i1> [#uses=1]
  %3 = add nsw i32 %A, 2                          ; <i32> [#uses=1]
  %r2.0 = select i1 %2, i32 2, i32 %3             ; <i32> [#uses=1]
  %4 = add nsw i32 %r1.0, %r2.0                   ; <i32> [#uses=1]
  ret i32 %4
}
```

(b)

Figure 6.5: Application of LLVM `-O3` to the LLVM IR generated by our `seq` tool (a) and `llvm-gcc` (b) respectively. `alloca` instructions, where possible, are promoted to SSA registers with `phi` instructions being inserted where necessary.

As an example, we show the LLVM IR generated for the independent redundancy source code (Figure 5.5) by our compiler framework in Figure 6.5a. For comparison alongside is the LLVM IR generated by LLVM itself in Figure 6.5b. Both have been optimised at `-O3` which includes the `mem2reg` pass. Both programs have the same behaviour. The most interesting difference between the two is the presence of the `phi` instructions in our own output. Whereas the `llvm-gcc` output looks very similar to the original source program, our output, once optimised by LLVM, uses `phi` functions to represent the two values added together before the function returns. In the next chapter we will explore the differences between target machine code generated by using LLVM and by our VSDG-based framework.

6.8 Summary

This chapter has explored the problem of scheduling and generating code from a PDG. We discovered that our PDG scheduling algorithm becomes more simple than those in the literature as a consequence of the γ-ordering transformation implemented in the previous chapter, which avoids illegal PDG subgraphs being formed, combined with our node splitting phase prior to scheduling. We detailed our scheduling algorithm, and then discussed how to generate code for statements, predicates and loops, showing output from our human-readable pseudocode generator. We then described how this is expanded to generate LLVM IR, allowing us to leave register allocation and target code generation to the LLVM compiler.

Chapter 7

Evaluation

7.1 Introduction

The key goal of this thesis is to address the question of whether it is feasible to use the VSDG as the sole IR in an optimising compiler. At the end of the last chapter we saw that when generating LLVM IR in our compiler framework, `phi` instructions are generated by the LLVM optimiser when the input program exhibits independent redundancy. However, this target is only an IR: real target architectures do not contain `phi` instructions.

To evaluate our compiler framework, we used our generated LLVM IR as input to the `llc` tool, which is the LLVM compiler back-end. `llc` can generate code for a variety of target architectures. We chose the Intel x86-64 [4] architecture as that was the instruction set for our development machine.

This chapter presents the following:

- We examine the LLVM IR generated by our compiler framework and compare it to the size of the LLVM IR generated by Clang [1] and `llvm-gcc` [5].
- We generate Intel x86-64 from the LLVM IR produced by the above three tools, and also from GCC. We compare and contrast the output from these 4 different compilation methods.
- We study generated code that has increasing numbers of shared computations to observe the effect that γ-ordering and node splitting on the PDG has on generated code.

7.2 Tools for comparison

We compared our compiler to existing tool chains in two stages.

1. Comparison of the LLVM IR generated by our compiler with the LLVM IR generated by Clang 1.1 and also `llvm-gcc` (GCC version 2.4.1 from LLVM build 2.7). We performed no optimisation on the VSDG or PDG in our compiler. This gives an intuition of the effects of the CFG → VSDG → PDG → CFG translation, and thus the removal and restoration of control flow.

2. Comparison of the Intel x86-64 code generated by `llc` from our compiler, `llvm-gcc` and Clang at different optimisation levels. In addition, we also generate code from GCC at the same optimisation levels.

7.3 Results

This section presents our results. We are interested in several factors:

- The number of nodes and edges in the VSDG in memory during compilation.
- The number of nodes and edges in the PDG in memory during compilation.
- The number of instructions in LLVM IR that our compiler generates compared to Clang and `llvm-gcc`.
- The number of Intel x86-64 instructions generated by the above and also GCC at different optimisation levels.

7.3.1 Without independent redundancy

Before examining examples of independent redundancy, we will look at comparisons in programs that do not contain it. We wrote a number of C programs that contained the features supported by both the VECC front-end and also our own tool (e.g. structured C without `goto`s or `switch` statements, containing only 0-trip loops). Due to limitations in our tool we limited the type of operations to integers, however we feel that this sufficiently illustrates our technique. This test set gradually increased in size, but did not introduce any code that would produce independent redundancy in the VSDG. Since we are primarily interested in the cost and effects of using the VSDG in a compiler, we did not perform any optimisations on the VSDG or PDG.

We compiled our test programs to produce LLVM IR. We compiled the same programs with the Clang and `llvm-gcc` front-ends. Figure 7.1 shows details of the size of the VSDG and PDG during compilation. For this test set, the number of nodes in the VSDG and PDG are similar. However, in some examples, the PDG has a much greater number of edges. This is due to

Program	Function lines	VSDG nodes	VSDG edges	PDG nodes	PDG edges
non_ir1.c	12	7	16	13	16
non_ir2.c	32	19	28	27	42
non_ir3.c	51	41	68	77	126
non_ir4.c	74	48	70	66	114
non_ir5.c	102	66	109	113	193
non_ir6.c	120	71	98	95	102

Figure 7.1: Size of the VSDG and PDG internal representations for a test set with no independent redundancy.

the increased number of data dependence edges required to have a well-formed PDG compared to a well-formed VSDG. However, the VSDG can be discarded once the PDG is built (in fact, it already is since `proc` and `seq` are separate tools), so we do not have to maintain both in memory at the same time.

Figure 7.2 shows the size of the LLVM IR produced by our compiler compared to Clang and `llvm-gcc`. To reiterate, no optimisation is being performed on the VSDG: it is only translated into a PDG and then back into a CFG. There is a large reduction in the lines of LLVM IR produced compared to Clang and `llvm-gcc`; especially so in the larger examples. This reduction is due to the fact that the implicitly SSA form VSDG, and hence the PDG with virtual registers, make generation of compact LLVM IR straightforward. As an example, Clang and `llvm-gcc` generate load instructions before each use of a variable and a store instruction after each new assignment, whereas our LLVM IR generator only produces loads and stores where necessary; that is for virtual registers that are assigned to more than once. Virtual registers are only assigned to more than once in our PDG within conditional branches and loops, and `non_ir6.c` consisted of straight-line code aside from two conditional branches, hence the comparative compactness.

Then we used the LLVM IR generated by the above three tools to generate code with `llc` at various optimisation levels. We also compiled the same programs with GCC at the same optimisation levels. We show the results for optimisation levels `-O0` and `-O3` in Figure 7.3. At optimisation level `-O0`, the code generated from the VSDG framework through `llc` is smaller than the code generated by `llc` from Clang and `llvm-gcc` input. Also, our code is smaller than that which is produced by GCC at `-O0`.

When optimisation level `-O3` is used, we also produce smaller code. The most striking observation is that our framework is using the same back-end as the LLVM IR produced by Clang and `llvm-gcc`, however our code is significantly more compact. Therefore, the LLVM IR we produce exposes many more opportunities for optimisers to exploit, just by translating into, and then out of, the VSDG.

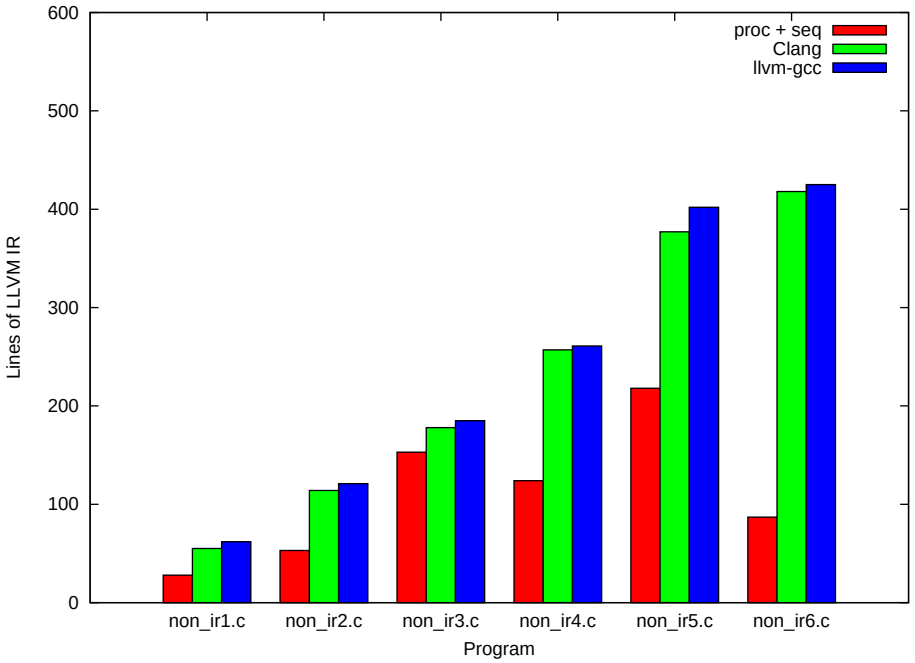


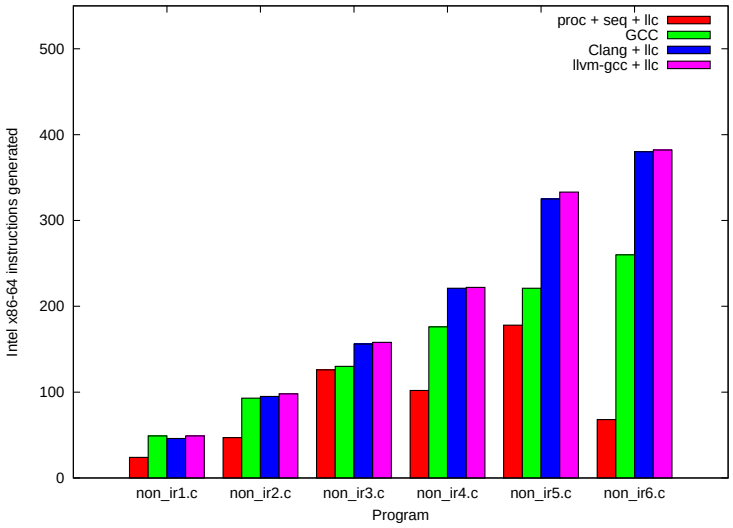
Figure 7.2: Lines of LLVM IR produced by our compiler (`proc + seq`) compared to Clang and `llvm-gcc`.

7.3.2 With independent redundancy

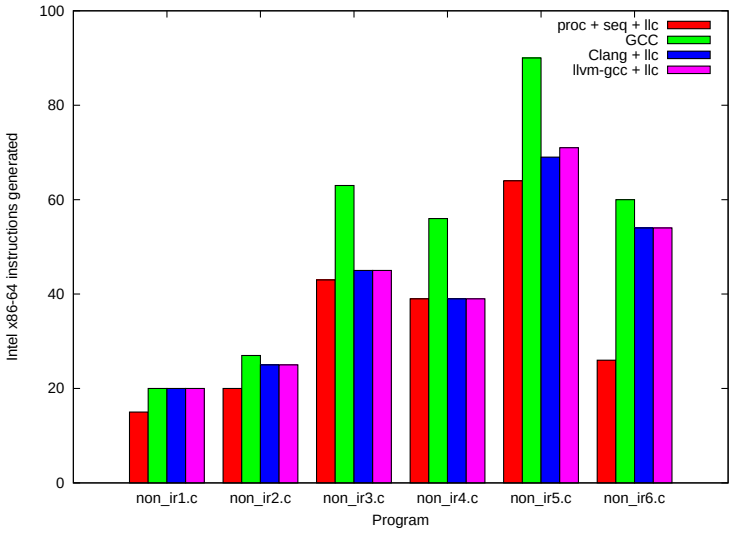
In the last chapter, we saw that whenever independent redundancy occurred in the VSDG, the resulting LLVM IR would contain `phi` instructions. When SSA is used in a compiler, it must be deconstructed before code can be generated. In LLVM, `phi` instructions are removed by inserting copy instructions in the generated code. We wrote a number of C programs that contained increasing numbers of shared computations.

Figure 7.4 shows, for increasing numbers of shared computations, the size of the VSDG and PDG in memory at compile time. It is easy to observe that the greater the number of shared computations, the greater the duplicated predicate nodes required through γ-ordering, thus the greater amount of node splitting required to restore well-formedness of the PDG. As before, we generated Intel x86-64 instructions from these PDGs using `llc`. We then generated code using the Clang and `llvm-gcc` front-ends, and also GCC. Figure 7.5 shows the results.

The primary observation is that an increasing number of shared computations radically increases the size of the generated code at `-O0`. This indicates that at this optimisation level, the `phi` functions, and their resulting copy instructions, are difficult for `llc` to optimise away. On `ir10.c`, which has 10 shared computations, the code generated by our framework at `-O0` has 306



(a)



(b)

Figure 7.3: Number of Intel x86-64 instructions generated by our compiler (`proc + seq + llc`) on non-independent redundancy code compared to Clang `+ llc`, `llvm-gcc + lcc` and GCC at optimisation levels `-O0` (a) and `-O3` (b).

Program	Shared computations	VSDG nodes	VSDG edges	PDG nodes	PDG edges
`ir1.c`	1	16	21	42	58
`ir2.c`	2	20	27	61	84
`ir5.c`	5	32	45	130	174
`ir10.c`	10	52	77	291	375

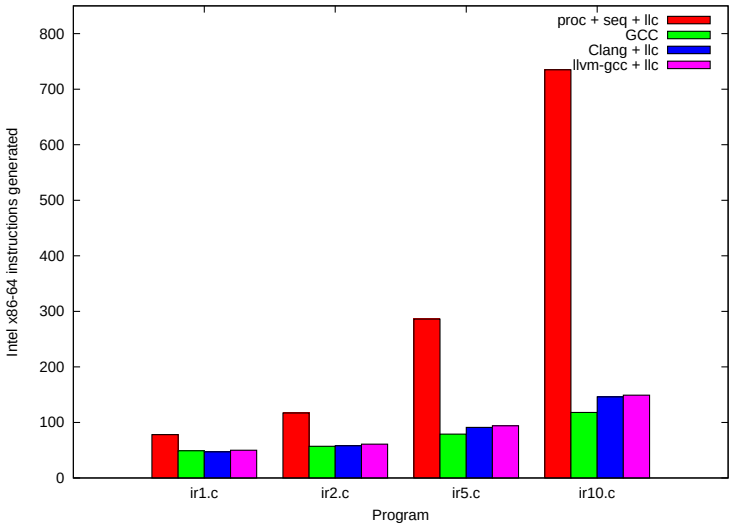
Figure 7.4: Size of the VSDG and PDG (after node splitting) internal representations for a test set with independent redundancy.

`movl` instructions accessing the stack frame compared to only 2 in the same code generated by GCC. Using the Clang and `llvm-gcc` front-ends with `llc` at the same optimisation level results in 49 and 51 `movl` instructions respectively. However, when we optimise at `-O3`, the more powerful optimisations performed by LLVM dramatically reduce the copy instructions required in the generated code. So much so that, aside from the blow-up in the size of the PDG in memory when shared computations are present, there is no other penalty for translating in and out of the VSDG, as long as sufficient optimisation is performed before code generation.

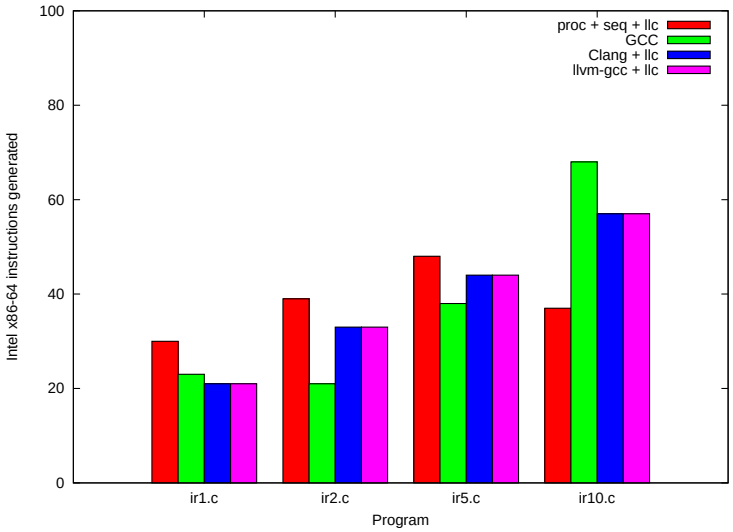
7.4 Summary

This chapter has highlighted, through implementation, the implications of translating out of the VSDG when generating code. One strength is the general compactness of code when no shared computations are introduced. In fact, with *no optimisation* occurring on the VSDG, our framework allows the LLVM back-end to produce code that is more compact than our comparators. This is notable due to the fact that both Clang and `llvm-gcc` use the same back-end, yet produce much larger code. Translating in and out of the VSDG exposes more potential optimisations in the program that `llc` can exploit. However, most importantly, it proves that we can avoid the NP-complete optimal sequentialisation problem by implementing a lazy strategy as it produces acceptable results.

We saw that when shared computations are present, the resulting `phi` nodes put pressure on the back-end. Without performing optimisation, an increasing number of shared computations can exponentially increase the size of the generated code. This is because increasing numbers of copy instructions are required. However, assuming that there is sufficient optimisation taking place (as we demonstrated with `-O3`), then there is no penalty for translating into, and out of, the VSDG. This is important as it proves that we can finally consider a new approach to compilation based on data flow, *not control flow*, in practice as well as in theory.



(a)



(b)

Figure 7.5: Number of Intel x86-64 instructions generated by our compiler (`proc + seq + llc`) on independent redundancy code compared to Clang + `llc`, `llvm-gcc + lcc` and GCC at optimisation levels `-O0` (a) and `-O3` (b).

Chapter 8

Conclusion

The VSDG is a bold compilation technique that takes a very different approach to compilation: "the best way to optimise control flow is to throw it away [and reconstruct it]" [88]. In previous work, the VSDG has been shown to be a promising IR. However, there were a number of criticisms and uncertainties about using the VSDG in a "real world" compiler. We believe we have addressed them in this thesis.

8.1 Contributions

The contributions of this thesis are:

An IR survey In Chapter 2, we have performed a comprehensive survey of compiler IRs over time. Our findings show that compilers have been dominated by CFG-based IRs for most, if not all, of their lifetime. However, IRs like the VSDG are beginning to reoccur in the literature as a tool enabling new approaches to optimisation. Thus, the VSDG deserves more investigation and attention as a whole program representation.

Irreducibility Irreducible functions have long caused problems for compilers. Most compiler optimisations – especially those that analyse and transform loops – fail to work on irreducible CFGs. More importantly for the VSDG, data flow IRs cannot be built from irreducible CFGs. Since most references to proofs of the presence of irreducibility dated back over 40 years, in Chapter 3, we performed an empirical study of irreducibility in current versions of open source software, and then compared them with older versions. We also studied machine-generated C code from a number of software tools. We concluded that irreducibility is extremely rare, and is becoming less common with time, so it is no longer a major hurdle to constructing IRs like the VSDG.

Construction Previous approaches to constructing the VSDG and similar IRs have been poorly documented or ignored in the literature altogether. In Chapter 4, we presented a modular approach to constructing the VSDG using structural analysis – a fine-grain form of interval analysis – on the CFG in SSA form (a typical IR available in most modern research compilers). By detecting patterns in the CFG that correspond to syntactic components of a source program, we can then use these patterns to guide VSDG construction; such as the insertion of γ-nodes and θ-nodes. Structural analysis is language independent, handles unstructured control flow, and allows the opportunity to detect and deal with irreducibility on the fly with node splitting. Although the VSDG does not support loops with unstructured exit, structural analysis can detect these, which could allow for future modification of the VSDG representation to cater for them.

Proceduralisation In Chapter 5 we presented our software tool, `proc`, that implements Lawrence's proceduralisation algorithms [83] for translating a VSDG into a PDG. We showed proof, via output from our tool, that his naïve algorithm, like previous VSDG proceduralisation attempts, produces an illegal PDG subgraph as defined by Ferrante et al [56]. The illegal subgraph exists in PDGs that correspond to more than one CFG. Specifically, this happens when there are one or more shared computations, that is, instructions which may or may not be executed down different execution paths due to γ nodes. Therefore, an evaluation strategy must be encoded for the PDG to be legal. We showed proof, via output from our tool, that Lawrence's effective algorithm successfully duplicates predicate nodes in the PDG on a program with a shared computation. We also modified Lawrence's proceduralisation framework to allow for traditional loops, and gave a worked example of how to proceduralise a nested loop. PDGs that are produced by our tool are output in a human-readable text format, and the grammar for this language, called `pdg`, is given in Appendix C.

Sequentialisation After generating a PDG in Chapter 5, we need to generate code. Therefore, in Chapter 6 we presented our software tool, `seq`. We studied previous attempts in the literature to sequentialise PDGs into a CFG, highlighting how the problem has been previously hampered by illegal PDG subgraphs. Since Lawrence's PDGs, after γ-ordering, can have statement nodes with multiple control flow parents, they are not well-formed according to Simons et al [108]. Therefore, we presented our node splitting approach that restores well-formedness to the PDG. Then, we presented our scheduling algorithm which decides on a single order of nodes in the graph. We showed how to generate code from statements, conditional branches and loops, using `seq`'s pseudocode output. Then,

we detailed our approach to generating LLVM IR from the tool.

Evaluation In Chapter 7, we compared our framework to existing compilers. We presented a number of observations. For a test set of programs without shared computations, our framework, with no optimisation, produces dramatically less LLVM IR than the Clang and `llvm-gcc` front-ends. Additionally, on the same test set with no optimisation, our VSDG framework produces more compact code than LLVM with Clang and `llvm-gcc` and also GCC. Additionally, we showed that as the number of shared computations increase in an input program, the greater pressure that is put on the code generator, in this case LLVM's `llc` back-end, to optimise away `phi` instructions that arise. LLVM's SSA deconstruction algorithm inserts a large quantity (almost exponential) of copy instructions when generating code at `-O0` from shared computations in our framework. However, when sufficient optimisation is taking place, these copy instructions are limited to the point that there is no penalty when translating into, and out of, the VSDG. Thus, we can finally consider a new approach to compilation based on data flow, *not control flow*, in practice as well as in theory.

8.2 Further work

We end this thesis with a number of directions for future work. Some of these were outside the scope of this thesis, and others represent different directions in which VSDG research could go.

Reducing copy instructions The evaluation of our framework shows that, in an SSA-based compiler such as LLVM, exponentially large numbers of `phi` instructions are present during compilation of shared computations, and the resulting copy instructions are not currently being optimised away at `-O0`. These `phi` instructions occur after we translate out of the PDG into LLVM intermediate code. At higher optimisation levels such as `-O3` LLVM is able to optimise these away. We would be interested in evaluating whether the number of copy instructions could be decreased by changes in our own framework, or by other means. Would it be possible to avoid exponential numbers of `phi` instructions altogether? We have included Lawrence's algorithm in the Appendix. There is definitely scope for making this more efficient in terms of time and space required. The tools that we have created during this thesis would be an ideal base for experimenting with this. The algorithm has been implemented, and writing additional tools for analysis of the algorithm would be a good place to start. How does the time and space efficiency of the algorithm scale over different programs? Are there any tweaks that can be made

when proceduralising programs with independent redundancy that keep the number of resulting ϕ-nodes low?

Leveraging research compilers Our implementation is experimental, but we feel there is enough promise in the VSDG to warrant it being constructed inside a fully functional compiler. How much modification to the VSDG would be required for it to be built in GCC or LLVM? Modifications would need to be made for unstructured exit to loops, either in the form of new nodes or semantics, or by transforming the source program or VSDG to remove them. This would then allow the VSDG handle large benchmarks such as SPEC [7] or the Linux kernel to show performance against existing compiler technology. For example, Lawrence's thesis [83] does not use θ-nodes for loops. Instead, he models them using infinite (treelike, regular) chains of μ-nodes. This solves all of the current restrictions with the VSDG such as it only being able to represent reducible programs and 1-trip loops. However, there are obvious connotations for representing loops as infinite chains of nodes – how would they be implemented in a compiler with finite memory? We predict that work on this could start by using the text file VSDG output of Johnson's VECC compiler, and then parsing the graph using the grammar described in the Appendix of his thesis [72]. Then, algorithms could be developed to transform these VSDGs, which use θ-nodes for loops, into the VSDGs that use infinite chains of μ-nodes for loops as described in Lawrence's thesis. What would the increase in the size of the VSDG be after this translation? Is there a neat notation or trick that could be used to represent infinite loops finitely?

Hardware compilation As previously noted by Johnson [72], the VSDG nodes have close mappings to equivalent hardware: γ nodes map to multiplexers, θ-nodes map to Moore state machines [86], `load` and `store` nodes map to memory access cycles, and other nodes produce combinatorial logic. We would be extremely interested to see a VSDG to hardware compiler, especially as the power of FPGA and GPU technology, and hence their ease of programming, increases. The Phoenix project [32] used a data-flow IR called Pegasus to compile programs to hardware. It used multiplexer nodes to represent conditional choice, which is similar to how the VSDG's γ-nodes work. Loops were represented using back-edges. This work on generating hardware is nearly 10 years old, and generating hardware is a much more commonplace theme in current research, especially with the wider availability of FPGA and GPU programming kits. It would be a good time to investigate the potential of the VSDG as an IR for generating hardware in this way. Like before, work could either begin from our own tool (i.e. instead of generating LLVM bitcode from the PDG, generate hardware) or the VSDGs output

by Johnson's VECC compiler (i.e. generate hardware by walking the VSDG).

Detecting shared memory As an extension of the above, Lawrence's proceduralisation algorithm assigns $\oplus$ gating conditions to shared computations. When generating hardware, memory accesses must be carefully arranged. Can gating condition analysis on the VSDG solve the problem of knowing, statically at compilation time, which computations will share the same memory? An $\oplus$ gating condition may specify exactly where two instructions nay need to access the same address in memory.

Better node splitting Our node splitting algorithm on the PDG is naïve in the sense that splitting decisions are made primarily on incoming edges, rather than in any context of the program being represented. Could a better node splitting algorithm be developed that reduces the overall size increase of the PDG? On a grander scale, is the PDG even the best IR to construct after translating out of the VSDG? The only reason that we require this node splitting is that the PDG has a number of well-formedness conditions that must be met in order to generate code from it. Could an altogether different IR be used or invented for acting as the midpoint between the VSDG and sequential code?

A functional back-end The VSDG has many similarities to functional programming. Lawrence showed in the Appendix of his thesis a technique for translating a VSDG into a functional program. A VSDG that does not have any side-effecting operations is referentially transparent, and therefore is translated into a pure functional program. For VSDGs that have side-effects, Lawrence shows that these state edges can be encoded by using Haskell's state monad. We further explored the algorithmic technique in our paper [111]. However, we have not yet implemented this algorithm as it was outside the scope of this thesis. Therefore, by using the output of Johnson's VECC compiler and by following this algorithm, we reckon it would be straightforward to begin generating functional programs. There would be a great deal of scope in investigating techniques for generating *good* functional programs. If this works, then would it be feasible to create a VSDG to functional language translator, resulting in an imperative-to-functional compiler? This would be highly novel as functional and imperative compilation are two very separate areas.

A functional front-end Assuming that functional programs could be generated from the VSDG (see above point), it would be interesting to consider the reverse: compiling functional programs into VSDGs. A parser for a functional language could be developed that generates the VSDG. This could be used as input to our own tool, and could act as the starting point for an investigation into alternative ways to compile functional

languages. We have not considered this currently, however looking at existing functional compilers such as GHC[1] may offer an insight into how to begin to tackle this problem.

An optimising interpreter Since functional languages are often interpreted, if it were possible to translate a functional program into a VSDG, could it be optimised using existing algorithms and then directly evaluated for output, thus creating an optimising interpreter? The efficiency of the source to VSDG construction algorithm would be vital here. This idea is based on the fast speed of optimisations on the VSDG as noted by Johnson and Upton [72, 123], and the tendency for functional programs to be interpreted, e.g. in the Hugs 98 Haskell interpreter[2].

Parallelism We have not touched upon the topic of parallelism in this thesis, but we feel that the VSDG would be a good starting point for investigation into instruction-level parallelism (ILP). Stateless VSDGs are referentially transparent, therefore can be evaluated in any order (perhaps in parallel). We feel it would be interesting to profile some programs that have been translated into the VSDG for their ILP potential. For example, since we know the exact instructions that require memory reads and writes in the VSDG, can we partition a program into memory-independent sections that can be executed in parallel? Work on program slicing on the PDG would be a good place to start when investigating this problem [91]. Since we translate the VSDG into the PDG, there may be a number of similarities in the techniques used.

A unifying IR? Would it be possible that sometime in the future, an IR like the VSDG could unify compilation of both imperative, functional, and hardware languages? Whilst still a distant goal, our work on the VSDG has shown small steps towards this. Whilst working with the VSDG we often thought that it represents the "essence" of von Neumann machine computing. Programs are represented as simple computations and memory accesses, yet, it is able to represent whole programs, it is efficient to optimise, and in this thesis, we have shown that generating good sequential code is possible too. Could the VSDG perhaps bridge the gap between imperative and functional compilers?

[1] `www.haskell.org/ghc/`
[2] `www.haskell.org/hugs/`

Appendix A

Construction

A.1 Omitted merge algorithms

We include the three merge algorithms from Chapter 4 that were omitted for space.

A.1.1 IfThenElse merge

The IfThenElse merge, as shown in Algorithm 16 is almost identical to the IfThen merge (Algorithm 9) in the main body of the thesis. As before, a γ-node is created for each variable used within the conditional statement. The C port of each γ-node is linked to the conditional instruction in the `if` statement. The T port links to the most up-to-date usage of the variable in the `then` block of the `if` statement. The F port links to the most up-to-date usage in the `else` block. If neither of these could be found, a top-link is created.

A.1.2 SelfLoop merge

The SelfLoop merge in Algorithm 17 is similar in concept to the WhileLoop merge shown in Algorithm 12 except that it only contains one block.

A.1.3 NaturalLoop merge

Johnson restricts his definition of the VSDG to represent programs where all iteration is handled by `while` or `for` loops, where there is only one loop exit [72]. Therefore, we show the placement of μ- and η- nodes in this algorithm, to reflect representations that allow multiple loop exits, such as Upton's GDDG [123].

The entire schema is contained within a back edge from the last block in the schema to the first. Each block in the schema has two successors: one is the next block in the schema (unless it is the final block, in which case this is the back edge) and another which is a loop exit. We place a μ node at the

Algorithm 16: Merge for the IfThenElse schema.

Input : RegionNodes, which contains the basic blocks from the IfThenElse schema.
Output: The merged IfThenElse fragment.

```
CondBlock ← RegionNodes(0);
TrueBlock ← RegionNodes(1);
FalseBlock ← RegionNodes(2);
CondFragment ← Fragments(CondBlock);
TrueFragment ← Fragments(TrueBlock);
forall the register variables v ∈ TrueBlock ∪ FalseBlock do
    create new GammaNode;
    if TrueFragment.CurrentNode(v) is not null then
        create new TrueEdge;
        TrueEdge.To ← TrueFragment.CurrentNode(v);
        MergedE ⊎ TrueEdge;
    else
        CreateTopLink(v,GammaNode.TPort);
    if FalseFragment.CurrentNode(v) is not null then
        create new FalseEdge;
        FalseEdge.To ← FalseFragment.CurrentNode(v);
        MergedE ⊎ FalseEdge;
    else
        CreateTopLink(v,GammaNode.FPort);
    create new CEdge;
    CEdge.From ← GammaNode.CPort;
    CEdge.To ← CondFragment.CurrentNode(CondBlock.T);
    MergedCurrentNode(v) ← GammaNode;
    MergedCurrentState(v) ← GammaNode;
    MergedN ⊎ GammaNode;
    MergedE ⊎ CEdge;
CurrentFragment ← MergeElements(RegionNodes);
ResolveTopLinks(CurrentFragment);
Fragments(CurrentBlock) ← CurrentFragment;
```

Algorithm 17: Merge for the SelfLoop schema.

Input : RegionNodes, which contains the basic block from the SelfLoop schema.
Output: The SelfLoop fragment.

```
CondBlock ← RegionNodes(0);
CondFragment ← Fragments(CondBlock);
create new ThetaHead;
create new ThetaTail;
forall the register variables v ∈ CondBlock do
    create new LEdge;
    LEdge.To ← CondFragment.FirstUse(v);
    LEdge.From ← ThetaHead.LPort;
    CreateTopLink(v,ThetaHead.IPort);
    create new REdge;
    REdge.From ← ThetaTail.RPort;
    REdge.To ← CondFragment.CurrentNode(v);
    CondFragment.CurrentNode(v) ← ThetaTail;
    CondFragment.CurrentState(v) ← ThetaTail;
    CondFragment.E ⊎ LEdge, REdge;
create new CEdge, SEdgeUp, SEdgeDown;
CEdge.From ← ThetaTail.CPort;
CEdge.To ← CondFragment.CurrentNode(CondBlock.T);
SEdgeUp.From ← CondFragment.FirstSideEffect;
SEdgeUp.To ← ThetaHead.STATE;
SEdgeDown.From ← ThetaTail.STATE;
SEdgeDown.To ← CondFragment.LastSideEffect;
MergedN ⊎ ThetaHead, ThetaTail;
MergedE ⊎ CEdge, SEdgeUp;
CurrentFragment ← MergeElements(RegionNodes);
ResolveTopLinks(CurrentFragment);
Fragments(CurrentBlock) ← CurrentFragment;
```

loop header, and a η node at each loop exit. The condition for each η node is linked to the terminating condition of each loop exiting block. We keep a reference to each block in the schema along with the η node that exits it. This map is used when merging to make sure top-links are resolved to the right η node.

Algorithm 18: Merge for the NaturalLoop region.

Input : RegionNodes, which contains the basic blocks in the NaturalLoop schema.
Output: The merged NaturalLoop fragment.

```
CondBlock ← RegionNodes(0);
CondFragment ← Fragments(CondBlock);
ctr ← 1;
create new Mu;
CondFragment ⊎ Mu;
forall the register variables v ∈ CondBlock do
    create new LEdge;
    LEdge.To ← CondFragment.FirstUse(v);
    LEdge.From ← Mu.LPort;
    CreateTopLink(v,Mu.IPort);
    CondFragment ⊎ LEdge;
while ctr < RegionNodes.Size do
    BodyBlock ← RegionNodes(ctr);
    BodyFragment ← Fragments(BodyBlock);
    forall the register variables v ∈ BodyBlock do
        create new LEdge;
        if v ∉ CondBlock then
            LEdge.From ← BodyFragment.FirstUse(v);
            LEdge.To ← Mu.LPort;
            CreateTopLink(v,Mu.IPort);
        else
            LEdge.To ← CurrentFragment.LastUse(v);
            LEdge.From ← BodyFragment.FirstUse(v);
            MergedCurrentNode(v) ← BodyFragment.FirstUse(v);
        create new Eta;
        BodyFragment ⊎ Eta;
        EtaMap(RegionNodes(ctr)) ← Eta;
        create new CEdge, SEdgeUp, SEdgeDown;
        CEdge.From ← Eta.CPort;
        CEdge.To ← BodyFragment.CurrentNode(BodyBlock.Terminator);
        SEdgeUp.From ← BodyFragment.FirstSideEffect;
        SEdgeUp.To ← Mu.STATE;
        SEdgeDown.From ← Eta.STATE;
        SEdgeDown.To ← BodyFragment.LastSideEffect;
        BodyFragment ⊎ CEdge, SEdgeUp, SEdgeDown;
    ctr ← ctr + 1;
CurrentFragment ← MergeElements(RegionNodes);
CurrentFragment ⊎ EtaMap;
ResolveTopLinks(CurrentFragment,EtaMap);
Fragments(CurrentBlock) ← CurrentFragment;
```

Appendix B

Proceduralisation algorithm

The VSDG to PDG implementation in Chapter 5 is based on the work in Lawrence's thesis [83]. For reference, we include the full algorithm here.

B.1 Lawrence's effective algorithm

Central is the procedure `buildPDG`, which works by a recursive, post-order traversal of the *dominator* tree of the VSDG[1], using a system of *gating conditions* computed during traversal to combine together the results of recursive calls. Importantly, this algorithm supports additional operations which avoid the flaws of naïve translation. At each stage of recursion, `buildPDG` returns a PDG fragment; the result of the outermost `buildPDG` call is the complete PDG (for the function).

Gating conditions are defined and explained in Section B.1.1. The main `buildPDG` procedure is given in Section B.1.2. Finally Section B.1.3 describes the extra operation of γ-ordering, avoiding the correctness problem of the naïve algorithm.

Further notation will also be useful. We write $D(n)$ for the nodes dominated by n, i.e. $\{n' \mid n \,\mathtt{dom}^*\, n'\}$, or, interchangeably, the subgraph of the dominator tree induced by those nodes. Thus $D(n_0)$ may stand for the entire dominator tree. Secondly, let $children(t)$ be the set $\{n' \mid \mathsf{idom}(n') = n\}$, and $\mathsf{succ}(n)$ be the operands of n.

Key to the algorithm is a lemma on dominator trees proved by Tu and Padua [120]: that for any node n, its *predecessors* are either $\mathsf{idom}(n)$, or descendants of $\mathsf{idom}(n)$. That is, the *operands* $\mathsf{succ}(n)$ of a node n are *children* in the dominator tree of some *ancestor* of n:

$$\mathsf{idom}(\mathsf{succ}(n))\ \mathtt{dom}^*\ n \tag{B.1}$$

[1] That is, the tree with the `return` node n_0 as root—unlike the dominator tree of a CFG, where the `entry` node is root. Note this is the opposite formulation to that of Lawrence's thesis [83], where VSDG edges were drawn in the reverse direction using a notation like that of Petri nets, and `buildPDG` thus traversed the *postdominator* tree.

Thus, every $n' \in \mathsf{succ}(D(n))$ is either a node in $D(n)$, or a child of some ancestor of n. The set of *external producers* ${}^{\bigstar}D(n)$ are the operands to nodes in $D(n)$ which are not themselves in $D(n)$:

$${}^{\bigstar}D(n) = \mathsf{succ}(D(n)) \backslash D(n)$$

From the above, this set can be computed incrementally during traversal:

$${}^{\bigstar}D(n) = \left(\mathsf{succ}(n) \cup \bigcup_{n' \in children(n)} ({}^{\bigstar}D(n')) \right) \backslash children(n) \tag{B.2}$$

B.1.1 Gating conditions

Gating conditions are used to control the actions of the sequentialisation algorithm. For VSDG nodes u and v, the gating condition $\mathsf{gc}^u(v)$ describes the set of *gating paths* from u to v, which are paths $u \xrightarrow{*} v$ where all nodes are dominated by u, expressed as the runtime conditions under which the result of v would be used in computing u *along one of those paths*. Gating conditions are based on the *gating functions* of Tu and Padua [120] with the addition of a disjunction constructor $\oplus$. Gating conditions are defined by the following grammar:

$c \in C ::=$	Λ	Always demanded
$\mid$	$\emptyset$	Not demanded
$\mid$	$\langle ? \rangle(g, c_t, c_f)$	For some γ-node g, according to the runtime value of the predicate of g, either c_t applies, or c_f does
$\mid$	$c_1 \oplus c_2$	The node is demanded if either c_1 or c_2 says it is

We treat the $\oplus$ constructor as both associative and commutative with the following normalisations continuously applied:

$$\begin{aligned} c \oplus \Lambda &\Rightarrow \Lambda \\ c \oplus \emptyset &\Rightarrow c \\ \langle ? \rangle(g, c, c) &\Rightarrow c \end{aligned}$$

These mean that in any gating condition of the form $\ldots \oplus c_i \oplus \ldots$, every c_i must be a $\langle ? \rangle$ gating condition. Operations will preserve the invariant that all such $\langle ? \rangle$ gating conditions have different γ-nodes as their first element.

Construction of gating conditions makes use of three utility functions which are defined recursively on the structure of their arguments.

Catenation $c_1 \cdot c_2$ is associative but not commutative.

$$\begin{aligned} \emptyset \cdot c &= \emptyset \\ \Lambda \cdot c &= c \\ \langle ? \rangle(g, c_t, c_f) \cdot c &= \langle ? \rangle(g, c_t \cdot c, c_f \cdot c) \\ (c_1 \oplus c_2) \cdot c &= (c_1 \cdot c) \oplus (c_2 \cdot c) \end{aligned}$$

(Note that handling the common cases of $c \cdot \emptyset = \emptyset$ and $c \cdot \Lambda = c$ explicitly, computes the same gating conditions more efficiently).

Union $c_1 \cup c_2$ is both associative and commutative:

$$\begin{gathered} \emptyset \cup c = c \cup \emptyset = c \\ \Lambda \cup c = c \cup \Lambda = \Lambda \\ \langle ? \rangle(g, c_t, c_f) \cup \langle ? \rangle(g, c'_t, c'_f) = \langle ? \rangle(g, c_t \cup c'_t, c_f \cup c'_f) \end{gathered}$$

In other cases, it must be that c_1 and c_2 are (potentially disjunctions of) $\langle ? \rangle$s. $c_1 \cup c_2$ identifies all the γ-nodes on both sides, and combines any $\langle ? \rangle$s *with the same γ-node* using the final rule above. If any of these result in Λ then that is the overall result, otherwise all the resulting $\langle ? \rangle$s are then combined together using $\oplus$ (thus preserving the invariant above).

Individual edges The function $cond : (N \times N) \to C$ gives a gating condition for any $n \to n'$ edge.

$$cond(e) = \begin{cases} \langle ? \rangle(g, \Lambda, \emptyset), & \text{if } e \text{ is a } \mathsf{true} \text{ edge from a } \gamma\text{-node } g \\ \langle ? \rangle(g, \emptyset, \Lambda), & \text{if } e \text{ is a } \mathsf{false} \text{ edge from a } \gamma\text{-node } g \\ \Lambda, & \text{otherwise} \end{cases}$$

B.1.2 The traversal algorithm

The algorithm is expressed as a procedure `buildPDG`(n) operating on a node n, and is given in Figure B.1. `buildPDG`(n) converts into PDG form *only* the nodes in the dominator subtree $D(n)$, producing PDG $P(n)$. Note that below we make extensive use of shorthand notation for edges. Specifically, given sets of nodes $N_1, N_2 \subseteq N$, we write $N_1 \to N_2$ to indicate edges $\{n_1 \to n_2 \mid n_1 \in N_1 \wedge n_2 \in N_2\}$, and similarly for *paths*, writing $N_1 \xrightarrow{*} n$ where $n \in N$.

Hierarchical Decomposition of Demand Conditions An essential task for the algorithm is to ensure that whenever a result of a node v might be demanded to evaluate node u (i.e. there is a path $u \xrightarrow{*} v$ in the VSDG), *control dependence edges* connect $P(u)$ to $P(v)$, ensuring that $P(u)$ causes execution

of $P(v)$ if necessary. The dominator tree allows all paths in the VSDG to be decomposed and considered an edge at a time, as follows.

At each step of recursion, $\texttt{buildPDG}(u)$ considers the *edges* $D(u) \to v$ *leaving* $D(u)$, thus $v \in {}^{\bigstar}D(u)$. As $D(u)$ moves from a single leaf node to $D(n_0)$, which is all the nodes in the VSDG, it eventually captures all edges. For a given u, such edges $D(u) \to v$ may be broken down into one of two cases:

1. Edges $u \to v$; these are handled by $\texttt{buildPDG}(u)$ itself.
2. Edges leaving $D(u')$ for some child $u' \in \mathit{children}(u)$; these are handled by the recursive calls to $\texttt{buildPDG}(u')$.

Composition of edges into *paths* is recorded using gating conditions. Specifically, $\texttt{buildPDG}(u)$ maintains the gating conditions $\texttt{gc}^u(v)$ for all $v \in {}^{\bigstar}D(u)$, describing the *gating paths* $u \xrightarrow{*} v$. Crucially, $\texttt{buildPDG}(u)$ does not need to handle non-gating paths $u \to v \to w$ via other nodes $v \notin D(u)$: it merely ensures that $P(u)$ causes execution of $P(v)$ as necessary, and the PDG subtree $P(v)$ will itself execute $P(w)$ recursively.

Dominator Trees and Gating Conditions Thus, gating paths $u \xrightarrow{*} v$ for $v \in {}^{\bigstar}D(u)$ can be broken into two cases:

1. Single edges $e = u \to v$. These cause v to be demanded exactly according to $\mathit{cond}(e)$.
2. Routes from u to v *via* some child u' of u (i.e. $u = \mathsf{idom}(u')$), where $v \in {}^{\bigstar}D(u')$. Thus, $\texttt{gc}^{u'}(v)$ describes the routes from u' to v, and so $\texttt{gc}^u(u') \cdot \texttt{gc}^{u'}(v)$ describes the routes from v to u *that go through* u'.

This allows $\texttt{buildPDG}(u)$ to compute $\texttt{gc}^u(v)$ by taking the union of the edges $u \to v$ and the routes via each child u' of u. Further, recalling the definition of ${}^{\bigstar}D(u)$ in Equation B.2, $\texttt{gc}^u(u')$, for $u' \in \mathit{children}(u)$, may be computed in the same way. For these nodes $u = \mathsf{idom}(u')$, so *all* paths $u \xrightarrow{*} u'$ are gating paths, and described by $\texttt{gc}^u(u')$.

Connecting the PDG Fragments It is these $\texttt{gc}^u(u')$, for $u' \in \mathit{children}(u)$, which determine how $P(u)$ is produced by combining the $P(u')$. Specifically, $\texttt{buildPDG}(u)$ calls a procedure `link` to add control dependence edges from $P(u)$ to each $P(u')$. The `link` procedure is shown in Figure B.2; it is this procedure we modify in order to incorporate the additional operation of our algorithm: γ-ordering (Section B.1.3). Thus, for an arbitrary VSDG edge $u \to v$, one of two cases applies:

1. $u = \mathsf{idom}(v)$. In this case, during execution of $\texttt{buildPDG}(u)$, the call to `link` will directly add an edge from $P(u)$ to $P(v)$.

2. $v \in {}^{\bigstar}D(u)$. In this case, the edge will be handled by the call to $\texttt{buildPDG}(\mathsf{idom}(v))$. This call dynamically encloses $\texttt{buildPDG}(u)$. That is, the edge $u \to v$ will (by recursive traversal) be concatenated onto all the paths $\mathsf{idom}(v) \xrightarrow{*} u$, and included in computation of $\mathsf{gc}^{\mathsf{idom}(v)}(v)$. This GC is then passed to `link`, which uses it to add a control dependence edge from the appropriate part of $P(\mathsf{idom}(v))$ to $P(v)$.

Return Values As $\texttt{buildPDG}(v)$ traverses the dominator tree, its return values are a triple:

- The set ${}^{\bigstar}D(v)$;
- A *partial* PDG $P(v)$ computing the results of v into register(s) r_v. This PDG is partial in that it has no edges to the PDG statements corresponding to the $v' \in {}^{\bigstar}D(v)$, and these will need to be added to make it a valid PDG. For simplicity, we assume the root node of these PDG fragments is a group node, containing at least:
  - For γ-nodes g, an appropriate PDG predicate node, with true and false child group nodes each containing an assignment $\mathsf{r_g} = \mathsf{r_{v'}}$ for the v' targeted by the appropriate true or false edge of g;
  - For arithmetic nodes, a statement node $\mathsf{r_v} = \mathsf{op}(\mathsf{r_{succ(t)}})$.
- For each $v' \in {}^{\bigstar}D(v)$, the Gating Condition $\mathsf{gc}^v(v')$, describing where edges to the PDG subtree $P(v')$ must be added to $P(v)$.

Topological Sorts A final key to the algorithm is how $\texttt{buildPDG}(u)$ processes the children $\overrightarrow{u_i}$ of u in *topological sort* order. Specifically, recall from the definition of ${}^{\bigstar}D(u_i)$ and the properties of dominator trees, that each $v \in {}^{\bigstar}D(u_i)$ is either another child of u, or not dominated by u. Since the VSDG is acyclic, we can sort the $\overrightarrow{u_i}$ so that whenever $u_i \in {}^{\bigstar}D(u_j)$ then u_j comes *before* u_i. Thus, each u_i comes *after* every u_j which is on a path from u to u_i. The algorithm uses this to consider the $\overrightarrow{u_i}$ in turn, such that when processing each u_i, all u_j on paths $u \xrightarrow{*} u_i$ have already been processed, and thus the gating condition $C(u_i)$ is the correct value for $\mathsf{gc}^u(u_i)$.

Adding data dependence edges Using the virtual register assignment that labels each PDG node, for each operand register we add an edge in E_D to the PDG node that produces that operand. Then, as in the naïve algorithm, the endpoints of each edge are moved up the CDG until they are between siblings.

`buildPDG`$(u \in N) =$

Let $C(v) = \emptyset$ be a map from transitions $v \in \mathsf{succ}(D(u))$ to GCs.
Let P store the initial PDG fragment for u. //see text
Let D store a set of transitions.
Let $\overrightarrow{u_i} = \mathit{children}(u)$.

//1. Process edges from u

Set $D = \mathsf{succ}(u)$. //Nodes whose results are used by u
For each $v \in D$,
set $C(v) = \bigcup_{e \in u \to v} \mathit{cond}(e)$.

//2. Recurse

For each u_i, let $(\bigstar D(u_i), P(u_i), \mathtt{gc}^{u_i}(v \in \bigstar D(u_i))) =$ `buildPDG`(u_i).

//3. Combine subtree results

Top-sort the $\overrightarrow{u_i}$ to respect $u_j \in \bigstar D(u_i) \Rightarrow i < j$. //see text
For each u_i in topological sort order,
//$C(u_i)$ is now a correct value for $\mathtt{gc}^u(u_i)$—see text
call `link`$(P, C(u_i), P(u_i))$. //`link` is defined in Figure B.2
for each $v \in \bigstar D(u_i)$, set $C(v) = C(v) \cup C(u_i) \cdot \mathtt{gc}^{u_i}(v)$.
Set $D = (D \cup \bigstar D(u_i)) \backslash \{u_i\}$.
Remove entry for $C(u_i)$. //Edges $D(u_j) \to u_i$ are *inside* $D(u)$

Normalise P (by merging group nodes with only one parent into parent).
Return $(\bigstar D(u), P(u), \mathtt{gc}^u(v \in \bigstar D(u))) = (D, P, C)$.

Figure B.1: The `buildPDG` algorithm. (Note mutable variables $C(\cdot)$, P and D.)

`link`(G, c, G') adds edges from (children of) PDG group node G to G' according to $c \in C$ as follows:

- `link`(G, Λ, G') adds a control edge from G to G'.
- `link`$(G, \langle ? \rangle(g, c^t, c^f), G')$, for γ-node g, identifies the corresponding PDG predicate node (as a child of G) and recurses on its `true` child with c^t and its `false` child with c^f (passing G' to both).
- `link`$(G, \emptyset, G')$ does nothing.
- `link`$(G, c_1 \oplus c_2, G')$ causes application of the γ-ordering transform, considered in Section B.1.3.

Figure B.2: The `link` procedure.

B.1.3 The γ-ordering transformation

The γ-ordering transformation is used to deal with cases of independent redundancy. Recall the VSDG in Figure 5.5, and assume the two γ-nodes and

their dominator tree children have been translated into PDG fragments with predicate nodes as the root. $P(n)$ is the PDG fragment for the shared `sub` computation. Naive treatment of these is not acceptable:

1. Adding control dependence edges to $P(n)$ from *b*oth PDG predicate nodes leads to an *i*llegal PDG.

2. Adding control dependence edges from each PDG predicate node to a *d*ifferent copy of $P(n)$ leads to a legal PDG but one in which the code for n could be dynamically executed twice.

In our algorithmic framework, cases of independent redundancy are identified by $\oplus$ gating conditions. We use $\oplus$ to guide the `link` procedure. A $\oplus$ gating condition is of the form $c_1 \oplus c_2 \oplus \ldots$ where each c_i is a $\langle ? \rangle$ gating condition. We choose one *d*ominant predicate node from these and order it before the other *s*ubsidiary ones, parallelling the choice of dominant variable in construction of Ordered Binary Decision Diagram. For gating conditions $c_1 \oplus c_2 \oplus \ldots$ the intuitive Boolean expression of the form $E_1 \vee E_2 \vee \ldots$ has no *a* priori specification of evaluation order, but one must be specified for a sequential computer with a single program counter.

The transformation proceeds as follows. Let P_d be the dominant predicate node, and consider in turn each subsidiary predicate node P_s. Remove the CDG edge $G \rightarrow P_s$ and make P'_s be a clone of P_s with *t*he same nodes as children. Then, add CDG edges $P_d.\mathsf{true} \rightarrow P_s$ and $P_d.\mathsf{false} \rightarrow P'_s$. Repeat for each remaining subsidiary node. Finally, the original call to `link` with gating condition $\langle ? \rangle(g_d, c_d^t, c_d^f) \oplus \overrightarrow{\langle ? \rangle(g_s, c_s^t, c_s^f)}$ can be completed with two recursive calls with n, firstly to $P_d.\mathsf{true}$ with gating condition $c_d^t \cup \overrightarrow{\langle ? \rangle(g_s, c_s^t, c_s^f)}$, and secondly to $P_d.\mathsf{false}$ similarly.

Appendix C

Grammar specification for `pdg` files

The `proc` tool presented in Chapter 5 has the ability to output a textual representation of the generated PDG. This allows the possibility for it to be used as input to an existing code generation tool, or for one to be developed as further work. The structure of the format is such that it is easy to parse, either using a parser generator software tool or by hand.

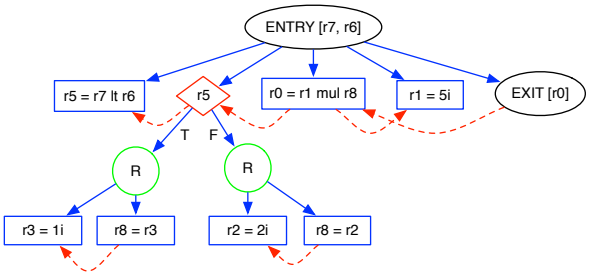
C.1 Example file

In Figure C.1 we show a PDG produced by the `proc` tool `dot` file output, and the textual representation produced by our `pdg` file generator. The syntax and over all appearance of our `pdg` files are very similar to that of Johnson's `vsdg` files [72]; since we are using these files as input, we opt to make our output look somewhat consistent with his input.

We follow the same convention as Johnson for modules and functions. Every file defines some number of modules, which in turn contain some number of functions. Each function consists of a name and a number of argument registers, and the function body is a list of nodes and edges. Each node has a name, type, operation type and result register, and optional operand registers if required. For example, a predicate node just requires a result register containing the location of the predicate test, whereas a statement node performing an `add` operation requires a result register and two operands to be added.

Each edge contains an identifier for the head and tail with "`->`" characters separating them. The tail of an edge contains a `T` or `F` label if coming from a predicate node. An edge also has a type, specifying whether it is part of the CDG or DDG.

Also in the manner of Johnson, we follow the same rules for visibility of names. Names defined in a module definition with the `public` qualifier have global scope, those defined *within* a module definition have module scope,



(a)

```
module if.vsdg {
   public function main(r7,r6) {
      // Entry node
      node node12 [type=region,op=ENTRY,arg={r7,r6}];

      // Exit node
      node node6 [type=stmt,op=EXIT,res=r0];

      // Nodes
      node node7 [type=stmt,op=mul,res=r0,l=r1,r=r8];
      node node0 [type=pred,res=r5];
      node node2 [type=region];
      node node3 [type=region];
      node node11 [type=stmt,op=lt,res=r5,l=r7,r=r6];
      node node9 [type=stmt,op=consti,res=2i];
      node node4 [type=stmt,op=asmt,res=r8,l=r3];
      node node5 [type=stmt,op=asmt,res=r8,l=r2];
      node node10 [type=stmt,op=consti,res=r3,l=1i];
      node node8 [type=stmt,op=consti,res=r1,l=5i];

      // Data edges
      edge node6 -> node7 [type=data];
      edge node7 -> node8 [type=data];
      edge node7 -> node0 [type=data];
      edge node0 -> node11 [type=data];
      edge node4 -> node10 [type=data];
      edge node5 -> node9 [type=data];

      // Control edges
      edge node12 -> node6 [type=control];
      edge node12 -> node7 [type=control];
      edge node12 -> node0 [type=control];
      edge node0:T -> node2 [type=control];
      edge node0:F -> node3 [type=control];
      edge node12 -> node11 [type=control];
      edge node3 -> node9 [type=control];
      edge node2 -> node4 [type=control];
      edge node3 -> node5 [type=control];
      edge node2 -> node10 [type=control];
      edge node12 -> node8 [type=control];
   }
}
```

(b)

Figure C.1: A PDG produced by the `proc` tool (a) and its `pdg` file output (b).

visible only within that module, and names defined in a function have function scope and are visible only within that function.

C.2 Grammar

In the grammar we use the following style conventions:

`Monospaced` Literal terminal symbols.

Bold italicised Other terminal literals such as identifiers and constants.

Italicised Non-terminal symbols.

C.2.1 Non-terminal rules

pdg:
    module
    pdg module

module:
    `module` *module_ name* { *module_ body* }

module_ name:
    identifier
    ϵ

module_ body:
    module_ modifier module_ item
    module_ modifier module_ item module_ body

module_ modifier:
    `public`
    ϵ

module_ item:
    `function` ***identifier*** (*argument_ list*) { *function_ body* }

argument_ list:
    identifier
    identifier , *argument_ list*

function_ body:
    function_ item
    function_ item function_ body

function_item:
    `node` ***identifier*** *parameters* `;`
    `edge` ***identifier*** *edge_label* `->` ***identifier*** *parameters* `;`

edge_label:
    `:` ***identifier***
    ϵ

parameters:
    `[` *parameter_list* `]`
    ϵ

parameter_list:
    parameter
    parameter `,` *parameter_list*

parameter:
    identifier `=` ***identifier***
    identifier `=` { *arg_list* }

arg_list:
    arg
    arg `,` *arg_list*

arg:
    identifier

C.2.2 Terminal rules

The ***identifier*** terminal follows the same syntax of those in the C programming language [77].

C.2.3 Parameters

Nodes and edges take a number of parameters.

C.2.3.1 Node parameters

Each PDG node takes one required parameter and five optional ones. The required parameter is `type`, which specifies the type of PDG node it is. The optional parameters are `op`, which specifies the type of operation that node performs, `res` which is the result register for that operation, `l` and `r`, which

represent the left and right operand registers respectively, and `arg`, which is a list of arguments for the `ENTRY` node. `op` parameters are similar to the operator node names in Johnson's VECC compiler. The `l` and `r` parameters can take constants and identifiers as well as registers. Registers are always prefixed by `r`, for example `r5`, and constants are suffixed by a letter: `i` for integers and `f` for floating point (e.g. `2i` and `6.7f`). In the case of the `call` node, the `l` operand takes an identifier representing the name of the function to call.

In addition to these parameters, predicate nodes that represent loops have two additional parameters: `iport` and `rport`. These are pairs of values and registers separated by a dot. The `iport` list contains the initial registers for values in that loop, and the `rport` list contains the registers for returned values in that loop. For example: `iport={x.r3},rport={x.r0}`. These are used by the `seq` code generator.

```
op:
     asmt                              [assignment]
     consti constf constid             [constant]
     ld st                             [plain load/store]
     vld vst                           [volatile load/store]
     tld tst                           [temporary load/store]
     neg add sub mul div mod           [signed arithmetic]
     udiv umod                         [unsigned arithmetic]
     lsh rsh ursh                      [arithmetic shift]
     not and or xor                    [bitwise arithmetic]
     fadd fsub fmul fdiv fneg          [floating point]
     call                              [function call]
     eq ne gt gte lt lte               [conditional test]
     ENTRY EXIT                        [entry/exit node]

type:
     pred                              [predicate node]
     stmt                              [statement node]
     region                            [region node]


res:
     identifier

l:
     identifier

r:
     identifier
```

C.2.4 Edge parameters

Edges either belong to the CDG or the DDG.

```
type:
     control
     data
```

References

[1] Clang: a C language family frontend for LLVM. `http://clang.llvm.org`.

[2] GCC, the GNU Compiler Collection. `http://www.gcc.gnu.org`.

[3] Graphviz. `http://www.graphviz.org`.

[4] Intel64 and IA-32 Architectures Optimization Reference Manual. `http://www.intel.com`.

[5] llvm-gcc – LLVM C front-end. `http://llvm.org/cmds/llvmgcc.html`.

[6] MATLAB Central File Exchange. `http://www.mathworks.com/matlabcentral/fileexchange`.

[7] SPEC – Standard Performance Evaluation Corporation. `http://www.spec.org`.

[8] TIOBE Programming Community Index for July 2009. `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`, July 2009.

[9] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. On finding lowest common ancestors in trees. In *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*, STOC' 73, pages 253–265, New York, NY, USA, 1973. ACM.

[10] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, & Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2006.

[11] F. Allen, M. Burke, P. Charles, R. Cytron, and J. Ferrante. An overview of the PTRAN analysis system for multiprocessing. *Journal of Parallel and Distributed Computing*, 5(5):617 – 640, 1988.

[12] F. E. Allen. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*, pages 1–19, New York, NY, USA, 1970. ACM.

[13] F. E. Allen and J. Cocke. Graph-theoretic constructs for program control flow analysis. Technical Report IBM Research Report RC 3923, T.J. Watson Research Center, Yorktown Heights, N.Y., July 1972.

[14] F. E. Allen and J. Cocke. A program data flow analysis procedure. *Commun. ACM*, 19(3):137, 1976.

[15] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'83)*, pages 177–189, New York, NY, USA, 1983. ACM.

[16] Z. Ammarguellat. A control-flow normalization algorithm and its complexity. *IEEE Trans. Softw. Eng.*, 18:237–251, March 1992.

[17] R. L. Ashenhurst. ACM forum. *Commun. ACM*, 30(3):195–196, 1987.

[18] R. L. Ashenhurst. ACM forum. *Commun. ACM*, 30(5):350–355, 1987.

[19] T. Ball and S. Horwitz. Constructing control flow from control dependence. Technical Report CS-TR-1992-1091, University of Wisconsin-Madison, 1992.

[20] R. A. Ballance and A. B. Maccabe. Program dependence graphs for the rest of us. Technical Report 92-10, University of New Mexico, Albuquerque, NM 87131, 1993.

[21] N. D. Barli, H. Mine, S. Sakai, and H. Tanaka. A thread partitioning algorithm using structural analysis. *Joho Shori Gakkai Kenkyu Hokoku*, 2000(74):37–42, 2000.

[22] R. S. Barton. A new approach to the functional design of a digital computer. In *Proceedings of the Western Joint IRE-AIEE-ACM Computer Conference (Western'61)*, pages 393–396, New York, NY, USA, 1961. ACM.

[23] S. Bates and S. Horwitz. Incremental program testing using program dependence graphs. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'93)*, pages 384–396, New York, NY, USA, 1993. ACM.

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

[25] G. Bell. A brief history of supercomputing: "The Crays", Clusters and Beowulfs, Centers. What next? `http://research.microsoft.com/en-us/um/people/gbell/supers/supercomputing-a_brief_history_1965_2002.htm`, 2002.

[26] N. Biggs. *Algebraic Graph Theory (2nd ed.)*. Cambridge University Press, Cambridge, England, 1993.

[27] G. Bilardi and K. Pingali. Algorithms for computing the static single assignment form. *J. ACM*, 50(3):375–425, 2003.

[28] B. Boissinot, A. Darte, F. Rastello, B. D. de Dinechin, and C. Guillon. Revisiting out-of-SSA translation for correctness, code quality and efficiency. In *Proceedings of the 7th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, pages 114–125, Washington, DC, USA, 2009. IEEE Computer Society.

[29] M. M. Brandis and H. Mössenböck. Single-pass generation of static single-assignment form for structured languages. *ACM Trans. Program. Lang. Syst.*, 16(6):1684–1698, 1994.

[30] P. Briggs, K. D. Cooper, T. J. Harvey, and L. T. Simpson. Practical improvements to the construction and destruction of static single assignment form. *Softw. Pract. Exper.*, 28(8):859–881, 1998.

[31] P. Briggs, K. D. Cooper, and L. Torczon. Rematerialization. In *PLDI '92: Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, pages 311–321, New York, NY, USA, 1992. ACM.

[32] M. Budiu and S. C. Goldstein. Pegasus: An efficient intermediate representation. Technical Report CMU-CS-02-107, Carnegie Mellon University, May 2002.

[33] D. Byers, M. Kamkar, and T. Pålsson. Syntax-directed construction of value dependence graphs. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, pages 692–703, Washington, DC, USA, 2001. IEEE Computer Society.

[34] P. Campbell, K. Krishna, and R. A. Ballance. Refining and defining the program dependence web. Technical report, University of New Mexico, Albuquerque, NM, USA., 1993.

[35] L. Carter, J. Ferrante, and C. Thomborson. Folklore confirmed: Reducible flow graphs are exponentially larger. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, pages 106–114, New York, NY, USA, 2003. ACM.

[36] R. J. Chevance and T. Heidet. Static profile and dynamic behavior of COBOL programs. *SIGPLAN Not.*, 13(4):44–57, 1978.

[37] F. C. Chow and M. Ganapathi. Intermediate program representations in compiler construction – a bibliography. *SIGPLAN Not.*, 18:21–23, November 1983.

[38] C. Click and M. Paleczny. A simple graph-based intermediate representation. In *Proceedings of the 1995 ACM SIGPLAN Workshop on Intermediate Representations (IR'95)*, pages 35–49, New York, NY, USA, 1995. ACM.

[39] J. Cocke. Global common subexpression elimination. In *Proceedings of a Symposium on Compiler Optimization*, pages 20–24, New York, NY, USA, 1970. ACM.

[40] C. Collberg, G. Myles, and M. Stepp. An empirical study of Java bytecode programs. *Softw. Pract. Exper.*, 37(6):581–641, 2007.

[41] M. E. Conway. A proposal for an UNCOL. *Communications of the ACM*, 1(10):5–8, Oct. 1958.

[42] R. P. Cook and I. Lee. A contextual analysis of Pascal programs. *Softw., Pract. Exper.*, 12(2):195–203, 1982.

[43] K. D. Cooper, Harvey, T. J., and K. Kennedy. Iterative data-flow analysis, revisited. Technical report, Rice University, 2003.

[44] K. D. Cooper, T. J. Harvey, and T. Waterman. Building a control-flow graph from scheduled assembly code. Technical report, Rice University, 2003.

[45] K. D. Cooper, L. T. Simpson, and C. A. Vick. Operator strength reduction. *ACM Trans. Program. Lang. Syst.*, 23(5):603–625, 2001.

[46] M. Corti and T. Gross. Approximation of the worst-case execution time using structural analysis. In *Proceedings of the 4th ACM International Conference on Embedded Software*, EMSOFT '04, pages 269–277, New York, NY, USA, 2004. ACM.

[47] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, Oct 1991.

[48] J. W. Davidson and C. W. Fraser. The design and application of a retargetable peephole optimizer. *ACM Trans. Program. Lang. Syst.*, 2:191–202, April 1980.

[49] J. B. Dennis. First version of a data flow procedure language. In *Programming Symposium, Proceedings Colloque sur la Programmation*, pages 362–376, London, UK, 1974. Springer-Verlag.

[50] J. B. Dennis. Data flow supercomputers. *Computer*, 13(11):48–56, 1980.

[51] E. W. Dijkstra. Letters to the editor: Go to statement considered harmful. *Commun. ACM*, 11(3):147–148, 1968.

[52] D. Ebner, F. Brandner, B. Scholz, A. Krall, P. Wiedermann, and A. Kadlec. Generalized instruction selection using SSA-graphs. In *LCTES '08: Proceedings of the 2008 ACM SIGPLAN-SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 31–40, New York, NY, USA, 2008. ACM.

[53] A. Erosa and L. J. Hendren. Taming control flow: A structured approach to eliminating goto statements. In *Proceedings of the 1994 IEEE International Conference on Computer Languages*, pages 229–240, Toulouse, France, 1994. IEEE Computer Society Press.

[54] J. Eukasiewicz. *Aristotle's Syllogistic From the Standpoint of Modern Formal Logic (2nd ed.)*. Oxford Clarendon Press, Oxford, England, 1957.

[55] J. Ferrante and M. Mace. On linearizing parallel code. In *Proceedings of the 12th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '85, pages 179–190, New York, NY, USA, 1985. ACM.

[56] J. Ferrante, M. Mace, and B. Simons. Generating sequential code from parallel code. In *Proceedings of the 2nd International Conference on Supercomputing*, ICS '88, pages 582–592, New York, NY, USA, 1988. ACM.

[57] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.

[58] C. W. Fraser. A retargetable compiler for ANSI C. *SIGPLAN Not.*, 26(10):29–43, 1991.

[59] C. W. Fraser and D. R. Hanson. *A Retargetable C Compiler: Design and Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[60] H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, STOC '83, pages 246–251, New York, NY, USA, 1983. ACM.

[61] M. J. Harrold, B. Malloy, and G. Rothermel. Efficient construction of program dependence graphs. In *Proceedings of the 1993 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'93)*, pages 160–170, New York, NY, USA, 1993. ACM.

[62] P. Havlak. Construction of thinned gated single-assignment form. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing (LCPC'94)*, pages 477–499, London, UK, 1994. Springer-Verlag.

[63] P. Havlak. Nesting of reducible and irreducible loops. *ACM Trans. Program. Lang. Syst.*, 19(4):557–567, 1997.

[64] M. S. Hecht. *Flow Analysis of Computer Programs.* Elsevier Science Inc., New York, NY, USA, 1977.

[65] M. S. Hecht and J. D. Ullman. Flow graph reducibility. In *Proceedings of the 4th ACM Symposium on Theory of Computing*, STOC '72, pages 238–250, New York, NY, USA, 1972. ACM.

[66] M. S. Hecht and J. D. Ullman. Characterizations of reducible flow graphs. *J. ACM*, 21(3):367–375, 1974.

[67] J. E. Hopcroft and J. D. Ullman. An $n \log n$ algorithm for detecting reducible graphs. In *Proceedings of the 6th Annual Princeton Conference on Information Sciences and Systems*, pages 119–122, 1972.

[68] W.-M. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The superblock: an effective technique for VLIW and superscalar compilation. *J. Supercomput.*, 7(1-2):229–248, 1993.

[69] J. Janssen and H. Corporaal. Making graphs reducible with controlled node splitting. *ACM Trans. Program. Lang. Syst.*, 19(6):1031–1052, 1997.

[70] N. Johnson and A. Mycroft. Combined code motion and register allocation using the value state dependence graph. In *Proceedings of the 12th International Conference on Compiler Construction*, CC '03, pages 1–16, Berlin, Heidelberg, 2003. Springer-Verlag.

[71] N. Johnson and A. Mycroft. Using multiple memory access instructions for reducing code size. In *Proceedings of the 13th International Conference on Compiler Construction (CC'04)*, pages 265–280, 2004.

[72] N. E. Johnson. Code size optimization for embedded processors. Technical Report UCAM-CL-TR-607, University of Cambridge, Computer Laboratory, November 2004.

[73] R. Johnson, D. Pearson, and K. Pingali. The program structure tree: computing control regions in linear time. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 171–185, New York, NY, USA, 1994. ACM.

[74] R. Johnson and K. Pingali. Dependence-based program analysis. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pages 78–89, New York, NY, USA, 1993. ACM.

[75] R. C. Johnson. *Efficient program analysis using dependence flow graphs*. PhD thesis, Ithaca, NY, USA, 1995.

[76] A. Johnstone, E. Scott, and T. Womack. What assembly language programmers get up to: Control flow challenges in reverse compilation. In *European Conference on Software Maintenance and Reengineering*, page 83, Los Alamitos, CA, USA, 2000. IEEE Computer Society.

[77] B. W. Kernighan. *The C Programming Language*. Prentice Hall Professional Technical Reference, 1988.

[78] J. Knoop, O. Rüthing, and B. Steffen. Optimal code motion: theory and practice. *ACM Trans. Program. Lang. Syst.*, 16(4):1117–1155, 1994.

[79] D. E. Knuth. An empirical study of FORTRAN programs. *Softw. Pract. Exper.*, 1(2):105–133, 1971.

[80] C. Lattner. LLVM and Clang: Advancing compiler technology. Keynote Talk, Free and Open Source Developers European Meeting, FOSDEM '11, Brussels, Belgium, February, 2011.

[81] C. Lattner and V. Adve. LLVM language reference manual. `http://llvm.cs.uiuc.edu/docs/LangRef.html`.

[82] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '04, page 75, Washington, DC, USA, 2004. IEEE Computer Society.

[83] A. C. Lawrence. Optimizing compilation with the Value State Dependence Graph. Technical Report UCAM-CL-TR-705, University of Cambridge, Computer Laboratory, Dec. 2007.

[84] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1:121–141, January 1979.

[85] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the hyperblock. pages 161–170, Los Alamitos, CA, USA, 1995. IEEE Computer Society Press.

[86] E. F. Moore. Gedanken Experiments on Sequential Machines. In *Automata Studies*, pages 129–153. Princeton U., 1956.

[87] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, San Francisco, California, 1997.

[88] A. Mycroft. The value state dependence graph. SSA Seminar, Autrans, France, 2009.

[89] K. J. Ottenstein. Intermediate program representations in compiler construction: a supplemental bibliography. *SIGPLAN Not.*, 19:25–27, July 1984.

[90] K. J. Ottenstein, R. A. Ballance, and A. B. MacCabe. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, pages 257–271, New York, NY, USA, 1990. ACM.

[91] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering symposium on Practical Software Development Environments*, SDE '84, pages 177–184, New York, NY, USA, 1984. ACM.

[92] D. Padua. The Fortran I Compiler. *Computing in Science and Engg.*, 2(1):70–75, 2000.

[93] M. Paleczny, C. Vick, and C. Click. The Java HotSpot$^{\text{TM}}$ server compiler. In *Proceedings of the 2001 Java^{TM} Virtual Machine Research and Technology Symposium (JVM'01)*, pages 1–12, Berkeley, CA, USA, 2001. USENIX Association.

[94] C. A. Petri. Kommunikation mit Automaten, PhD thesis. Technical report, University of Bonn, 1962.

[95] K. Pingali, M. Beck, R. Johnson, M. Moudgill, and P. Stodghill. Dependence flow graphs: an algebraic approach to program dependencies. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '91, pages 67–78, New York, NY, USA, 1991. ACM.

[96] M. Richards and C. Whitby-Strevens. *BCPL – The Language and its Compiler*. Cambridge University Press, 1980.

[97] B. K. Rosen. High-level data flow analysis. *Commun. ACM*, 20(10):712–724, 1977.

[98] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'88)*, pages 12–27, 1988.

[99] H. J. Saal and Z. Weiss. Some properties of APL programs. In *APL '75: Proceedings of Seventh International Conference on APL*, pages 292–297, New York, NY, USA, 1975. ACM.

[100] H. J. Saal and Z. Weiss. An empirical study of APL programs. *Computer Languages*, 2(3):47 – 59, 1977.

[101] A. Salvadori, J. Gordon, and C. Capstick. Static profile of COBOL programs. *SIGPLAN Not.*, 10(8):20–33, 1975.

[102] J. Sammet. Farewell to Grace Hopper – end of an era! *Commun. ACM*, 35:128–131, April 1992.

[103] V. Sarkar. Automatic partitioning of a program dependence graph into parallel tasks. *IBM J. Res. Dev.*, 35:779–804, September 1991.

[104] S. Schäfer and B. Scholz. Optimal chain rule placement for instruction selection based on SSA graphs. In *SCOPES '07: Proceedings of the 10th International Workshop on Software & Compilers for Embedded Systems*, pages 91–100, New York, NY, USA, 2007. ACM.

[105] D. Seal. *ARM Architecture Reference Manual*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2000.

[106] M. Sharir. Structural analysis: A new approach to flow analysis in optimizing compilers. *Comput. Lang.*, 5(3):141–153, 1980.

[107] B. Simons, D. Alpern, and J. Ferrante. A foundation for sequentializing parallel code. In *Proceedings of the Second Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '90, pages 350–359, New York, NY, USA, 1990. ACM.

[108] B. Simons and J. Ferrante. An efficient algorithm for constructing a control flow graph for parallel code. Technical Report TR 03.465, IBM Santa Teresa Laboratory, San Jose, California, USA, Feb. 1993.

[109] V. C. Sreedhar, R. D.-C. Ju, D. M. Gillies, and V. Santhanam. Translating out of static single assignment form. In *Proceedings of the 6th International Symposium on Static Analysis (SAS'99)*, pages 194–210, London, UK, 1999. Springer-Verlag.

[110] J. Stanier. Removing and Restoring Control Flow with the Value State Dependence Graph. Technical report, University of Sussex, School of Informatics, Oct. 2011.

[111] J. Stanier and A. Lawrence. The value state dependence graph revisited. In *Proceedings of the Workshop on Intermediate Representations*, WIR '11, pages 53–60, 2011.

[112] B. Steensgaard. Sequentializing program dependence graphs for irreducible programs. Technical Report MSR-TR-93-14, Microsoft Research, 1993.

[113] E. Stoltz, M. P. Gerlek, and M. Wolfe. Extended SSA with factored use-def chains to support optimization and parallelism. In *Proceedings of the 27th Annual Hawaii International Conference on System Sciences*, pages 43–52, 1993.

[114] R. Tarjan. Testing flow graph reducibility. In *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*, STOC '73, pages 96–107, New York, NY, USA, 1973. ACM.

[115] R. E. Tarjan. Fast algorithms for solving path problems. *J. ACM*, 28(3):594–614, 1981.

[116] R. Tate, M. Stepp, and S. Lerner. Generating compiler optimization from proofs. In *POPL '10: Proceedings of the 37th Annual ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 389–402, New York, NY, USA, 2010. ACM.

[117] R. Tate, M. Stepp, Z. Tatlock, and S. Lerner. Equality saturation: A new approach to optimization. In *POPL '09: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 264–276, New York, NY, USA, 2009. ACM.

[118] L. Torczon and K. Cooper. *Engineering A Compiler*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2007.

[119] J.-P. Tremblay and P. G. Sorenson. *Theory and Practice of Compiler Writing*. McGraw-Hill, Inc., New York, NY, USA, 1985.

[120] P. Tu and D. Padua. Efficient building and placing of gating functions. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, PLDI '95, pages 47–55, New York, NY, USA, 1995. ACM.

[121] S. Unger and F. Mueller. Handling irreducible loops: Optimized node splitting versus DJ-graphs. *ACM Trans. Program. Lang. Syst.*, 24(4):299–333, 2002.

[122] E. Upton. Optimal sequentialization of gated data dependence graphs is NP-complete. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, PDPTA '03, pages 1767–1770. CSREA Press, June 2003.

[123] E. Upton. Compiling with data dependence graphs. Technical report, University of Cambridge, Computer Laboratory, July 2006.

[124] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13(2):181–210, Apr 1991.

[125] D. Weise, R. F. Crew, M. Ernst, and B. Steensgaard. Value dependence graphs: representation without taxation. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, pages 297–310, New York, NY, USA, 1994. ACM.

[126] M. Wolfe. Beyond induction variables. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI '92, pages 162–174, New York, NY, USA, 1992. ACM.

[127] F. K. Zadeck. The development of static single assignment form. Presented at the SSA Seminar, Autrans, France, April, 2009.