Chapter 1

Graphs and gating functions —(*J. Stanier*)

1 Introduction

Many compilers represent the input program as some form of graph in order to aid analysis and transformation. A cornucopia of program graphs have been presented in the literature and implemented in real compilers. Therefore it comes as no surprise that a number of program graphs use SSA concepts as the core principle of their representation. These range from very literal translations of SSA into graph form, to more abstract graphs which are implicitly SSA. This ~~section~~ aims to introduce a selection of program graphs which use SSA concepts, and examine how they may be useful to a compiler writer.

chapter

~~1.1 Background concepts~~

Before we begin, we will outline some simple graph theoretic concepts. A graph G consists of the pair (V, E) where V is the set of all vertices and $E \subseteq N \times N$ is the set of all edges. An edge is a pair of vertices, representing a relationship between them. We illustrate this in the simple example in Figure 1. The set of vertices V contains the elements $\{v_1, v_2, v_3, v_4\}$ and the set of edges E contains the elements $\{(v_1, v_2), (v_1, v_3), (v_3, v_4)\}$. We are concerned with *directed* graphs, where an edge (a, b) encodes some relation from a to b.

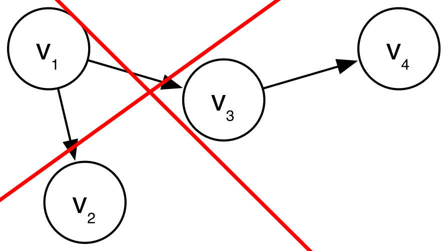
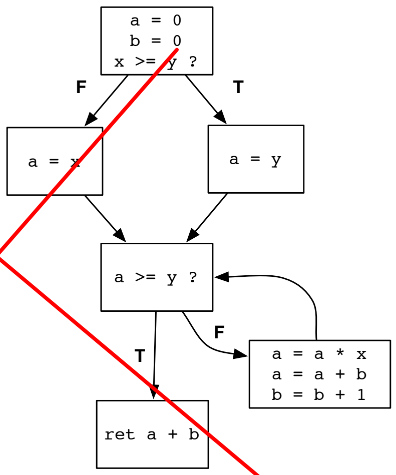


Fig. 1. A simple graph.

~~Graphs are a natural way of expressing many problems in mathematics and computer science. They are also useful for representing programs internally in a compiler~~. One of the seminal graph representations is the Control Flow Graph (CFG), which was introduced by Allen [1] to explicitly represent possible control paths in a program. ~~In the CFG, vertices are called basic blocks and contain straight-line instructions. When control enters a basic block, all instructions must be executed in that block. The directed edges that connect basic blocks together represent transfer of control flow from one block to another. An illustration of a CFG is given in Figure 2, showing a translation between some three-address code and the resulting graph.~~ Traditionally, the CFG is used to convert a program into SSA form [5]. Additionally, representing a program in this way makes a number of operations simpler to perform, such as identifying loops, discovering irreducibility and performing interval analysis techniques.

CFG is considered a prerequesite for this book.

```
1: a = 0;
   b = 0;
2: if x >= y goto 4
3: a = x;
   goto 5;
4: a = y;
5: if a >= y goto 6
   a = a * x;
   a = a + b;
   b = b + 1;
   goto 5;
6: ret a + b;
```



~~**Fig. 2.** Some three address code and the resulting CFG after translation.~~

The CFG models control flow, but many graphs model *data flow*. This is useful as a large number of compiler optimizations are based on data flow. The graphs we consider in this section are all data flow graphs, representing the data dependences in a program. We will look at a number of different SSA-based graph representations. These range from those which are a very literal translation of SSA into a graph form to those which are more abstract in nature. An introduction to each graph will be given, along with diagrams to show how sample programs look when translated into that particular graph. Additionally, we will touch on the literature describing the usage of a given graph with the application that it was used for.

focus on dataflow graphs.
- present != SSA-based graph
- provide the usage of given graph

2 The SSA Graph

We begin our exploration with a graph that is very similar to SSA: the SSA Graph. Many different variations exist in the literature, so we take ours from Cooper et al [4]. An SSA Graph consists of vertices which represent operations (such as add and load) or ϕ-functions, and directed edges connect uses to definitions. The ~~incoming~~ outgoing edges to a vertex represent the arguments required for that operation, and the ~~outgoing~~ incoming 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 adding use-definition chains. We present some sample code in Figure 3. This is translated into an SSA Graph in Figure 4. Note that 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. We have annotated operators with variable names to show the correspondence between the operations in the graph and in the SSA form program.

SSA graph

use-def chains are implicit under SSA.

represent is as a CFG and put it on Fig 4

```
begin: a = 0;
       i = 0;
loop:  a = a * i;
       i++;
       if i < 100 goto loop;
end:   print(a + i);
```

```
begin: a_0 = 0;
       i_0 = 0;
loop:  a_1 = φ(a_0,a_2);
       i_1 = φ(i_0,i_2);
       a_2 = a_1 * i_1;
       i_2 = i_1 + 1;
       if i_2 < 100 goto loop;
end:   a_3 = φ(a_1,a_2);
       i_3 = φ(i_1,i_2);
       print(a_3 + i_3);
```

Fig. 3. Some sample code translated into SSA form.

Most authors of this book consider that use-def chains are implicit under SSA. You should consider that. So not clear to me what is different than "linear SSA"

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 optimizations by using standard graph traversal and transformation techniques. ~~This can make many optimizations much easier to perform than on the standard linear SSA form, especially when the focus is on loop optimization~~. It is possible to augment the SSA Graph to model memory dependencies [6]. This is achieved by adding additional *state edges* that enforce an order of interpretation. These edges are extensively used in the Value State Dependence Graph, which we will look at later, after we touch on the concept of gating functions.

- good to have graph
- can add memory dependencies

chapter code_selection

Are you sure it is SSA-graph not simply SSA? Again I do not see the subtle difference.

In the literature, the SSA Graph has been used to detect a variety of induction variables in loops [21, 8], also for performing instruction selection techniques ~~[6, 15]~~, operator strength reduction [4], rematerialization [2], and has been combined with an extended SSA language to aid compilation in a parallelizing compiler [16]. 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.

optimizations that use SSA graphs

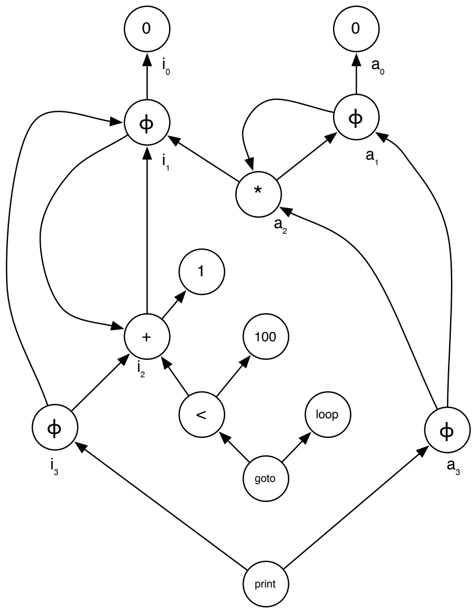


Fig. 4. Our sample program translated into an SSA Graph.

3 Gating functions

you should precise what you mean by interpreted. Indeed, phi function can be interpreted, it depends on where the execution flows. But this is true that you need the CFG. Maybe it is what you want to say.

A phi function cannot be directly "interpreted". => gating functions

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. By this logic, we cannot directly interpret the SSA Graph. Being able to interpret our IR is a useful property as it gives the compiler writer more information when implementing optimizations, and also reduces the complexity of performing code generation. Gated Single Assignment [14] form is an extension of SSA with *gating functions*. These gating functions are directly interpretable versions of ϕ-nodes, and replace ϕ-nodes in the representation. There are three forms of gating function and we take our definition from Tu and Padua [17]:

not clear how P is computed and if we can always compute it

not clear why P does not appear on mu

the 3 different gating functions

– 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. We replace ϕ-functions at loop headers with μ functions.
– The η function determines the value of a variable when a loop terminates. An 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.

It is easiest to understand these gating functions by means of an example. Figure 5 shows how our earlier code in Figure 3 translates into GSA form. ~~Here, we can see the use of both μ and η gating functions. At the header of our sample loop the ϕ-functions have been replaced by μ functions which determines between the initial and iterative values of a and i. After the loop has finished executing, the two η functions propagate the correct value from the corresponding μ function.~~

How it is writen this is a chapter on VSDG. So we should state it clearly from the beginning.

I would prefer to see all these references in a last section at the really end of the chapter

These gating functions are important as the concept will form components of the VSDG. GSA has seen a number of uses in the literature. The original usage of GSA was by Ballance et al. [14] as an intermediate stage in the construction of the Program Dependence Web. Havlak [9] presents an algorithm for construction of a simpler version of GSA – Thinned GSA – which is constructed from a CFG in SSA form. Tu and Padua [17] present an algorithm that constructs SSA and GSA simultaneously in a single process. ~~We leave further investigation of the construction algorithms as an exercise to the reader as they are lengthily and beyond the scope of this chapter.~~

references about GSA

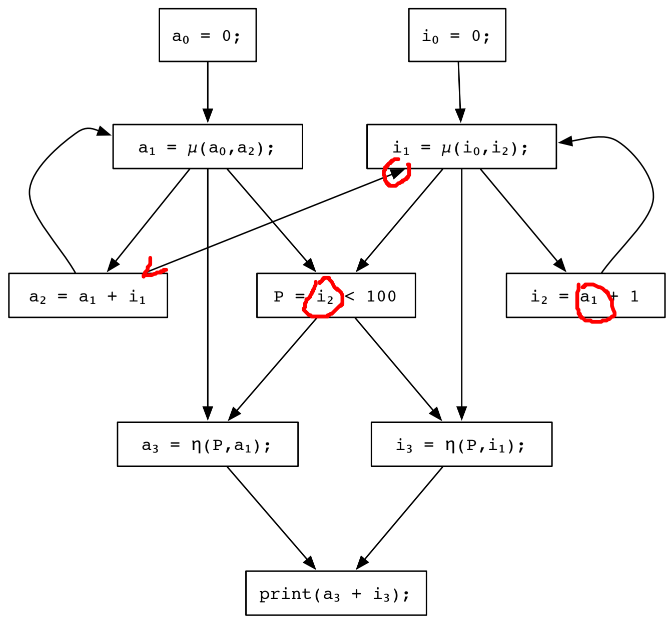


Fig. 5. A graph representation of our sample code in GSA form.

This is interesting but it lacks concret considerations. We need an example to illustrate these points

By using gating functions it becomes possible to construct IRs based solely on data dependencies. These IRs are sparse in nature compared to the CFG, making them good for analysis and transformation. This is also a more attractive proposition than generating and maintaining both a control flow graph and data flow graph, which can be complex and prone to human error. One approach has been to combine both of these into one representation, as is done in the Program Dependence Graph [7]. Alternatively,

data flow graph + gating enough to represent a program we can utilize gating functions along with a data flow graph for an effective way of representing whole program information using data flow information.

4 Towards the Value State Dependence Graph

The gating functions defined in the previous section were used in the development of a sparse data flow graph IR called the Value Dependence Graph (VDG), which was designed to completely remove the CFG as the basis of optimizations [20]. The VDG represents programs as data dependencies and does not contain any control flow information. In the VDG, there is an edge (n, v), ~~drawn as an arrow $n \rightarrow v$,~~ if a node n requires the value v in order to compute its own value. The key benefit of this representation is that there is no enforced ordering of operations.

VDG: data dependencies no enforced ordering

You say either too much or not enough. Example welcomed

? Selection in the VDG is represented by γ-nodes, which implement the same behavior as the γ gating function. ? Function calls and loops are represented by λ-nodes, with loop bodies translated into tail recursive procedures. However, there was a problem with the VDG: failure to preserve the terminating properties of a program. The original paper states that *"Evaluation of the VDG may terminate even if the original program would not..."*. This problem was addressed by the creation of the Value State Dependence Graph (VSDG) [12]. We take our definition from Johnson and Mycroft. [11]

raw description of VDG with lambda as a closure and loops as tail recursive procedures

this is not a pb but a feature

(dis)advantages of VDG vs (T)GSA?

only needed values represented=>may terminate while program does not.

The Value State Dependence Graph is a directed graph consisting of operation ? nodes, loop and merge nodes together with value- and state-dependency edges. Cycles are permitted but must satisfy various restrictions. A VSDG represents a single procedure; this matches the classical CFG but differs from the VDG in which loops were converted to tail-recursive procedures called at the logical start of the loop.

this makes sens only if you develop more the VDG.

An example VSDG is shown in Figure 6. In (a) we have the original C source for a recursive factorial function. The corresponding VSDG (b) shows both value and state edges and a selection of nodes.

4.1 Definition of the VSDG

This is not a definition. Be less formal. Simply introduce the notations required below.

~~**Definition 1.**~~ *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 $E_V \subseteq N \times N$, state-dependency edges $E_S \subseteq N \times N$. The labelling function ℓ associates each node with an operator.*

one entry node, several exit nodes, state & value dep edges. Labelling for nodes.

~~VSDGs have to satisfy two well-formedness conditions. Firstly ℓ and the (E_V) arity must be consistent, e.g. that a binary arithmetic operator must have two inputs;~~ secondly, ? the VSDG corresponds to a structured program, e.g. that there are no cycles in the ? VSDG except those mediated by θ (loop) nodes.

structured CFG is more than this for me

??

Value dependency (E_V) indicates the flow of values between nodes, ~~and must be preserved during register allocation and code motion~~. State dependency (E_S), for this paper, represents two things; the first is essential sequential dependency required by the original program, e.g. a given load instruction may be required to follow a given store instruction without being re-ordered, and a `return` node in general must wait for an

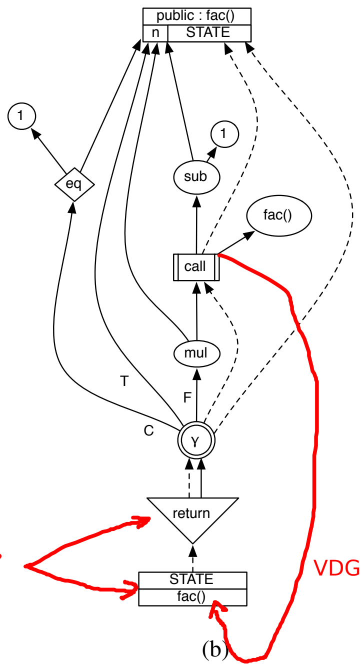
value dep: scalar data dependencies
state dep: other dep.

what about Delta (loops). We do not see the difference with GSA graph otherwise.

VSDG example.
no loop

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

(a)



(b)

why do we need 2 nodes here?

VDG would have a cycle?

Fig. 6. A recursive factorial function, whose VSDG illustrates the key graph components—value dependency edges (solid lines), state dependency edges (dashed lines), a `const` node, a `call` node, two γ-nodes, a conditional node (~~≠~~), and the function entry and exit nodes. The left-hand γ-node returns the original function argument if the condition is true, or that of the expression otherwise. The right-hand γ-node behaves similarly for the state edges, returning either the state on entry to the function, or that returned by the `call` node.

eq

only one gamma node on the figure.

earlier loop to terminate even though there might be no value-dependency between the loop and the `return` node. The second purpose is that state-dependency edges can be added incrementally until the VSDG corresponds to a unique CFG. Such state dependency edges are called *serializing* edges.

state dep added to create CFG.

An edge (a,b) is always represented with an arrow from a to b.
In the remaining of the chapter you use only once the notation "succ" and twice "successor". You can easily avoid it.
So I would remove all those notations that could be confusing and that are not used in the remaining of the chapter.
A V-successor could be renamed as a consumer node.

~~An~~ edge (n_1, n_2) represents the flow of data or control *from* n_1 to n_2, i.e. ~~in~~ the *forwards data flow direction*, so we will see ~~n_1 as a predecessor of n_2. Similarly we will regard~~ n_2 as a successor of n_1. If we wish to be specific we will write V-successors or S-successors ~~for respectively E_V and E_S successors~~. Similarly, we will write $succ_V(n)$, $pred_S(n)$ and the like for appropriate sets of successors or predecessors, and $dom(n)$ and $pdom(n)$ for sets of dominators and post-dominators respectively. We will draw pictures in the VDG form, with arrows following the *backwards data flow direction*, so that the edge (n_1, n_2) will be represented as an arrow *from* n_2 to n_1.

notations: succ, pred, dom, ...

The VSDG inherits from the VDG the property that a program is implicitly represented in Static Single Assignment (SSA) form[5]: a given operator node, n, will have zero or more E_V-successors using its value. Note that, in implementation terms, a single register can hold the produced value for consumption at all successors; it is therefore useful to talk about the idea of an output *port* for n being allocated a specific register, r,

output port

convoluted sentence

why "right-hand", why "R"? not defined. Same notation as for the caption of figure 6?

to abbreviate the idea of r being used for each edge (n_1, n_2) where $n_2 \in succ(n_1)$. Similarly, we will talk about (say) the “right-hand input port” of a subtraction instruction, or of the R-input of a θ-node.

? ?

4.2 Node Labelling with Instructions

There are four main classes of VSDG nodes, based on those of the *tri*VM Intermediate Language [10]: value nodes (representing pure arithmetic), γ-nodes (conditionals), θ-nodes (loops), and state nodes (side-effects).

redundant with above description

Value Nodes The majority of nodes in a VSDG generate a value based on some computation (add, subtract, etc) applied to their dependent values (constant nodes, which have no dependent nodes, are a special case).

γ-Nodes The γ-node is similar to the γ gating function in being dependent on a control predicate, rather than the control-independent nature of SSA ϕ-functions.

Do not need a formal def. Redundant.

Definition 2. *A γ-node $\gamma(C, T, F)$ evaluates the condition dependency C, and returns the value of T if C is true, otherwise F.*

We generally treat γ-nodes as single-valued nodes (contrast θ-nodes, which are treated as tuples), with the effect that two separate γ-nodes with the same condition can be later combined into a tuple using a single test. Figure 7 illustrates two γ-nodes that can be combined in this way.

How do you represent a tuple graphically?

combine gamma nodes with same conditional using tuples

use the tuple representation directly.

```
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;
```

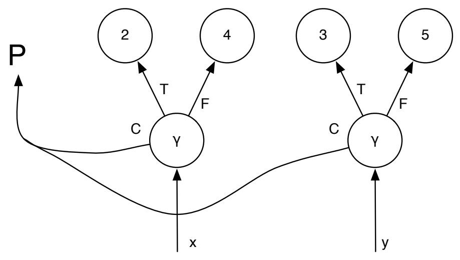


Fig. 7. Two different code schemes (a) & (b) map to the same γ-node structure.

θ-Nodes The θ-node models the iterative behaviour of loops, modelling loop state with the notion of an *internal value* which may be updated on each iteration of the loop. It has five specific ports which represent dependencies at various stages of computation.

Delta for loops.

~~**Definition 3.**~~ *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 last internal value is returned through the X port.*

just need to say that I,R,L,X might be tuples.

A loop which updates k variables will have: a single condition port C, initial-value ports $I_1, \ldots, I_k$, loop iteration ports $L_1, \ldots, L_k$, loop return ports $R_1, \ldots, R_k$, and loop exit ports $X_1, \ldots, X_k$. The example in Figure 8 shows a pair (2-tuple) of values being used for I, R, L, X, one for each loop-variant value.

I,R,L,X can be tuples.

For some purposes the L and X ports could be fused, as both represent outputs within, or exiting, a loop (the values are identical, while the C input merely selects their routing). We avoid this for two reasons: (i) we have operational semantics for VSDGs G and these semantics require separation of these concerns; and (ii) our construction of G^{noloop} requires it.

give the intuition and refer to the chapter on semantics. ?

?

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

... = j;
```

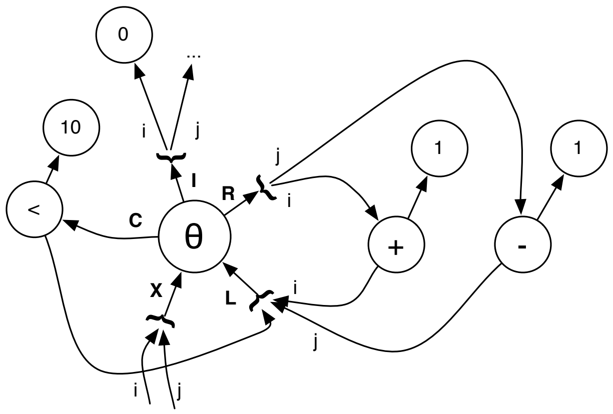


Fig. 8. An example showing a `for` loop. Evaluating the **X** port triggers it to evaluate the **I** value (outputting the value on the **L** port). While **C** evaluates to true, it evaluates the **R** value (which in this case also uses the θ-node's **L** value). When **C** is false, it returns the final internal value through the **X** port. As `i` is not used after the loop there is is no dependency on the `i` port of **X**.

The θ-node directly implements pre-test loops (`while`, `for`); post-test loops (`do...while`, `repeat...until`) are synthesised from a pre-test loop preceded by a duplicate of the loop body. At first this may seem to cause unnecessary duplication of code, but it has two important benefits: (i) it exposes the first loop body iteration to optimization in post-test loops (cf. loop-peeling), and (ii) it normalizes all loops to one loop structure, which both reduces the cost of optimization, and increases the likelihood of two schematically-dissimilar loops being isomorphic in the VSDG.

Delta node :: while loop

clarify it.

why is it important?

State Nodes Loads, stores, and their volatile equivalents, compute a value and/or state (non-volatile loads return a value from memory without generating a new state). Accesses to volatile memory or hardware can change state independently of compiler-aware reads or writes (cf. IO-state [3]).

loads/stores/volatiles => state nodes

A state node can compute a value???

non-volatile load means there is no write?

I understand the use of state edges, but why do we have state nodes not simply nodes?

The `call` node takes both the name of the function to call and a list of arguments, and returns a list of results; it is treated as a state node as the function body may read or update state.

see remark Fig. 6. Here we understand that N_infty make the job of return while in Fig. 6 you have a "return" node.

Node N_infty
Number of args <= #registers

We maintain the simplicity of the VSDG by imposing the restriction that *all* functions have *one* return node (the exit node N_∞), which returns at least one result (which will be a state value in the case of `void` functions). To ensure that function calls and definitions are ~~colourable~~ allocatable, we suppose that the number of arguments to, and results from, a function is smaller than the number of physical registers—further arguments can be passed via a stack as usual.

the fact that we necessarily have N_infty node has already been stated in the def of VSDG. This seems redundant. The remark on the number of arguments is not very interesting.

Note also that the VSDG neither forces loop invariant code into nor out-of loop bodies, but rather allows later phases to determine, by adding serializing edges, such placement of loop invariant nodes for later phases.

5 Summary

A compiler's intermediate representation can be a graph, and many different graphs exist in the literature. We can represent the control flow of a program as a Control Flow Graph (CFG) [1], where straight-line instructions are contained within basic blocks and edges show where the flow of control may be transferred to once leaving that block. A CFG is traditionally used to convert a program to SSA form [5]. We can also represent programs as a type of data flow graphs, and SSA can be represented in this way as an SSA Graph [4]. The SSA Graph has been used to detect a variety of induction variables in loops [21, 8], also for performing instruction selection techniques [6, 15], operator strength reduction [4], rematerialization [2], and has been combined with an extended SSA language to aid compilation in a parallelizing compiler [16].

Gating functions can be used to create directly interpretable ϕ-functions. These are used in Gated Single Assignment Form [14, 17].

We then described the Value State Dependence Graph (VSDG) [12], which is an improvement on the Value Dependence Graph [20]. It uses the concept of gating functions, data dependencies and state to model a program. Detailed semantics of the VSDG are available [12], as well as semantics of a related IR: the Gated Data Dependence Graph [19]. Further study has taken place on the problem of generating code from the VSDG [18, 13], and it has also been used to perform a combined register allocation and code motion algorithm [11].

This citation is to make chapters without citations build without error. Please ignore it: [**?**].

References

1. Frances E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, 1970.
2. Preston Briggs, Keith D. Cooper, and Linda 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.
3. Cliff Click and Michael Paleczny. A simple graph-based intermediate representation. *SIGPLAN Not.*, 30(3):35–49, 1993.
4. Keith D. Cooper, L. Taylor Simpson, and Christopher A. Vick. Operator strength reduction. *ACM Trans. Program. Lang. Syst.*, 23(5):603–625, 2001.

5. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.
6. Dietmar Ebner, Florian Brandner, Bernhard Scholz, Andreas Krall, Peter Wiedermann, and Albrecht 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.
7. Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987.
8. Michael P. Gerlek, Eric Stoltz, and Michael Wolfe. Beyond induction variables: detecting and classifying sequences using a demand-driven ssa form. *ACM Trans. Program. Lang. Syst.*, 17(1):85–122, 1995.
9. Paul Havlak. Construction of thinned gated single-assignment form. In *In Proc. 6rd Workshop on Programming Languages and Compilers for Parallel Computing*, pages 477–499. Springer Verlag, 1993.
10. N. Johnson. *tri*VM Intermediate Language Reference Manual. Technical Report UCAM-CL-TR-529, University of Cambridge, Computer Laboratory, 2002.
11. Neil Johnson and Alan Mycroft. Combined code motion and register allocation using the value state dependence graph. In *In Proc. 12th International Conference on Compiler Construction (CC03) (April 2003*, pages 1–16, 2003.
12. Neil E. Johnson. Code size optimization for embedded processors. Technical Report UCAM-CL-TR-607, University of Cambridge, Computer Laboratory, November 2004.
13. Alan C. Lawrence. Optimizing compilation with the Value State Dependence Graph. Technical Report UCAM-CL-TR-705, University of Cambridge, Computer Laboratory, December 2007.
14. Karl J. Ottenstein, Robert A. Ballance, and Arthur B. MacCabe. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 257–271, New York, NY, USA, 1990. ACM.
15. Stefan Schäfer and Bernhard Scholz. Optimal chain rule placement for instruction selection based on ssa graphs. In *SCOPES '07: Proceedingsof the 10th international workshop on Software & compilers for embedded systems*, pages 91–100, New York, NY, USA, 2007. ACM.
16. Eric Stoltz, Michael P. Gerlek, and Michael Wolfe. Extended ssa with factored use-def chains to support optimization and parallelism. In *In Proceedings of the 27th Annual Hawaii International Conference on System Sciences*, pages 43–52, 1993.
17. Peng Tu and David Padua. Efficient building and placing of gating functions. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 47–55, New York, NY, USA, 1995. ACM.
18. Eben Upton. Optimal sequentialization of gated data dependence graphs is np-complete. In *PDPTA*, pages 1767–1770, 2003.
19. Eben Upton. Compiling with data dependence graphs, 2006.
20. Daniel Weise, Roger F. Crew, Michael Ernst, and Bjarne Steensgaard. Value dependence graphs: representation without taxation. In *POPL '94: Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 297–310, New York, NY, USA, 1994. ACM.
21. Michael Wolfe. Beyond induction variables. *SIGPLAN Not.*, 27(7):162–174, 1992.