Dependence-Based Program Analysis

Richard Johnson
Keshav Pingali

Department of Computer Science
Cornell University, Ithaca, NY 14853

Abstract

Program analysis and optimization can be speeded up through the use of the *dependence flow graph* (DFG), a representation of program dependences which generalizes def-use chains and static single assignment (SSA) form. In this paper, we give a simple graph-theoretic description of the DFG and show how the DFG for a program can be constructed in $O(EV)$ time. We then show how forward and backward dataflow analyses can be performed efficiently on the DFG, using constant propagation and elimination of partial redundancies as examples. These analyses can be framed as solutions of dataflow equations in the DFG. Our construction algorithm is of independent interest because it can be used to construct a program's control dependence graph in $O(E)$ time and its SSA representation in $O(EV)$ time, which are improvements over existing algorithms.

1 Introduction

A number of recent papers have focused attention on the problem of speeding up program optimization [FOW87, BMO90, CCF91, PBJ$^+$91, CFR$^+$91, DRZ92]. Most optimization algorithms are based on *dataflow analysis*. Classic examples are Kildall's constant propagation algorithm [Kil73], and Morel and Renvoise's algorithm for elimination of partial redundancies [MR79]. These algorithms are usually implemented using vectors of boolean, integer or real values to represent sets of assertions, such as `x` *is 5* here, or `y+z` *is available* here. One vector is associated with each point in the control flow graph and initialized appropriately. Vector values are computed iteratively by propagating information from the inputs of statements to their outputs in the case of *forward* analysis, and from outputs to inputs in the case of *backward* analysis. The analysis terminates when all statements have a consistent set of input and output assertions.

Although easy to implement, this approach has a number of disadvantages.

- Information is propagated throughout the control flow graph, not just to where it is needed for optimization. For example, in constant propagation it suffices to propagate information from definitions of variables to their uses. In common subexpression elimination, it is unnecessary to propagate availability of an expression to points where the variables of the expression are dead.
- When the vector at some point in the program is updated, the entire control flow graph below that point (or above it, in backward analysis) may be re-analyzed, even if there are few points in that region affected by the update.
- Many optimizations benefit from analysis performed in stages, but this is difficult to do in the standard approach. Consider redundancy elimination in the following program. To deduce that the computation of `y` is redundant, we must first deduce that the computation of `w` is redundant. This kind of analysis in stages is contrary to the standard approach, which considers all assertions simultaneously.

```
...
z = a + b
w = a + b
...
x = z + 1
y = w + 1
...
```

Def-use chains provide a partial solution to these problems. They permit information to flow directly between definitions and uses without going through unrelated statements. However, def-use chains suffer from three drawbacks. First, def-use chains cannot be used for backward dataflow problems, such as the elimination of redundant computations, because they do not incorporate sufficient information about the control structure of the program. Second, this lack of control flow information in def-use chains affects the precision of analysis even in forward dataflow problems such as

[1]This research was supported by an NSF Presidential Young Investigator award CCR-8958543, NSF grant CCR-9008526, ONR grant N00014-93-1-0103, and a grant from Hewlett-Packard Corporation.

[0]*** To appear in ACM SIGPLAN'93 PLDI, June 1993 ***

constant propagation [WZ85, PBJ+91]. Finally, the worst-case size of def-use chains is $O(E^2V)$ where E is the number of edges in the control flow graph and V is the number of variables [RT81].

The size problem can be overcome by using a "factored" representation of def-use chains called *static single assignment* (SSA) form, which has worst-case asymptotic size $O(EV)$ [CFR+89, CFR+91]. However, SSA form cannot be used for backward dataflow problems. A generalization of the SSA approach, called the *sparse data flow evaluation graph*, has been proposed to address this problem, but sparse graphs take $O(N^3)$ time to construct, where N is the number of nodes in the control flow graph [CCF91, DRZ92].

In this paper, we show how these problems can be solved using the *dependence flow graph* (DFG), which can be viewed as a generalization of def-use chains and SSA form. The DFG was suggested to us by the work of Cartwright and Felleison who showed the advantages of an executable representation of program dependences [CF89]. We previously introduced the DFG using a dataflow machine style operational semantics [PBJ+91, Bec92]. Dataflow machine graphs are also the basis of the program dependence web (PDW) of Ballance, McCabe and Ottenstein [BMO90], as well as the original SSA graphs due to Shapiro and Saint [SS70]. However, our experience in implementing and using a representation based on these ideas is that a full-blown dataflow graph representation is neither necessary nor desirable.

The main contribution of this paper is the distillation and incorporation of the essence of the dataflow graph representation into a traditional optimizing compiler framework. We accomplish this as follows.

In Section 2, we give a simple *graph-theoretic* characterization of dependence flow graphs. This characterization permits the exorcism of the dataflow execution model from the description of DFGs. It also brings out key connections between our work and prior work on representing control and data dependences.

In Section 3, we describe how to construct DFGs. An important step in this construction is determining when two nodes in a control flow graph have the same control dependences. We describe how to do this in $O(E)$ time. This algorithm is of independent interest since it can be used to build a program's control dependence graph in $O(E)$ time and its SSA representation in $O(EV)$ time, which are improvements over existing algorithms [CFS90, CFR+89].

In Section 4, we show how to use the DFG in a forward dataflow problem: constant propagation with dead code elimination. This algorithm is faster than the standard control flow graph algorithm, yet it does as good a job of optimizing programs.

In Section 5, we show how to solve a backward dataflow problem: anticipatability of expressions, which is an important step in the elimination of partial redundancies [MR79].

Finally, in Section 6, we describe what we have learned so far in our implementation.

2 A graph-theoretic characterization of dependence flow graphs

We give a graph-theoretic characterization of def-use chains, static single assignment form, and the dependence flow graph. This characterization formalizes the relationship between these program representations, permits the design of an efficient construction algorithm for DFGs, and aids in proving optimization algorithms correct.

2.1 Terminology

Definition 1 *A* **control flow graph** *(CFG) is a directed graph with distinguished nodes* `start` *and* `end` *such that all nodes are reachable from* `start` *and all nodes have a path to* `end`*.* `start` *is the only node with no predecessors, and* `end` *is the only node with no successors.*

For convenience in describing algorithms that operate on CFGs, we introduce explicit switch and merge nodes to separate branching and merging of control flow from computation. A **switch** node is essentially a conditional jump that redirects control flow to one of multiple outgoing edges based on the value of an expression computed within the node. A **merge** node performs no computation but simply serves as the target of multiple control flow edges. An **assignment statement** node performs any general, non-branching computation.

It is useful to extend the standard notions of dominance, postdominance and control dependence so that they apply to *edges* as well as to nodes in the CFG.

Definition 2 *A node or edge x is said to* **dominate** *node or edge y in a directed graph if every path from* `start` *to y includes x.*

A node or edge x is said to **postdominate** *node or edge y in a directed graph if every path from y to* `end` *includes x.*

A node or edge x is said to be **control dependent** *on node n if x postdominates all edges on some path from n to x, but x does not postdominate n. (Intuitively, n is a conditional branch that determines if control will pass through x.)*

2.2 Def-use chains

We begin by recalling the standard definition of *def-use chains*.

Definition 3 *A definition of variable* `x` *is said to* **reach** *a use of* `x` *if there is a control flow path from the definition to the use that does not pass through any other definition of* `x`*.*

A **def-use chain** *for variable* `x` *is a node pair (n_1, n_2) such that n_1 defines* `x`*, n_2 uses* `x`*, and the definition of* `x` *at n_1 reaches the use of* `x` *at n_2.*

For our purpose, it is convenient to recast this in terms of control flow *edges* rather than nodes.

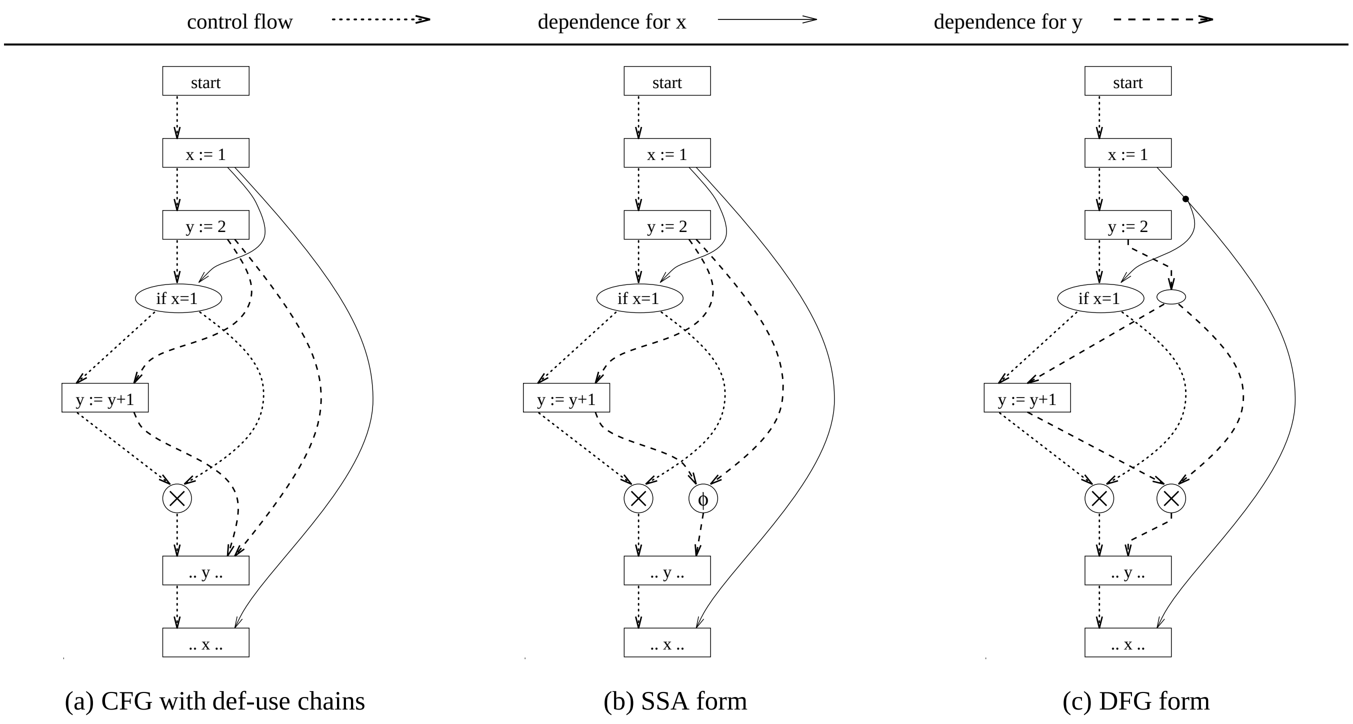


Figure 1: A Comparison of Program Representations

Definition 4 *A* **def-use chain** *for variable* `x` *is an edge pair* (e_1, e_2) *such that*

1. *the source of* e_1 *defines* `x`*,*
2. *the destination of* e_2 *uses* `x`*, and*
3. *there is a control flow path from* e_1 *to* e_2 *with no assignment to* `x`*.*

Figure 1(a) shows a control flow graph with def-use chains. The limitations of def-use chains have been discussed extensively in the literature [WZ85, PBJ+91], and we summarize them here. Consider the problem of constant propagation using def-use chains: the standard algorithm replaces a use of a variable with a constant if the right hand side of every definition reaching that use is that constant [ASU86]. In Figure 1(a), this algorithm determines that the use of `x` in the conditional branch can be replaced by the constant `1`, and this fact is determined without propagating the value of `x` through the assignment to `y` that is between the definition and use of `x`. Similarly, it determines that the right hand side of the statement `y:=y+1` can be replaced by the constant `3`. However, this algorithm cannot determine that the last use of `y` in this program can be replaced by the constant `3`, since there are two def-use edges carrying different constants that reach this use. By contrast, the standard dataflow analysis algorithm for constant propagation on the control flow graph will find this constant, since it deduces correctly that the false side of the conditional branch is dead[2].

[2]Details of this algorithm are discussed in Section 4.

To summarize, algorithms using def-use chains can produce less optimized code than algorithms performing dataflow analysis directly on the control flow graph. In addition, def-use chains cannot be used for backward dataflow analysis, and the worst-case size of def-use chains is $O(E^2V)$ [RT81], which is rather large.

2.3 Static single assignment form

Static single assignment form solves the size problem of def-use chains by introducing a so-called ϕ-function to combine def-use edges having the same destination [CFR+89, CFR+91]. In an SSA representation, each use of a variable is reached by exactly one definition or ϕ-function. Figure 1(b) shows the SSA form for the previous example. Notice that the def-use edges for variable `y` are combined by a ϕ-function placed at the merge in the control flow graph. SSA edges have the following graph-theoretic characterization.

Definition 5 *An* **SSA edge** *for variable* `x` *corresponds to an edge pair* (e_1, e_2) *such that*

1. *there exists a definition of* `x` *that reaches* e_1*,*
2. *there exists a use of* `x` *reachable from* e_2*,*
3. *there is no assignment to* `x` *on any control flow path from* e_1 *to* e_2*, and*
4. e_1 *dominates* e_2*.*

The first two conditions assert that an SSA edge connects two points on some path from a definition of `x` to a use of x reached by that definition. Conditions 3 and 4 ensure that the only definitions that reach e_2 are those that reach e_1; otherwise there would be ϕ-functions between e_1 and e_2 and there would be no SSA edge from e_1 directly to e_2. The worst-case size of the SSA representation is $O(EV)$. This solves the size problem of def-use chains, but the SSA form cannot be used directly in backward dataflow analysis problems.

2.4 Dependence flow graph

Figure 1(c) shows the dependence flow graph for the running example. Unlike def-use chains, which go directly from definitions to uses, a DFG edge for a variable x can bypass a region of the control flow graph only if this region is a single-entry single-exit region that does not contain an assignment to x, since such a region has neither data nor control information that is of interest to program analysis. The following theorem defines single-entry single-exit regions formally, and states an important connection between control dependences and DFGs.

Theorem 1 *The following are equivalent.*

- *e_1 and e_2 enclose a single-entry single-exit region.*
- *e_1 dominates e_2, e_2 postdominates e_1, and every cycle containing e_1 also contains e_2 and vice versa.*
- *e_1 and e_2 have the same control dependences.*

For lack of space, we omit the proof of this theorem.

A related structure called a *hammock* has been discussed in the literature [Kas75]. Hammocks are not the same as single-entry single-exit regions since the exit node in a hammock can be the target of edges outside the hammock; besides, the algorithm for finding hammocks is $O(EN)$.

Just as def-use chains are "intercepted" by ϕ-functions in the SSA representation, they are intercepted by `switch` and `merge` operators in the DFG. DFG edges can be characterized as follows:

Definition 6 *A* **DFG edge** *for variable* x *corresponds to an edge pair* (e_1, e_2) *such that*

1. *there exists a definition of* x *that reaches* e_1,
2. *there exists a use of* x *reachable from* e_2,
3. *there is no assignment to* x *on any control flow path from* e_1 *to* e_2,
4. e_1 *dominates* e_2,
5. e_2 *postdominates* e_1, *and*
6. *every cycle containing* e_1 *also contains* e_2 *and vice versa.*

Conditions 1 through 4 are the same as in the SSA representation. In the DFG, the merge operator plays the same role as ϕ-functions do in SSA form. Conditions 4 through 6 formally specify that the region of the control flow graph between e_1 and e_2 must be a single-entry single-exit region. For example, in Figure 1(c) the region of the control flow graph between the assignment to x and the use of x is a single-entry single-exit region containing no definition of x. Thus there is a dependence edge from the definition of x directly to the use of x. However, this region contains a definition of y, so dependence edges for y cannot bypass this region; they are intercepted by a switch operator at the conditional branch. In this way, dependences in the DFG are intercepted by merges and switches at merge points and branches respectively in the control flow graph.

3 Constructing DFGs

We now describe the construction algorithm for dependence flow graphs. We first outline our algorithm for identifying single-entry single-exit regions and then show how this information is used to build the DFG.

3.1 Finding single-entry single-exit regions

Abstractly, decomposing a control flow graph into single-entry single-exit regions can be viewed as providing a "parse tree" of the control flow structure. In structured programs, syntactic constructs such as `if-then-else` and `while` loops provide information for determining single-entry single-exit regions. For general control flow graphs, however, we need an efficient algorithm for discovering this information.

Consider any two single-entry single-exit regions. It is easy to show that if they overlap, then either one is nested within the other, or the intersection is itself a single-entry single-exit region. If we only consider regions that are not formed by sequentially composing smaller regions, then we can show that single-entry single-exit regions are pairwise either nested, disjoint, or sequentially ordered. Thus these regions give a hierarchical decomposition of a control flow graph's structure.

For lack of space, we sketch our $O(E)$ algorithm for finding single-entry single-exit regions. A longer paper giving the details of this algorithm can be obtained from the authors. Given a control flow graph, we want to find sets of edges having the same control dependences. Such edges are totally ordered, and each pair of consecutive edges are the entry/exit of a single-entry single-exit region.

To find sets of edges having the same control dependences, note that we can insert a dummy node on each edge and then compute the property for nodes[3]. We then reduce the problem to one of finding sets of *cycle equivalent* nodes in a related graph.

Definition 7 *Control flow nodes* a *and* b *are said to be* **cycle equivalent** *if every cycle containing* a *also contains* b *and vice versa.*

[3]Adding E nodes does not change the $O(E)$ time complexity of our algorithm.

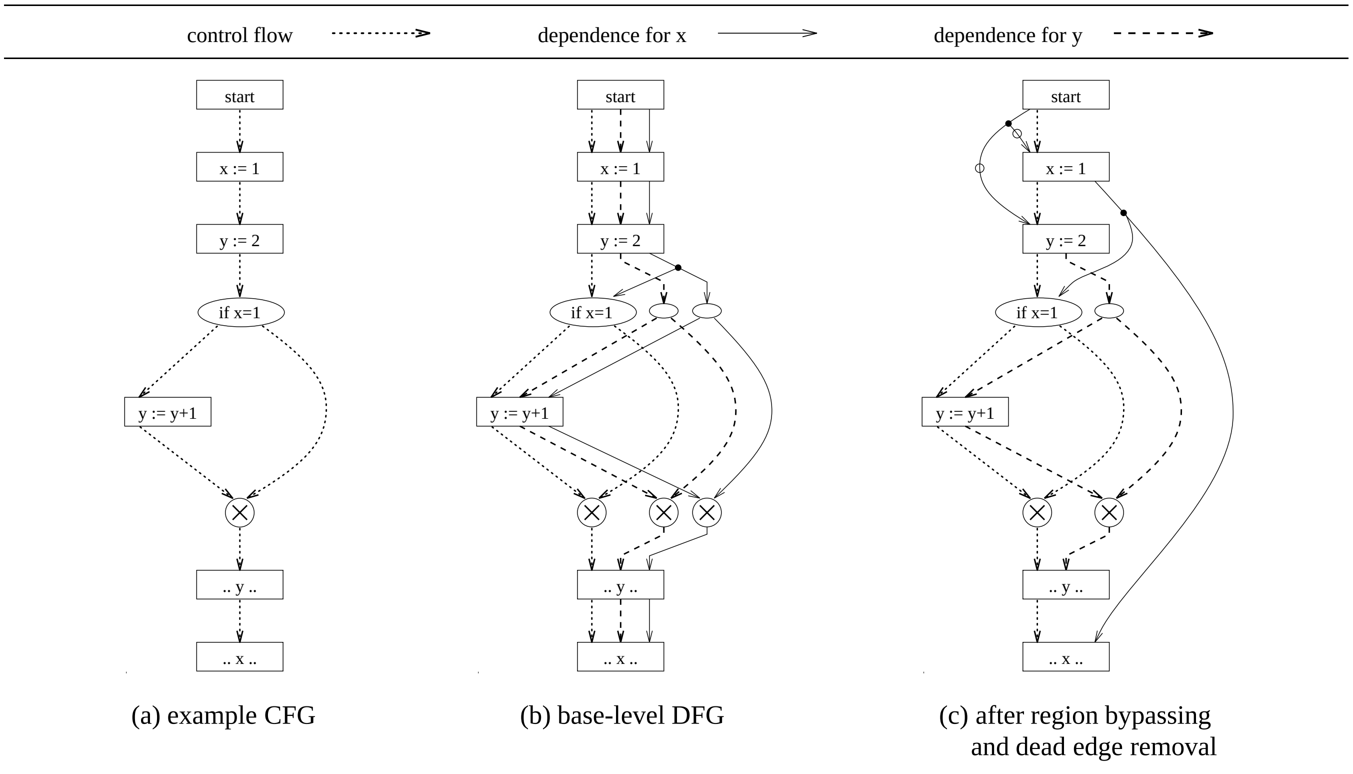


Figure 2: An Illustration of DFG Construction

Claim 1 *Nodes a and b have the same control dependences if and only if a and b are cycle equivalent in the strongly connected component formed by adding the edge* `end` $\rightarrow$ `start` *to the CFG.*

The proof is straightforward using Theorem 1. Computing cycle equivalence in directed graphs appears difficult, but we further reduce the problem to that of finding cycle equivalence in a related undirected graph.

Claim 2 *Nodes a and b are cycle equivalent in a strongly connected component S if and only if the corresponding nodes a' and b' are cycle equivalent in the undirected graph G formed from S as follows:*

- *expand S into S' by splitting each node n into nodes n_i, n' and n_o such that inedges to n connect to n_i, outedges from n originate from n_o, and there are directed edges $n_i \rightarrow n'$ and $n' \rightarrow n_o$.*
- *undirect all edges in S' to form G.*

Note that any cycle in S has a corresponding cycle in G, and although G contains cycles not in S, these cycles do not affect the cycle equivalence relation on nodes in G that correspond to nodes in S. Our algorithm for finding undirected cycle equivalence is based on depth-first search and runs in $O(E)$ time; the details are omitted. In a companion paper, we show that this algorithm can be used to construct the factored control dependence graph of a program in $O(E)$ time. A surprising aspect of this algorithm is that it does not require the computation of the dominator or postdominator relations.

3.2 The DFG construction algorithm

Suppose single-entry single-exit regions are discovered by the algorithm sketched above. The following steps outline the DFG construction algorithm and are illustrated in Figures 2(b)-(c). In the example, each assignment statement is a single-entry single-exit region, as is the `if-then-else` construct.

1. *Determine the variables defined within each single-entry single-exit region.* This is accomplished by an "inside-out" traversal of the regions. This information will be used in a later step to allow dependence flow paths to bypass regions not relevant to the dependence path. In our example, each assignment statement defines one variable, and the `if-then-else` defines `y`.
2. *Create a base-level DFG with no region bypassing.* Simply insert V dependence edges in parallel with each control flow edge. Figure 2(b) shows the base-level DFG.
3. *Perform region bypassing using the information found in Step 1.* Use a forward flow algorithm that maintains the most recent dependence source for each variable. When a region is bypassed, some dependences are cut.
4. *Remove dead flow edges generated by bypassing.* Use a backward propagation starting from edges cut during

the region bypassing. Figure 2(c) shows the graph after region bypassing and dead edge removal.

3.3 Discussion

Multiedges: Due to region bypassing, it is often the case that many DFG edges for a single variable share a common source. For example, in Figure 2(c) two dependence edges start at the assignment `x:=1`. We find it convenient to refer to such a collection as a *multiedge*. We refer to the source of a multiedge as its *tail*, and its successors as *heads*. We will use the term *edge* to refer to the tail and a particular head of a multiedge. From Theorem 1, it follows that the tail and all the heads of a multiedge are totally ordered by dominance/postdominance. In the following sections on dataflow analysis using DFGs, our use of multiedges allows us to separate the flow of information between a node and its dependence flow successors into two parts: propagation between a node and a multiedge tail, and propagation between a multiedge tail and its multiple heads.

Control edges: It is convenient to ensure that the DFG is connected and rooted at `start`. To do this, we introduce a dummy variable defined at `start` and used in each statement that has no other variables on its right hand side. Note that these additional edges are simply control edges indicating a node's control dependence region. In Figure 2(c), the dependences for this dummy variable are indicated by a solid arc with a circle.

Region Bypassing: Bypassing single-entry single-exit regions of the control flow graph is useful because it speeds up optimization. However, the DFG-based optimization algorithms described in this paper work correctly even if some or no bypassing at all is performed. Abstractly, this means that any equivalence relation on CFG edges that is finer than control dependence equivalence can be used to construct the DFG. For example, we can use a relation in which two edges are equivalent if and only if they are in the same basic block — this will permit bypassing of assignment statements but not of control structures.

Constructing the SSA Representation: If the SSA representation of a program is desired, we can construct it in $O(EV)$ time by first building the DFG representation and then eliding switches and converting merges to ϕ-functions. Unlike the standard algorithm [CFR+89], our algorithm does not require computation of the dominance relation or dominance frontiers and is therefore much simpler to implement.

4 Forward dataflow analysis

In this section, we present constant propagation as an example of forward dataflow analysis using the DFG.

Consider Figure 3(a). The first use of `z` can be replaced by `1` and the second by `2`. The right hand sides of the two definitions of `x` can now be simplified to the constant `3`, and the final use of `x` can be replaced by `3`. Most constant propagation algorithms in the literature, such as the def-use chain algorithm [ASU86], discover such *all-paths constants*. However, additional constants may be found if we ignore definitions inside dead regions of code. In Figure 3(b), the predicate of the conditional can be determined to be constant. By ignoring the definition on the unexecuted branch, the use of `x` in the last statement can be determined to have value 1. Such *possible-paths constants* are common in code generated from inline expansion of procedures or macros [WZ85], but algorithms that use def-use chains alone do not find these constants.

```
if (p) then
   { z := 1;
     x := z+2 }
else
   { z := 2;
     x := z+1 }
y := x
```

(a) all-paths

```
p := true
if (p) then
   { x := 1 }
else
   { x := 2 }
y := x
```

(b) possible-paths

Figure 3: Examples of Constant Propagation

We will discuss the standard CFG algorithm, which solves a set of dataflow equations in the control flow graph, and the DFG algorithm, which solves a set of equations in the dependence flow graph. Both algorithms find all-paths and possible-paths constants, but the DFG algorithm is asymptotically faster by a factor of $O(V)$.

We use Kildall's framework for constant propagation [Kil73]. Define a lattice consisting of all constant values and two distinguished values $\top$ and $\perp$. Uses of variables are assigned values from the lattice during constant propagation. Initially, each use is mapped to $\perp$, meaning that we have no information yet about the values of the variable at runtime. A use is mapped to $\top$ when the algorithm cannot determine that the use is a constant (e.g. if the use is reached by two definitions whose right hand sides are `3` and `4`.) At the end of constant propagation, the interpretation of the lattice value assigned to a use of a variable `x` is as follows:

- $\perp$ This use was never examined during constant propagation; it is dead code.
- c This use of `x` has the value c in all executions.
- $\top$ This use of `x` may have different values in different executions.

4.1 The CFG algorithm

At each edge, we maintain a vector of lattice values having an entry for each variable. Intuitively, these vectors summarize the possible values of variables at each program point. These vectors are initialized to $\sigma_\perp$, the vector with $\perp$ in every entry, and they are updated monotonically as the algorithm

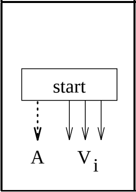
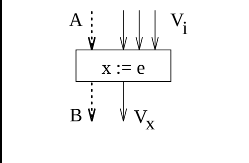
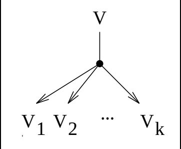
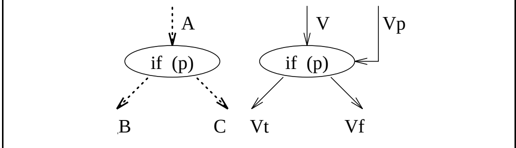
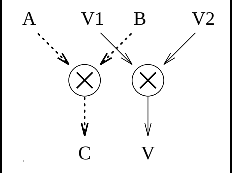
start	assignment	multiedge	switch	merge
				
$\sigma_{\mathrm{A}} = \sigma_{\top}$	$\sigma_{\mathrm{B}} = \sigma_{\mathrm{A}}[\mathtt{x} \mapsto \mathtt{e}\{\sigma_{\mathrm{A}}\}]$		$\sigma_{\mathrm{B}} = \begin{cases} \sigma_{\mathrm{A}} & \textbf{if } \mathrm{p}[\sigma_{\mathrm{A}}] = \mathit{true} \lor \mathrm{p} = \top \\ \sigma_{\perp} & \text{otherwise} \end{cases}$ $\sigma_{\mathrm{C}} = \begin{cases} \sigma_{\mathrm{A}} & \textbf{if } \mathrm{p}[\sigma_{\mathrm{A}}] = \mathit{false} \lor \mathrm{p} = \top \\ \sigma_{\perp} & \text{otherwise} \end{cases}$	$\sigma_{\mathrm{C}} = \sigma_{\mathrm{A}} \sqcup \sigma_{\mathrm{B}}$
(a) control flow graph dataflow equations				
$\mathrm{V_i} = \top$	$\mathrm{V_x} = \mathtt{e}\{\mathrm{V_i}\}$	$\mathrm{V}_i = \mathrm{V}$	$\mathrm{Vt} = \begin{cases} \mathrm{V} & \textbf{if } \mathrm{Vp} = \mathit{true} \lor \mathrm{Vp} = \top \\ \perp & \text{otherwise} \end{cases}$ $\mathrm{Vf} = \begin{cases} \mathrm{V} & \textbf{if } \mathrm{Vp} = \mathit{false} \lor \mathrm{Vp} = \top \\ \perp & \text{otherwise} \end{cases}$	$\mathrm{V} = \mathrm{V1} \sqcup \mathrm{V2}$
(b) dependence flow graph dataflow equations				

Figure 4: Dataflow Equations for Constant Propagation

proceeds.

Computing these vectors can be viewed as solving a system of dataflow equations. One equation describing the output vectors(s) in terms of the input vector(s) is associated with each node. Figure 4(a) shows the equation scheme for constant propagation on the control flow graph. In this figure, σ_{A} represents the vector of lattice values stored at edge A. Since the values of variables at `start` are unknown, we set the vector at `start` to $\top$ for all variables. The output vector of an assignment statement `x:=e` is obtained by evaluating expression `e` using the values available at the statement input, and updating the `x` entry of the output vector to reflect this value. Expression `e` evaluates to $\perp$ (or $\top$) if any operand of `e` is $\perp$ (or $\top$) in the input vector. At a `switch`, the output vector is identical to the input vector for each direction that control flow can take; the output vector is $\sigma_{\perp}$ for directions that control flow cannot possibly take. At a `merge`, the output vector is simply the least upper bound of the input vectors.

These equations can be solved using a standard worklist algorithm. Unfortunately, the asymptotic complexity of this algorithm is poor. If V is the number of program variables and E is the number of CFG edges, then the algorithm requires $O(EV)$ space and $O(EV^2)$ time. The inefficiency arises because lattice values must be propagated along control flow paths from definitions of variables to their uses.

4.2 The DFG algorithm

The DFG algorithm propagates values for each variable separately along edges in the DFG. The set of DFG dataflow equations has one equation for each DFG edge. As discussed in Section 2, a DFG edge d can be viewed as representing a pair of CFG edges (e_1, e_2), and the dataflow equation for d computes the lattice value of the associated variable in the region between e_1 and e_2. Figure 4(b) shows the equation scheme for solving constant propagation using the DFG. These equations are similar to the ones for the control flow graph algorithm; the only new feature is the rule that propagates the value at the tail of a DFG multiedge to its heads. For example, in the program of Figure 1(c), this rule can be used to propagate the value of `x` from the assignment `x:=1` to the two uses of `x` in the program.

As with the control flow algorithm, a simple worklist-based algorithm can be used to solve the dataflow equations. Whereas the control flow algorithm performed $O(V)$ work each time a node is processed, the DFG algorithm performs work only for the relevant dependences at each node. Therefore, the asymptotic complexity of the DFG algorithm is $O(EV)$. In addition, the algorithm avoids propagating information through single-entry single-exit regions in which there are no assignments to the relevant variable.

A variety of enhancements to this algorithm are possible. For example, the Multiflow compiler performed predicate analysis to determine additional constants: if the predicate at a switch is `x=1`, we can propagate the constant `1` for `x` on the true side of the conditional even if we cannot determine the value of x for the false side [LFK+93]. It is easy to extend both the DFG and CFG algorithms to accomplish this, but this extension seems difficult in SSA-based algorithms [WZ91] since SSA edges bypass switches in the CFG.

We omit the proof of correctness of the DFG algorithm. Given the structural properties of DFG edges, it is a simple matter to project values from the DFG onto the corresponding CFG edges and then show that these values are consistent with the values determined by the CFG algorithm.

5 Backward dataflow analysis

In this section, we describe the use of the DFG in performing backward dataflow analysis, using the computation of *anticipatable expressions* as an example. We then show how anticipatable expressions can be used in a powerful optimization called *elimination of partial redundancies*, which subsumes common subexpression elimination and loop-invariant removal [MR79].

5.1 Anticipatability

Definition 8 *An expression e is* **totally (partially) anticipatable** *at a point p if, on every (some) path in the CFG from p to* `end`*, there is a computation of e before an assignment to any of the variables in e. We denote total (partial) anticipatability as ANT (PAN).*

ANT and PAN are usually computed for all expressions in the program simultaneously, but we will focus attention on a single expression to keep the discussion simple. The CFG equations for the computation of anticipatability are shown in Figure 5. The solution of the equations for ANT can be obtained iteratively, starting with an initial approximation in which ANT is true everywhere in the program except at `end`. This initial approximation permits ANT to propagate through loops correctly while ensuring that the "boundary condition" at `end` is satisfied. Similarly, the equations for PAN can be solved iteratively, starting with an initial approximation in which PAN is false everywhere in the program.

We now discuss the solution of dataflow equations for ANT and PAN in the DFG. We first discuss expressions involving a single variable (such as `x+1, y*3`) and then generalize to multivariable expressions (such as `x+y, x*y`).

Figure 5 shows the DFG equations for ANT and PAN computations for an expression `x+1`. These equations are similar to the CFG equations. The rule for multiedges propagates anticipatability information from the heads to the multiedge tail. The intuition behind this rule is the following: by the definition of the DFG, the heads of a multiedge postdominate its tail, and there can be no definitions of variable x in the portion of the CFG between the tail and any of the heads; therefore, if the expression is totally (partially) anticipatable at any head, then it is also totally (partially) anticipatable at the tail. Another way of looking at this is that the tail of a multiedge accumulates the contributions to anticipatability from nodes with the same set of control dependences.

In the solution of the CFG equations for ANT, the initial approximation has ANT true everywhere except at `end` which provides the "boundary condition" where ANT is false. DFG edges do not go to `end`, but the role of `end` in providing the boundary condition can be played by statements that use x but do not compute the expression `x+1` — dependences for x at these statements are set to false. Similarly, if a variable x is live on one side of a conditional branch but dead on the other, then the dependence for x is initialized to false on the dead side of the switch.

Once the DFG propagation is done, the values of ANT at points in the CFG can be found by projecting from the DFG into the CFG: simply set ANT to true at every point in the the single-entry single-exit region between the head and tail of every dependence edge for which ANT is true at the head.

An example of ANT computation is shown in Figure 6. We start with ANT being true at all DFG edges except at expressions other than `x+1` that use x; in the example, ANT is true at all edges except d4, where it is false. The values at d4 and d5 are combined together by the multiedge rule, so ANT at d2 is true. Since ANT is true at d6, ANT at d3 remains true. ANT at d2 and d3 are true, so ANT is true at d1. Projecting ANT onto the CFG sets ANT to true at every point between the definition of x and the two computations of `x+1`.

We can reduce the problem of computing ANT for multivariable expressions like `x+y` to the single-variable ANT discussed above as follows.

Definition 9 *An expression e is* **totally anticipatable relative to variable** x *at a point p if, on every path in the CFG from p to* `end`*, there is a computation of e before an assignment to* x*.*

It follows that an expression is totally anticipatable at a point p if it is totally anticipatable relative to *all* of its variables at p. A similar notion can be defined for partial anticipatability.

To compute ANT for `x+y`, we initialize all dependences for x and y to true, except for the boundary dependences which are set to false. Propagation along dependences for x and y proceeds independently using the single-variable rules described earlier. Once DFG propagation is completed, we project ANT relative to x and ANT relative to y onto the CFG edges as before, and assert that ANT is true wherever it is true relative to both x and y separately.

Figure 7 gives an example of multivariable ANT for the expression `x+y`. We solve the single-variable ANT problems relative to x and y independently. Solving single-variable ANT relative to x yields true at d1 and d3 and false at d2; projecting the results onto control flow edges yields ANT true relative to x at e2 through e7. Single-variable ANT relative to y is true at d6 through d8 and is false at d4 and d5; projecting the results onto control flow edges yields ANT

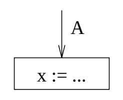
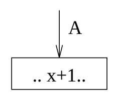
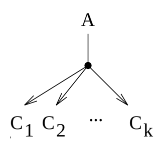
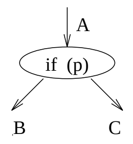
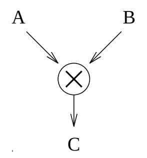
statement	expression	multiedge	switch	merge
				
$\mathrm{ANT_A} = false$ $\mathrm{PAN_A} = false$	$\mathrm{ANT_A} = true$ $\mathrm{PAN_A} = true$		$\mathrm{ANT_A} = \prod \mathrm{ANT_{B,C}}$ $\mathrm{PAN_A} = \sum \mathrm{PAN_{B,C}}$	$\mathrm{ANT_{A,B}} = \mathrm{ANT_C}$ $\mathrm{PAN_{A,B}} = \mathrm{PAN_C}$
(a) control flow graph dataflow equations				
	$\mathrm{ANT_A} = true$ $\mathrm{PAN_A} = true$	$\mathrm{ANT_A} = \sum \mathrm{ANT_{C_i}}$ $\mathrm{PAN_A} = \sum \mathrm{PAN_{C_i}}$	$\mathrm{ANT_A} = \prod \mathrm{ANT_{B,C}}$ $\mathrm{PAN_A} = \sum \mathrm{PAN_{B,C}}$	$\mathrm{ANT_{A,B}} = \mathrm{ANT_C}$ $\mathrm{PAN_{A,B}} = \mathrm{PAN_C}$
		$\mathrm{PP_A} \overset{def}{=} \sum_{i \neq j} \mathrm{ANT_{C_i}} \cdot \mathrm{PAN_{C_j}}$ $\mathrm{AV_{C_i}} = \mathrm{PP_A}$	$\mathrm{AV_{B,C}} = \mathrm{AV_A}$	$\mathrm{PP_{tail(A)}} \overset{def}{=} \mathrm{ANT_C} \cdot \mathrm{AV_B}$ $\mathrm{PP_{tail(B)}} \overset{def}{=} \mathrm{ANT_C} \cdot \mathrm{AV_A}$ $\mathrm{AV_C} = \mathrm{AV_A} \cdot \mathrm{AV_B}$
	$\mathrm{DELETE_A} \overset{def}{=} \mathrm{AV_A}$	$\mathrm{INSERT_A} \overset{def}{=} \mathrm{PP_A} \cdot \sim \mathrm{AV_A}$		
(b) dependence flow graph dataflow equations				

Figure 5: Dataflow Equations for ANT/PAN and EPR

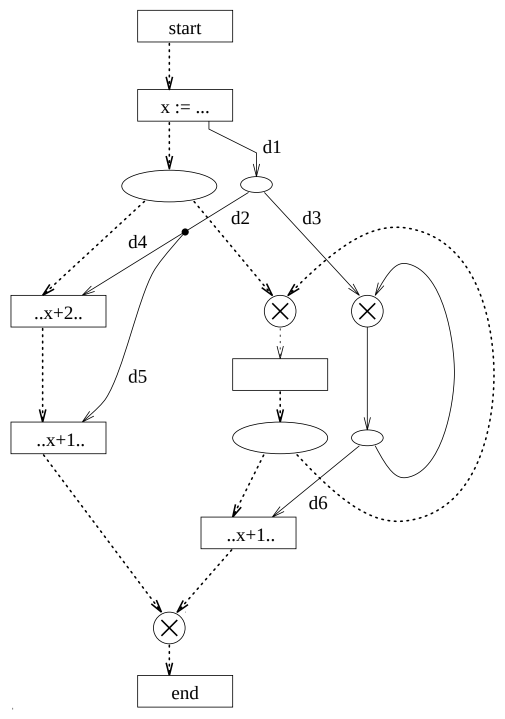


Figure 6: An Example of Single-Variable Anticipatability

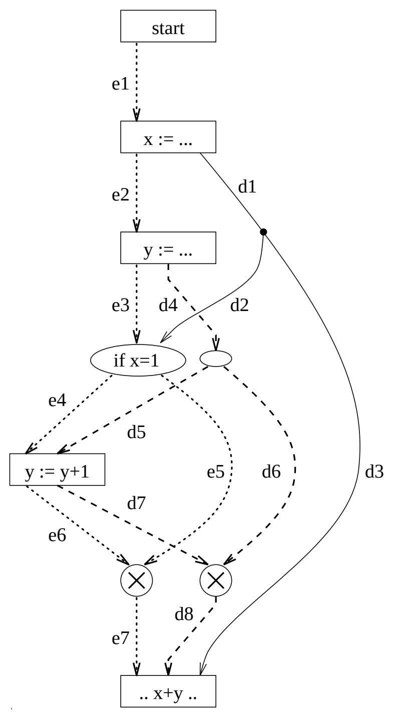


Figure 7: An Example of Multivariable Anticipatability

true relative to y at e5 through e7. We combine these separate results to get x+y anticipatable at e5, e6 and e7.

In our approach, the propagation of ANT occurs only in that portion of the program in which at least one of the variables in the expression is live. More elaborate approaches are possible. For example, we can avoid propagating false from expressions other than x+y that use x or y by computing ANT in two phases as is done in some CFG-based dataflow analyses [DRZ92]. It is also possible to compute ANT directly by simultaneously following dependences for all variables used in the expression, relying on a depth-first numbering scheme to order these dependences. We will not discuss these more complex alternatives any further due to lack of space.

5.2 Elimination of partial redundancies

Partial and total anticipatability can be used in a powerful optimization called *elimination of partial redundancies* (epr), which subsumes common subexpression elimination and loop-invariant removal. A computation is said to be redundant if it follows a computation of the same value on some execution path. If a redundant computation is preceded by computations of the same value on *all* execution paths, we say the computation is *totally* redundant; otherwise we say the computation is *partially* redundant. A classic dataflow algorithm for the removal of partial redundancies is due to Morel and Renvoise [MR79]. We complete our discussion of DFG-based analysis by showing how we can implement epr.

The basic idea is to insert new computations into the CFG where it is safe and profitable to do so, thereby making partially redundant computations totally redundant. Totally redundant computations may be replaced by a use of a new temporary variable that is properly assigned at the preceding computations. After removing these totally redundant computations, no execution path will contain more instances of a computation than it did originally, and some paths will contain fewer instances of the computation.

A program point is safe for insertion of a computation of e if e is anticipatable at that point, but what about profitability? Inserting computations of e wherever it is anticipatable, and then deleting computations of e wherever it has become available eliminates partial redundancies; however, this strategy may perform superfluous code motion. For example, in Figure 6 this strategy will place x+1 immediately after the assignment to x and delete the other computations of x+1, even though there is no redundancy in the original program. There has been much discussion in the literature about code motion strategies [DS88, Dha91, KRS92], but to our knowledge there is no experimental data showing the superiority of any single strategy.

Our approach to epr has the virtue of simplicity. Figure 5 shows the dataflow equations for the dependence-based algorithm. ANT and PAN are backward dataflow problems, while AV is a forward problem. PP, INSERT, and DELETE are local definitions that do not propagate. The rationale behind these rules is as follows. Once we have computed ANT and PAN as described in Section 5.1, we determine where it may be profitable to place (PP) computations. There are two rules. The merge rule inserts a computation into a region if it is anticipatable and partially available at the output of the merge — after insertion, the expression becomes totally available at the output of the merge, so computations below the merge can be deleted. The multiedge rule eliminates redundancies within a control region: it is profitable to place a computation at the tail of a multiedge if the expression is anticipatable at the tail and partially anticipatable at two or more heads. This is equivalent to saying that placement is profitable at the tail if the computation is totally anticipatable at one head and partially anticipatable at another head. Finally, we insert a computation at a point if placement is profitable but the expression is not available, and we delete computations where the expression is available.

The rule for multiedges can be refined further to fine-tune the placement of code. Instead of hoisting a computation to the tail of the multiedge, it may be desirable to place it at the head where the value of the expression is first used in some other computation, since that is the earliest place where the computation must be available. This kind of refinement is easy to do in the DFG since the dependence edges tell us exactly where we must look to find the desired information. We are evaluating these heuristics and we refer the interested reader to the forthcoming thesis of one of the authors [Joh93].

Our epr algorithm is simple in part because it is edge-based rather than node-based like conventional presentations of epr. Placing computations at nodes is complicated by the presence of control flow edges whose source is a switch and whose destination is a merge, such as the back edge of repeat-until loops. This complication is eliminated by adding empty basic blocks to split such edges [MR79], but these blocks must later be removed if no code is moved into them. DFG algorithms are naturally edge-based and avoid these complications. Our epr algorithm propagates information only through the portion of the control flow graph where the variables in the expression are live. It can also skip over single-entry single-exit regions of the control flow graph where there is no definition or use of the variables in the expression. This is not possible in CFG-based approaches. Finally, the DFG is built only once prior to optimization (it is, of course, updated as optimization is performed). This approach is simpler to implement than other approaches that build a special-purpose graph for *each expression* for which partial redundancies must be eliminated [DRZ92].

6 Conclusions

In this paper, we have presented an approach to speeding up program analysis and optimization that is based on the use of the dependence flow graph. First, we gave a graph-theoretic characterization of the DFG that tied together related work on def-use chains, static single assignment form, and control dependences. Then we sketched a fast algorithm for constructing the DFG, which is of independent interest because it can be used to construct a factored control dependence graph of a program in $O(E)$ time, a factor of N improvement over the best existing algorithm. Finally, we showed how the DFG can be used for both forward and backward dataflow analyses. Our approach avoids inefficiencies by (1) propagating information for a variable x only where needed, bypassing single-entry single-exit regions of the graph which contain no definition of x, and by (2) performing work proportional to the number of variable references at each assignment statement.

We have not addressed the problem of optimization in stages described in Section 1. Note that performing redundancy elimination in “dependence order” in the example of Section 1 achieves the desired ordering. The general picture is more complex because of merges, but we believe that a dependence-based approach is the right one.

In this paper, we have focused on the use of the DFG for optimizations. For parallelization, the simple picture of the DFG in this paper can be extended to include aliasing, data structures, anti- and output dependences, loop recognition, and distance/direction information for loop-carried dependences. Our treatment of aliasing, and anti- and output dependences is discussed in an earlier paper [BJP91]. We are implementing a DFG tool-kit for parallelization and optimization, and we will report on experimental results in another paper. We conclude with a discussion of what we have learned so far from our implementation.

First, we have found that it is neither necessary nor desirable to use a full-blown dataflow graph representation of imperative language programs as an intermediate form. Our current representation retains control flow information and permits us to expose only relevant dependences in any phase of the compiler; for example, an optimization phase could expose only def-use information, as we have done in this paper. We are aware of successful functional language compilers (for dataflow machines) that represent programs as full-blown dataflow graphs, but we attribute their success to the relative simplicity of functional languages and the closeness between the intermediate and machine languages.

Second, we believe that renaming of variables to accomplish single assignment (static or dynamic) is orthogonal to dependence representation. In our opinion, single assignment has nothing to do with the DFG (or for that matter, static single assignment form) — it is best to view these representations as a way of knitting control dependence information with def-use information. This view has two advantages. First, aliasing can be handled very simply [BJP91]. Second, control dependences can also be combined with anti- and output dependences without the need for a new conceptual framework.

Finally, in the context of optimization, control dependence *equivalence* is more important than control dependences per se. The construction of the DFG requires an equivalence relation on control flow edges, and control dependence equivalence is the coarsest equivalence relation that can be used for this purpose. Realizing this led us to invent an algorithm that computes control dependence equivalences directly, and this in turn led us to an $O(E)$ algorithm for constructing a factored control dependence graph.

To place our work in perspective, it is useful to understand the differences between the DFG and the program dependence graph (PDG) [FOW87]. The PDG of a program is the union of its control and data dependences. There have been many efforts to give a formal semantics to the PDG, with the objective of using the semantics in correctness proofs of program transformations [HPR88, Sel89, CF89]. However, this has proved to be difficult. For example, it has been shown that two programs with the same PDG have the same input-output behavior, but the proof is long and intricate even for structured programs. Other efforts give a constructive definition of the PDG by transforming the denotational semantics of an imperative language, but the construction and the final result are hard to decipher. These difficulties in giving semantics to a program, once it has been broken down into its control and data dependences, are not unlike the difficulties in giving semantics to a program once it has been broken down into assignments and GOTOs. Like quarks in nuclei [GM64] or conditional jumps in the stored program computer, control dependence is a deep and fundamental notion. However, quarks do not exist in isolation, and conditional jumps are implicit and hidden in modern programming language control structures. Similarly in the context of optimization, control and data dependences should be fused together to give structure to dependences as we have done in the DFG. It is straightforward to give a semantics to the DFG if one is desired, but more importantly, such a semantic account is unnecessary since DFG edges have precise structural properties that can be used in correctness proofs. Put simply, the DFG gives a way of propagating information in the control flow graph while bypassing uninteresting regions, and the amount of bypassing is a useful compromise between too much and too little.

7 Acknowledgements

Thanks to Mayan Moudgill for early work on DFG-based optimizations and to Micah Beck for his stimulating input throughout the entire project. Thanks to Richard Huff, Wei Li, Mayan Moudgill, Paul Stodghill and Mark Charney for their comments on the paper.

References

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.

[Bec92] Micah Beck. *Translating FORTRAN to Dataflow Graphs*. PhD thesis, Department of Computer Science, Cornell University, May 1992.

[BJP91] Micah Beck, Richard Johnson, and Keshav Pingali. From control flow to dataflow. *Journal of Parallel and Distributed Computing*, 12:118–129, 1991.

[BMO90] Robert A. Ballance, Arthur B. Maccabe, and Karl J. Ottenstein. The Program Dependence Web: A representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 257–271, June 20–22, 1990.

[CCF91] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages*, pages 55–66, January 21–23, 1991.

[CF89] Robert Cartwright and Matthias Felleisen. The semantics of program dependence. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 13–27, June 21–23, 1989.

[CFR$^+$89] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 25–35, January 11–13, 1989.

[CFR$^+$91] 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 Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[CFS90] Ron Cytron, Jeanne Ferrante, and Vivek Sarkar. Compact representations for control dependence. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 337–351, June 20–22, 1990.

[Dha91] Dhananjay M. Dhamdhere. Practical adaptation of the global optimization algorithm of Morel and Renvoise. *ACM Transactions on Programming Languages and Systems*, 13(2):291–294, April 1991.

[DRZ92] Dhananjay M. Dhamdhere, Barry K. Rosen, and F. Kenneth Zadeck. How to analyze large programs efficiently and informatively. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 212–223, June 17-19, 1992.

[DS88] K.-H. Drechsler and M. P. Stadel. A solution to a problem with Morel and Renvoise's "Global Optimization by Suppression of Partial Redundancies". *ACM Transactions on Programming Languages and Systems*, 10(4):635–640, October 1988.

[FOW87] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependency graph and its uses in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, June 1987.

[GM64] Murray Gell-Mann. A schematic model of baryons and mesons. *Physics Letters*, 8(3):214–215, February 1964.

[HPR88] Susan Horwitz, Jan Prins, and Thomas Reps. On the adequacy of program dependence graphs for representing programs. In *Conference Record of the 15th Annual ACM Symposium on Principles of Programming Languages*, pages 146–157, January 13–15, 1988.

[Joh93] Richard Johnson. *Dependence-Based Compilation (working title)*. PhD thesis, Department of Computer Science, Cornell University, 1993. Expected in September.

[Kas75] V. N. Kas'janov. Distinguishing hammocks in a directed graph. *Soviet Math. Doklady*, 16(5):448–450, 1975.

[Kil73] Gary A. Kildall. A unified approach to global program optimization. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 194–206, October 1–3, 1973.

[KRS92] Jens Knoop, Oliver Rüthing, and Bernhard Steffen. Lazy code motion. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 224–234, June 17-19, 1992.

[LFK$^+$93] P. Geoffrey Lowney, Stefan M. Freudenberger, Thomas J. Karzes, W. D. Lichtenstein, Robert P. Nix, John S. O'Donnell, and John C. Ruttenberg. The Multiflow trace scheduling compiler. *Journal of Supercomputing*, 7(1/2), January 1993.

[MR79] Etienne Morel and Claude Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, February 1979.

[PBJ$^+$91] Keshav Pingali, Micah Beck, Richard Johnson, Mayan Moudgill, and Paul Stodghill. Dependence Flow Graphs: An algebraic approach to program dependencies. In *Conference Record of the 18th Annual ACM Symposium on Principles of Programming Languages*, pages 67–78, January 21–23, 1991.

[RT81] John H. Reif and Robert E. Tarjan. Symbolic program analysis in almost-linear time. *SIAM Journal on Computing*, 11(1):81–93, February 1981.

[Sel89] Rebecca P. Selke. A rewriting semantics for program dependence graphs. In *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*, pages 12–24, January 11–13, 1989.

[SS70] R. M. Shapiro and H. Saint. The representation of algorithms. Technical Report CA-7002-1432, Massachusetts Computer Associates, February 1970.

[WZ85] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. In *Conference Record of the 12th Annual ACM Symposium on Principles of Programming Languages*, pages 291–299, January 14–16, 1985.

[WZ91] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.