Sparse Data-Flow Analysis for DAG Parallel Programs *

Eric Stoltz and Michael Wolfe

Department of Computer Science and Engineering
Oregon Graduate Institute of Science & Technology
P.O. Box 91000
Portland, OR 97291-1000
(503) 690-1121 ext. 7404
{stoltz,mwolfe}@cse.ogi.edu

October 14, 1994

Abstract

We contrast abstract flow of control in a sequential program environment which uses a control flow graph (CFG) to the abstraction of execution order within DAG parallelism using a simple precedence graph (PG). We note that often the analogous concepts are *duals* of each other with regard to universal and existential quantifiers.

We are studying sparse data-flow analysis techniques, which include methods of placing operators at confluence points to merge the flow of information. While the placement of merge operators is well-known for CFGs (at the iterated dominance frontier of nodes with non-identity transfer functions), we determine and prove correct placement points for merge operators within a PG in the case of reaching definitions. We also show how to conservatively implement the placement of parallel merge operators in an efficient manner with respect to the reaching definitions problem.

1 Flow of Information within Sequential Programs

1.1 An Abstract Representation

To describe control flow of sequential programs, the abstraction of the *Control Flow Graph* (CFG) is used. The CFG summarizes potential paths of control through the program. Formally, the CFG is a graph G = <*V,E,Entry,Exit*>, where V is a set of nodes representing basic blocks in the program, E is a set of edges representing sequential control flow in the program, and *Entry* and *Exit* are nodes representing the unique entry point into the program and the unique exit point from the program. The CFG has been the most common intermediate representation for program flow analysis for a number of years [1, 9].

Data-flow analysis is a method of collecting information about a program, such as reaching definitions, dominance, or live variables [1, 10]. The first two examples of data-flow analysis,

*Supported in part by NSF grant CCR-9113885 and a grant from Intel Corporation and the Oregon Advanced Computing Institute.

reaching definitions and dominance, are *forward* data-flow problems, since the flow of information is in the same direction as control flow. Live variable analysis is an example of a *backward* data-flow problem, since the information at the end of each basic block is dependent upon the information at the entry of control flow successors. Thus, with backward data-flow problems information flows in the direction opposite that of control flow. A transfer function at each node summarizes the data-flow effect about the desired information at that node. A typical method of extracting information from the CFG is to iterate a set of data-flow equations until the information available at each point (typically the beginning and end of each control flow node) has converged. For this paper, we will focus on the forward flow of information.

When paths of control within the CFG converge at a confluence node (a node with more than one predecessor), data-flow analysis needs to merge the incoming information from each of its data-flow predecessors. Merging information at these points can lose precision, but may be unavoidable when attempting to keep the quantity of stored information to a minimum. When a confluence node in the CFG is reached, exactly one of the predecessors of that node will have just been executed. An important problem is to prevent quadratic growth of information, which can occur with reaching definitions. In the case of a CFG where n definitions and n uses are separated by a confluence node, n^2 reaching definition chains are generated to connect each definition to its possible uses. This problem has been discussed in detail elsewhere [4, 5].

In this paper we focus on the *reaching definitions problem* as a running example of a forward data-flow problem: for each use (or *fetch*) of a variable, what are the possible definition points that reach this use? In particular, the rest of this section examines the notion of *killing* data-flow information within sequential control flow graphs, while the next section looks at representation and execution order for parallel precedence graphs. Section 3 reviews sparse data-flow representations, methods for merging reaching information within sequential graphs, and where to place merge operators to accomplish this task. Section 4 examines these questions for parallel precedence graphs, identifying the minimal set of nodes at which to place parallel merge operators: given a set of nodes, $\mathcal{S}$, we present a new concept, the *meet* of $\mathcal{S}$, defined in terms of reachability and when iterated is shown to be equal to that minimal set. We also provide an efficient method for parallel merge operator placement, proving that it is a safe, albeit conservative, technique. Finally, Section 5 summarizes and discusses future directions.

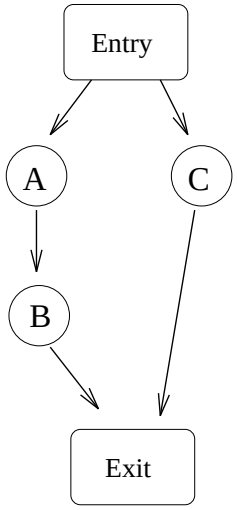
1.2 The Dominance Relation and Shields

Information (reaching definitions, for our running example) along a straight-line path in a CFG can be killed by another definition for the same variable. If nodes X and Y both lie along a straight-line path with X preceding Y, and both define variable v, then the definition at Y kills any definitions at X. Note that (degeneratively, in this case) every path to Y passes through X. Although non-killing definitions of variables exist, such as assignments to indexed variables or arbitrary procedure parameters passed by reference, we do not discuss these concepts in this paper (except for special pseudo-assignment merge operators explained later).

The ideas of the preceding paragraph are now formalized. Within a CFG, node B *dominates* node C if every path from *Entry* to C must pass through B, and is written as $B\ dom\ C$. If every path from A to C must pass through B, we say that B *shields* A from C, or that $B \in shield(A,C)$. We note that $B\ dom\ C$ is just a particular instance from the set of shields for *Entry* and C, where $B \in shield(Entry,C) \Longrightarrow B\ dom\ C$.

```
   ...
Parallel Sections
Section A
   ...
Section B, Wait(A)
   ...
Section C
   ...
End Parallel Sections
   ...
```

(a)



(b)

Figure 1

Shielding is important when dealing with the flow of information in a CFG, since $B \in$ *shield*(A,C) means that B can kill any information generated from A on all paths to C. In particular, any definition of v within B prevents all definitions of v in A from reaching C. It is not our intent to closely examine the properties of shielding in this paper, but to highlight concepts pertaining to the flow of information in a sequential flow graph which have analogous concepts within the context of parallel execution order.

2 Parallel Execution Order

A precedence graph is an abstraction which imposes order of execution among its nodes. Precedence graphs can easily be used to express DAG parallelism [6] by using `Wait` clauses to enforce constraints between section nodes. A precedence graph is also a simplified, special case of a Parallel Program Graph [12] which only contains *synchronization* edges, with the synchronization condition that all code in a node completes before beginning execution of any successor.

For the purposes of this paper, we deal with DAG parallelism (a subset of *task* parallelism [7]), specifically explicit parallel sections fashioned after the `Parallel Sections` construct [11], which is similar to the `cobegin-coend` syntax of Brinch Hansen [2]. An example is shown in Figure 1(a), where `Section B, Wait(A)` means that all code in Section A must complete before the code in Section B may begin.

2.1 An Abstract Representation

The ordering of sections is arranged within a *precedence graph* (PG), an abstract representation which dictates what sections may execute in what order. Formally, a PG is a directed graph

P = $< V_P, E_P, Entry_P, Exit_P >$, where V_P is a set of nodes, each representing a section in a parallel block, E_P is the set of edges which represent wait-dependence arcs (corresponding to the `Wait` syntax described above), and $Entry_P$ and $Exit_P$ are the *co-begin* and *co-end* nodes, respectively. We will always show the *Entry* node in a PG, but will often omit the *Exit* node, since for purposes of illustration only a partial representation of the PG is usually needed, and the *Exit* node is seldom a factor concerning the forward flow of information (which we focus on in this paper) through the PG. The wait-dependence arcs impose a partial order upon the nodes of a PG. If there is no partial order between two sections, they may execute in any order relative to each other – perhaps in parallel. An example precedence graph is shown in Figure 1(b), where sections A and C (and, possibly, B and C) might execute concurrently.

We note that a PG must be acyclic, since any cycles would create a deadlock. A section node of a PG "uses" or "defines" a variable if any of the code within that section uses or defines that variable.

A confluence node in a PG has quite different semantics than that in a sequential CFG. While precisely one of the predecessors at a confluence node in a CFG will be executed, *all* predecessors of a confluence node in the PG must execute before confluence node itself executes. Essentially, a confluence node is waiting upon all its predecessors, so they must all execute before the confluence node executes. When paths meet within a PG, information might also merge. It is important to note that it is possible that merging information could be in conflict – since all predecessors are executed, we could have multiple definitions of the same variable, for instance.

We also note an important property of precedence graphs – they are insensitive to transitive edges. This will become clear when we look at how information flows between section nodes in a precedence graph – with respect to the *reaches* relation for definitions.

2.2 The Reaches Relation for Definitions Within a PG

When does node B wait for node A? When there is path in the precedence graph from A to B, i.e., if A can reach B, then B waits upon A. Since any path from A to B is sufficient, it now becomes clear why the addition of transitive edges to a precedence graph adds no new information. In fact, the *transitive reduction*[†] of a precedence graph contains the smallest number of vertices and edges which captures all the information of the original graph.

Definition 1 (Reaching Definitions Within a PG.) *Within a PG, a definition of v at section node X* reaches *section node Y if no path from X to Y contains a definition of v, except at X or Y.*

Kill information is computed quite differently within a PG as opposed to a CFG [8]. In a PG, a definition of v in node A is killed before reaching node C if *any* path from A to C passes through node B, where B contains a definition of v. We note that the *shield* and *reaches* relations are duals of each other with respect to killing data-flow information:

CFG Information from A can be killed by B before reaching C if *all* paths from A to C pass through B.

[†]The transitive reduction of graph G is any graph G′ with the same vertices as G, but with as few edges as possible, such that the transitive closure of G′ is equal to the transitive closure of G.

PG Information from A can be killed by B before reaching C if *any* path from A to C passes through B.

It is important to contrast two uses of the term "reaches". On the one hand we speak of path reachability; here, A reaches B in a graph if there exists a path from A to B. On the other hand, we will often be referring to reaching definitions, which is concerned with the flow of a particular type of information through a graph. In this case path reachability is not sufficient; definitions (or, more generally, data-flow information) can be killed along paths in a graph due to non-identity transfer functions. Most of the time the meaning is clear from context, but when not, we will attempt to be explicit concerning usage.

3 Merging Information in a CFG

At what points in the CFG do we need to merge information? Certainly not at all confluence nodes, since there may not be any "interesting" nodes along either branch leading to the confluence node, where "interesting" nodes are those with non-identity transfer functions for a given data-flow problem. We define *merge nodes* as nodes at which information converges from more than one predecessor, which are equivalent to the *meet* nodes of sparse evaluation graphs [3]; these identify nodes which merge data-flow information, as opposed to confluence nodes in a control-flow graph. Information is merged at these nodes in a sparse intermediate representation by inserting a merge *operator* to collect the information from each data-flow predecessor.

For reaching definitions, merge operators (known as ϕ-functions) for variable v need to be placed at the *iterated join* of the set of interesting nodes [5]. In this case, interesting nodes are those in which there is a definition to v. The *join* of two nodes is defined as follows:

Definition 2 *The* join *of nodes X and Y, $J(X,Y)$ =*

$$\left\{ Z \mid \exists Z_X, Z_Y \text{ with } Z_X \rightarrow Z \text{ and } Z_Y \rightarrow Z, \text{ paths } p_X : X \xrightarrow{*} Z_X \text{ and } p_Y : Y \xrightarrow{*} Z_Y, p_X \cap p_Y = \emptyset \right\}$$

The join of a set of nodes, $\mathcal{S}$, denoted $\mathrm{J}(\mathcal{S})$, is defined to be the union of the pairwise joins $\forall X, Y \in \mathcal{S}$, i.e., $\mathrm{J}(\mathcal{S}) = \bigcup_{X,Y \in \mathcal{S}} \mathrm{J}(X,Y)$. The iterated join, $\mathrm{J}^+(\mathcal{S})$, is defined as the limit of increasing sequences of nodes defined by:

$$\begin{aligned} \mathrm{J}^1(\mathcal{S}) &= \mathrm{J}(\mathcal{S}) \\ \mathrm{J}^2(\mathcal{S}) &= \mathrm{J}(\mathcal{S} \cup \mathrm{J}^1(\mathcal{S})) \\ \mathrm{J}^{i+1}(\mathcal{S}) &= \mathrm{J}(\mathcal{S} \cup \mathrm{J}^i(\mathcal{S})) \end{aligned}$$

The effect of a ϕ-function for reaching definitions is that it acts as a killing definition for all other definitions which reach that confluence node (the merge operator *collects* the reaching definitions at that point). A ϕ-function also represents a use for the reaching definitions along each incoming control flow edge via its arguments. Thus, all reaching definitions can be discovered by traversing the links associated with each ϕ-function argument.

For reaching definitions, the iterated join of the set of nodes with non-identity transfer functions for variable v is the correct placement for ϕ-function since it insures the following properties:

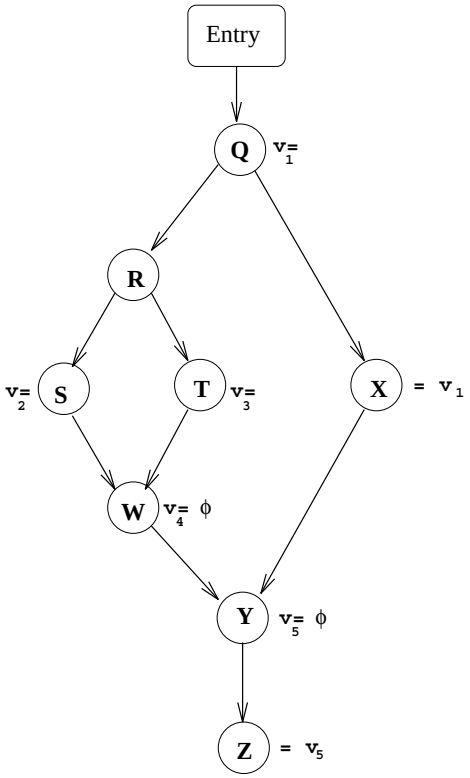


Figure 2

- each join point K for v collects (directly or indirectly) all reaching definitions of v at K.
- each variable use of v at CFG node A will have a single reaching definition. That is, in the set $\mathcal{D} = \mathcal{S} \cup \mathrm{J}^+(\mathcal{S})$, nodes which contain variable definitions or join points of v, there exists exactly one element $E \in \mathcal{D}$, such that E *dom* A and $E \in$ *shield*$(d, A) \forall d \in \mathcal{D}$.

Calculating the iterated join of $\mathcal{S}$, $\mathrm{J}^+(\mathcal{S})$, may seem expensive, but it has been shown [5] to be equal to the *iterated dominance frontier* of $\mathcal{S}$. The dominance frontier (DF) of a node X is all nodes Z such that X does not strictly dominate Z, but X dominates some predecessor of Z. Again, $\mathrm{DF}(\mathcal{S}) = \bigcup_{X \in \mathcal{S}} \mathrm{DF}(X)$, and DF^+ is defined as above for J^+. For the relation $\mathrm{DF}^+(\mathcal{S}) = \mathrm{J}^+(\mathcal{S})$ to hold, the *Entry* node of the CFG must be in $\mathcal{S}$, which is a reasonable assumption for imperative, sequential programs, one that implies that all variables used get defined along any possible path through the program.

We present an example in Figure 2. Killing definitions of v occur at $\mathcal{S} = \{Q, S, T\}$. $\mathrm{DF}^1(\mathcal{S}) = \{W\}$, and $\mathrm{DF}^2(\mathcal{S}) = \mathrm{DF}^+(\mathcal{S}) = \{W, Y\}$. Thus, ϕ-functions are placed at W and Y, creating v_4 and v_5.

We note that $W \in$ *shield(S,Z)* since any killing definition of v placed at W would prevent any definitions at S (or T) from reaching Z. This example also illustrates the properties of $\mathrm{DF}^+(\mathcal{S})$. First, we see that the ϕ-function at W directly collects the two reaching definitions of v (v_2 and

v_3) which reach this confluence point. The ϕ-function at Y directly collects the definition from Q, v_1, but indirectly collects v_2 and v_3 via v_4, the ϕ-function at W. All reaching definitions are preserved either directly or indirectly, but the definition chains are now linearized. Second, each use of v now has exactly one reaching definition. The use of v at X is reached by the definition from Q. Q *dom* X, and, vacuously in this instance, $Q \in shield(d, X) \forall d \in \mathcal{D}$. The use of v at Z is reached by v_5, the ϕ-function at Y. Here, Y *dom* Z, and $Y \in shield(d, Z) \forall d \in \{Q, S, T, W\}$.

Considering the problem of merging information within a data-flow framework in general, it suffices to merge information at the iterated dominance frontier of the interesting nodes [3].

For reaching definitions, ϕ-functions are placed at $\mathrm{DF}^+(\mathcal{S})$, where $\mathcal{S}$ is the set of all definition sites for each variable [5]. A ϕ-function has an argument for each control flow predecessor, thus coalescing all definitions which reach that node into a single definition. In this way, linearity of reaching definitions is achieved. In the case alluded to in Section 1, with n definitions separated from n uses of a variable by a merge, a ϕ-function would be placed at the merge to collect all n definitions. Each of the n uses would then get their definition from the ϕ-function, resulting in only $2n$ definition chains, pruned from the original n^2.

4 Merging Information in a Precedence Graph

4.1 Interesting nodes and Merge Nodes within a PG

At what points in a PG do we need to merge information? We need to merge information at the precedence section nodes in which information first comes together. However, as opposed to sequential control flow, a variable may not be defined along every path which reaches a confluence point; as long as it is defined along *some* path which reaches a use for that variable, a definition for that variable will be available. This important distinction suggests that identifying merge points as the iterated dominance frontier of interesting nodes in the PG may not be correct. To see why, examine Figure 3(a), using our running example of reaching definitions. If v is only defined at node X, then any use of v at W, Z, or A will have that definition available, since X will always have been executed before any of these other sections execute. But the $\mathrm{DF}^+(X)$ $= \{W, Z, A\}$; clearly a merge node is not necessary when only a single definition of a variable reaches any point. Since a precedence graph guarantees execution of all predecessors, we need not be concerned about a definition of v flowing from node Y. Since Y does not define v, it does not contribute to the reaching definitions of v for the other nodes.

To see where merge operators are needed in a PG, first examine Figure 3(b), in which v is defined in sections X and Y, while used in section A. Since both definitions reach A without either killing the other, a merge operator is needed at A. However, we need merge only two definitions, even though there are three predecessors. Thus, a merge function for a PG only needs arguments for predecessors with definitions which reach the confluence node along that path. This highlights another major difference between sequential and parallel merges; therefore we will use a new operator, the ψ-function, as the merge operator for reaching definitions within the PG [13]. The ψ-function is similar to the ϕ-function in that it acts as a non-killing definition in terms of data-flow analysis, but it is also a use for all definitions which reach the ψ-function via its arguments. By collecting multiple reaching definitions the ψ-function linearizes definition chains within a PG in the same manner as the ϕ-function within a CFG.

To identify precisely where to place parallel merge operators in a PG we begin with an

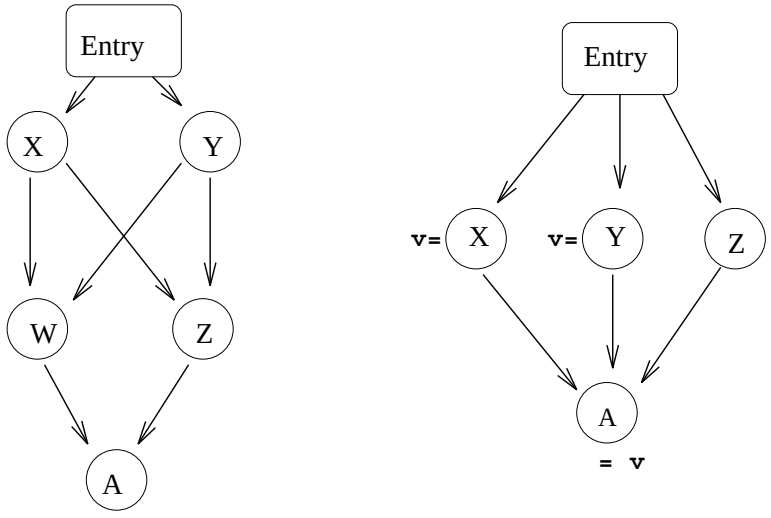


Figure 3

important definition:

Definition 3 *The* meet *of nodes X and Y, M(X, Y) =*

$$\left\{ Z \mid \forall Z_X, Z_Y \text{ with } Z_X \to Z \text{ and } Z_Y \to Z, \text{ and } \forall \text{ paths } p_X : X \xrightarrow{*} Z_X, p_Y : Y \xrightarrow{*} Z_Y, p_X \cap p_Y = \emptyset \right\}$$

We note that the *meet* of two nodes is the dual definition to *join*, as it uses a universal quantifier as opposed to the existential quantifier of *join*. For a set of nodes $\mathcal{S}$, $\mathrm{M}(\mathcal{S})$ is defined in the usual pairwise manner: $\mathrm{M}(\mathcal{S}) = \bigcup_{X,Y \in S} \mathrm{M}(X,Y)$. We also define $\mathrm{M}^+(\mathcal{S})$ as the limit of increasing sequences analogous to that used for join and dominance frontier:

$$\begin{aligned} \mathrm{M}^1(\mathcal{S}) &= \mathrm{M}(\mathcal{S}) \\ \mathrm{M}^2(\mathcal{S}) &= \mathrm{M}(\mathcal{S} \cup \mathrm{M}^1(\mathcal{S})) \\ \mathrm{M}^{i+1}(\mathcal{S}) &= \mathrm{M}(\mathcal{S} \cup \mathrm{M}^i(\mathcal{S})) \end{aligned}$$

The definition of Join (Definition 2, and the basis of work to place ϕ-functions [5]) is a well-known concept. Although in a CFG $\mathrm{J}^+(\mathcal{S}) = \mathrm{J}(\mathcal{S})$ [15], the dual definition of join for PGs, meet, does not possess this property. Consider Figure 3(a). Let $\mathcal{S}$= {*X, Y*}. Then $\mathrm{M}(\mathcal{S})$ = {*W,Z*}; in fact, $A \notin \mathrm{M}(\mathcal{S})$, but $A \in M(\mathcal{S} \cup M(\mathcal{S})) = \mathrm{M}(X, Y, W, Z) = \{W, Z, A\}$.

The meet of two nodes possesses one of the important properties which characterize nodes in a PG: it is unaffected by transitive edges. To prove this claim, we first formalize the concept of a transitive edge as follows:

Definition 4 *Edge E:* $X \to Y$ *added to graph G is a transitive edge if* $\exists Z \in G \ni X \xrightarrow{*} Z \xrightarrow{+} Y$.

We now show that the central concept of PGs, path reachability, is unaltered in the presence of transitive edges.

Theorem 1 *Path reachability in a PG is unaffected by transitive edges.*

Proof:
Consider PG G$'$, consisting of G plus transitive edge $E : X \rightarrow Y$. Since all edges in G exist in G$'$, if A reached B in G, A reaches B in G$'$. Now, let A reach B in G$'$, but assume that A does not reach B in G. Then path p_1: $A \stackrel{+}{\rightarrow} B$ in G$'$ must include E, else no distinction is possible between paths in G and G$'$. Thus, p_1 must be of the form $A \stackrel{*}{\rightarrow} X \rightarrow Y \stackrel{*}{\rightarrow} B$. By Definition 4, $X \stackrel{*}{\rightarrow} Z \stackrel{+}{\rightarrow} Y$ in G. Thus, path p_2: $A \stackrel{*}{\rightarrow} X \stackrel{*}{\rightarrow} Z \stackrel{+}{\rightarrow} Y \stackrel{*}{\rightarrow} B$ exists in G. By contradiction, we have demonstrated equivalence of path reachability between G and G$'$. ∎

We next demonstrate that the meet of a set of nodes is also unaffected in the presence of transitive edges.

Theorem 2 *The meet relation is insensitive to transitive edges.*

Proof:
We use G and G$'$ as defined in the proof of Theorem 1, except that E is any transitive edge added to G. We first show that for nodes X and Y, M(X,Y) in G is equal to M(X,Y) in G$'$ by means of double inclusion.

1. Let $Z \in \mathrm{M}(X,Y)$ in G. We show that $Z \in \mathrm{M}(X,Y)$ in G$'$. By Definition 3 for meet, the intersection of all pairs of paths in G from X and Y to predecessors of Z is empty. Now consider G$'$, which includes edge E: $A \rightarrow B$. Assume $Z \in \mathrm{M}(X,Y)$ in G, but $Z \notin \mathrm{M}(X,Y)$ in G$'$. Then, in G$'$ there exists node V such that $V \stackrel{+}{\rightarrow} Z$ with $X \stackrel{*}{\rightarrow} V$ and $Y \stackrel{*}{\rightarrow} V$. If no path from $X \stackrel{*}{\rightarrow} V$ or $Y \stackrel{*}{\rightarrow} V$ passes through A, then V does not exist, since the only difference between G and G$'$ is edge E. Thus, without loss of generality, at least X, and perhaps Y, has a path to V which passes through A. But from A, no nodes are reachable in G$'$ that were not reachable in G, as Theorem 1 demonstrated. Hence, if V exists in G$'$ it exists in G, since its existence is predicated upon reachability. We conclude that since V does not exist in G it cannot exist in G$'$. By contradiction, $Z \in \mathrm{M}(X,Y)$ in G$'$.

2. Let $Z \in \mathrm{M}(X,Y)$ in G$'$. By Definition 3 the intersection of all pairs of paths from X and Y to predecessors of Z is empty in G$'$. Since the edges in G are a subset of the edges in G$'$, any pair of paths from X and Y to predecessors of Z which exists in G exists in G$'$, and is empty in G$'$ by assumption. Thus, that pair of paths is empty in G, and $Z \in \mathrm{M}(X,Y)$ in G.

Now consider the M($\mathcal{S}$), where $\mathcal{S}$ is a set of nodes. Since $\mathrm{M}(\mathcal{S}) = \bigcup_{X,Y \in \mathcal{S}} \mathrm{M}(X,Y)$, we apply the property just proved to each pair X, Y to obtain the desired result for S: M($\mathcal{S}$) in G equals M($\mathcal{S}$) in G$'$. ∎

Given a set of interesting nodes $\mathcal{S}$, merge operators for PGs need to be placed at the iterated meet of $\mathcal{S}$. In terms of reaching definitions, given a variable v and a set $\mathcal{S}$, where $\mathcal{S}$ is the

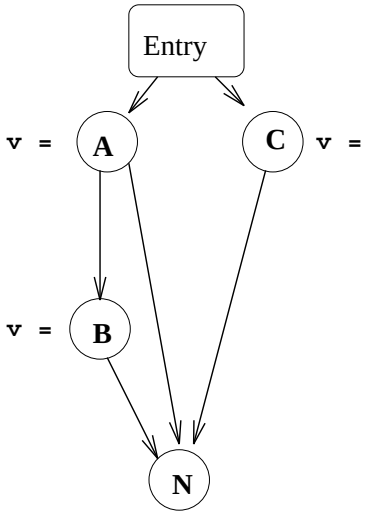


Figure 4

set of section nodes in a PG which define v, ψ-functions need to be placed at $\mathrm{M}^+(\mathcal{S})$, where a ψ-function for v at section node A collects all definitions of v that reach A. That is, there is an argument of the ψ-function for each predecessor of A that has a definition of v reaching A. Figure 4 shows the case where even though an edge exists from a definition of v (in node A) to the confluence node N, the ψ-function placed at N will only collect the definitions from nodes B and C. That is because the definition at A gets killed by the definition at B in this PG. In this case, $\mathcal{S}$= $\{A, B, C\}$, and $\mathrm{M}(\mathcal{S}) = \mathrm{M}+(\mathcal{S}) = \{N\}$, but we note that the edge $A \rightarrow N$ is a transitive edge and $\mathrm{M}(B, C) = \{N\}$.

We will first prove that it is sufficient to place ψ-functions at $\mathrm{M}^+(\mathcal{S})$. Since $\mathcal{S}$represents nodes with non-identity transfer functions, this result holds for data-flow problems in general by extending Definition 1 for reaching definitions to kill information for other data-flow problems. The concept of iterated meet is a refinement of the ψ-function placement method suggested earlier [13], in that the iterated meet is smaller and, in fact, the minimal set.

How do ψ-functions affect reaching definitions in a PG? If node N is reached by ψ-function s and s is reached by definition d (where s collects d as a ψ-argument), then d reaches N indirectly via a ψ-function. In general, it may be that one or more ψ-functions lie on the path from d to N. In that case, d reaches N indirectly via a *ψ-chain*. Thus, a definition or ψ-function in a PG which reaches node N in the sense of Definition 1 is called a *direct* reaching definition, whereas a definition which reaches node N via a *ψ-chain* is called an *indirect* reaching definition.

We now prove the following important results: that placing ψ-functions at the iterated meet of the set of nodes which define a variable maintains the properties of (1) unique reaching definitions, (2) collects all definitions (directly or indirectly) which before ψ-function placement could reach each node, and (3) is the minimal set at which to place such merge operators.

Theorem 3 *In a PG, with ψ-functions for v placed at $M^+(\mathcal{S})$, all uses of v within node N will be reached (in the sense of Definition 1) by exactly one definition (including ψ-functions) of v.*

Proof:
Let G be a PG before placing ψ-functions, and G_ψ be the same graph after ψ-function placement.

Within G, let the set of nodes with definitions of v be $\mathcal{S}$, $\mathcal{S}' \subseteq \mathcal{S}$ be the set of nodes in $\mathcal{S}$ which have definitions which reach N, and $\mathcal{T} \subseteq \mathcal{S}$ be the set of nodes in $\mathcal{S}$ with paths that reach N.

EXISTENCE. We first show that any use which had at least one reaching definition in G has at least one reaching definition in G_ψ. Since $\mathcal{S}' \neq \emptyset$, let $A \in \mathcal{S}'$ in G. Then all paths $p_A : A \xrightarrow{*} Z$, with $Z \rightarrow N$, contain no definitions of v (except at A). For all p_A in G_ψ, if the definition of v in A does not directly reach N, then there must be at least one ψ-function along some p_A. In this case, at least one ψ-function reaches N.

UNIQUENESS. We consider cases:
(*i*) Only one $W \in \mathcal{S}'$ reaches N ($\mid \mathcal{S}' \mid = 1$). In this case, the definition in W kills any other definitions which may exist in nodes of $\mathcal{T}$. Then, $\forall t1, t2 \in \mathcal{T}$, $\exists p_{t1}$:$t1 \xrightarrow{+} N$, p_{t2}: $t2 \xrightarrow{+} N$, $W \in p_{t1} \cap p_{t2}$ Thus, $N \notin \mathrm{M}(\mathcal{T})$, and more generally, no node on any p_W:$W \xrightarrow{+} N \in \mathrm{M}(\mathcal{T})$ (except, perhaps, W). Repeating this argument, no node in any p_W - $\{W\} \in \mathrm{M}^+(\mathcal{T})$. Thus, in G_ψ only the definition of v in W reaches N, since no additional definitions (ψ-functions) created in G_ψ can directly reach N.
(*ii*) Multiple definitions of v reach N from $\mathcal{S}'$, with $N \in \mathrm{M}^+(\mathcal{S})$. Then a ψ-function will be placed at the beginning of node N in G_ψ, and uses of v within N will be reached by that ψ-function.
(*iii*) Multiple definitions of v reach N from $\mathcal{S}'$, with $N \notin \mathrm{M}^+(\mathcal{S})$. Assume N is reached by more than one definition from members of $\{\mathcal{S} \cup \mathrm{M}^+(\mathcal{S})\}$. Call this set R_1. Then either (a) $N \in \mathrm{M}(\mathrm{R}_1)$, which contradicts our assumption, or (b) $\forall A, B \in \mathrm{R}_1$ $\mathrm{M}(A, B)$ is non-empty (since A and B reach N), and we call the set of all elements of $\mathrm{M}(\mathrm{R}_1)$ which have paths that reach N R_2. Repeating this process, we note that R must converge at R_+ since R_{n+1} is always composed of nodes closer to N, along the paths from nodes in R_1 to N, than the nodes in R_n. If the set R_+ consists of exactly one node (it can't be zero by the existence proof) we have a contradiction of assumption, and if it contains more than one node (which can't include N by assumption) R has not converged. A contradiction is again reached as long as G contains a finite number of nodes.

Thus, in all cases, we have shown that in G_ψ precisely one reaching definition will exist for each use which had at least one reaching definition in G. ∎

Theorem 4 *Within a PG, with ψ-functions placed at $M^+(\mathcal{S})$, any use of v at node N will be reached directly or indirectly by all definitions of v which reached N before placing ψ-functions.*

Proof:
Let G be a PG before placing ψ-functions, and G_ψ be the same graph after ψ-function placement. We consider two cases:
(*i*) Only one definition of v reaches N in G. This case is handled similarly to case (*i*) in Theorem 3, and the single definition which reached N in G will reach N in G_ψ.
(*ii*) Multiple definitions of v reach N in G. All definitions for v from node A which reach N in G reach N indirectly in G_ψ via a ψ-chain. To show this, consider all paths p from A to N in G. By Definition 1 no paths from A to N in G contain definitions of v. Since, by assumption, a definition of v in G_ψ does not reach N from A, by Definition 1 there must exist a definition along some path from A to N which did not exist in G. That definition can only be a ψ-function. If there is just one ψ-function along the path then it collects all definitions which reach it, and that ψ-function will now reach N, resulting in the definition of v reaching N indirectly. If there is more than one ψ-function along any path from A to N, the argument is repeated. By induction, a definition of v in A will reach N via a ψ-chain. Thus, we have shown that reachability of all definitions is maintained when placing ψ-functions. ∎

We now show that the $\mathrm{M}^+(\mathcal{S})$ is the minimal set at which to place merge operators, which will hold in the specific case of ψ-functions.

Theorem 5 *Within a PG, for a set of nodes $\mathcal{S}$ which define v, $M^+(\mathcal{S})$ is the smallest set at which to place ψ-functions in order to insure unique reaching definitions at all nodes.*

Proof:
Given $\mathcal{S}$ for variable v, consider any element $N \in \mathrm{M}^+(\mathcal{S})$. Let $N \in \mathrm{M}^j(\mathcal{S})$, for the minimum $j \geq 1$. Then $\exists X, Y \in \mathrm{M}^{j-1}(\mathcal{S})$ (where $\mathrm{M}^0(\mathcal{S}) = \mathcal{S}$) such that all pairs of paths from X and Y to predecessors of N are empty. Since X and Y contain definitions of v (either assignments to v or ψ-functions for v), the definitions at X and Y (or, perhaps, a later definition of v within some node along one of these disjoint paths) both reach N. Thus, by removing the ψ-function for v at N, any use of v within N would be reached by multiple definitions. ∎

4.2 Depth-first Renaming

Computing the iterated meet seems somewhat impractical from its definition. Again, referring to the reaching definitions problem within a CFG, after placing ϕ-functions a technique known as *renaming* transforms each variable definition into a unique name and each use into the name of its unique reaching definition [5]. The method employed to perform this renaming is depth-first, in that it recursively traverses the dominator tree in a depth-first order, keeping a stack of current definitions for each variable. The key property that this renaming scheme satisfies is that at each node the correct "current" definition (an original definition or ϕ-function) of each variable is the most recent definition on the depth-first path to this node from *Entry*, i.e., the definition on top of the definition stack [5, Lemma 10]. In fact, a depth-first traversal of any spanning tree of the CFG will also satisfy this property. Unfortunately, a depth-first traversal of the nodes of a PG will not satisfy this key property with merge operators at $\mathrm{M}^+(\mathcal{S})$. For

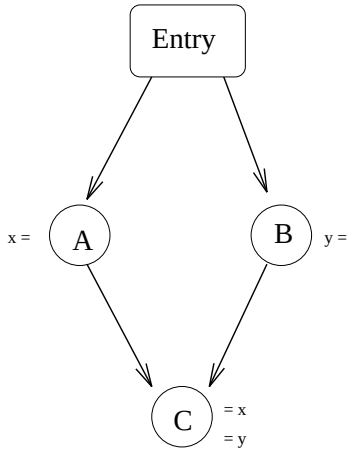


Figure 5

instance, in Figure 5, no ψ-function is needed at node C for either x or y, since only one definition of each variable reaches node C (in the sense of Definition 1). Suppose the depth-first traversal of the PG visits node C after node A; when visiting node C, the current definition of variable x will be the definition in A, but the current definition of variable y will be wrong.

In order to use a depth-first renaming algorithm, we introduce additional ψ-functions as placeholders [13, 14]. The renaming algorithm described in this earlier work visits the PG nodes in a depth-first traversal of a spanning tree that satisfies topological ordering, and identifies and removes these spurious ψ-functions. In fact, we place ψ-functions at the iterated dominance frontier; the next section proves that this is a safe approximation to the iterated meet.

4.3 Efficient Implementation

What method can be used which is relatively efficient and yet correctly propagates information between section nodes of a PG? We need to look more closely at how information flows between nodes in a PG, keeping in mind that a precedence graph has different semantics compared to a CFG.

Since information flowing through the PG is described in terms of reachability, we have found the concept of *reaching frontier* useful. This concept describes reachable nodes in a PG in a way that is analogous to the dominance frontier for nodes within a CFG.

Definition 5 *The* reaching frontier *of X, RF(X) =*

$$\{Z \mid \textit{ X reaches a predecessor of Z, but X does not reach all predecessors of Z}\}$$

The reaching frontier of a set $\mathcal{S}$, $\mathrm{RF}(\mathcal{S})$, is defined to be the union of the reaching frontiers of all elements of $\mathcal{S}$, i.e., $\mathrm{RF}(\mathcal{S}) = \bigcup_{X \in \mathcal{S}} RF(X)$. The *iterated reaching frontier*, $\mathrm{RF}^+(\mathcal{S})$, is defined similarly to that for join, meet, and dominance frontier. The reaching frontier is used to relate important properties between the meet and dominance frontier. To implement the placement of operators which merge information within a PG, we would like to show that $\mathrm{M}^+(\mathcal{S}) \subseteq \mathrm{RF}^+(\mathcal{S}) \subseteq \mathrm{DF}^+(\mathcal{S})$.

How are the meet and reaching frontier related? The analogous relations in sequential CFGs, join and dominance frontier, are shown to be equal when iterated, with the provision that *Entry* $\in \mathcal{S}$. However, *Entry* adds no information to either the meet or the reaching frontier in a PG. $\mathrm{M}(Entry, X) = \emptyset \; \forall X$ since *Entry* reaches all nodes, and thus there is always a path from *Entry* to any node on any path from X. Also, $\mathrm{RF}(Entry) = \emptyset$, since *Entry* reaches all predecessors of all nodes.

We can also show that $\mathrm{RF}^+(\mathcal{S}) \neq \mathrm{M}^+(\mathcal{S})$. Simply choose the set $\mathrm{T} = \{X, Entry\}$. Then $\mathrm{M}(\mathrm{T}) = \emptyset$, so $\mathrm{M}^+(\mathrm{T}) = \emptyset$, while $\mathrm{RF}(\mathrm{T})$ clearly may not be empty. We now show that in general $\mathrm{M}^+(\mathcal{S}) \subseteq \mathrm{RF}^+(\mathcal{S})$.

Theorem 6 *$M^+(\mathcal{S}) \subseteq RF^+(\mathcal{S})$*

Proof: Let $Z \in \mathrm{M}(\mathcal{S})$. Then there is a node $X \in \mathcal{S}$ such that X has a path that reaches a predecessor of Z, but X cannot reach all predecessors of Z or else there would be no path from any other node that did not intersect some path from X to each predecessor of Z (which would imply that $Z \notin \mathrm{M}(\mathcal{S})$). So, we have $Z \in \mathrm{RF}(X)$ and $Z \in \mathrm{RF}(\mathcal{S})$.
Finally, $\mathrm{M}(X) \subseteq \mathrm{RF}(X) \Longrightarrow \mathrm{M}^+(X) \subseteq \mathrm{RF}^+(X)$. ∎

We also show that $\mathrm{DF}(\mathcal{S})$ is not in general a subset or superset of $\mathrm{RF}(\mathcal{S})$. In Figure 6, $\mathrm{DF}(X) = \{A, Z\}$, but $\mathrm{RF}(X) = \{Z\}$, since it reaches all predecessors of A. It's also easy to find

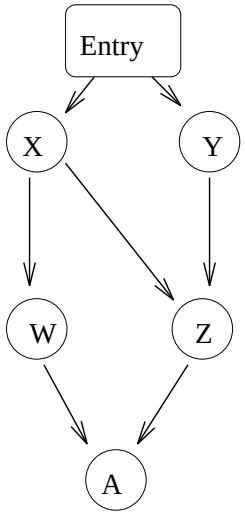


Figure 6

a graph where X reaches a predecessor of Z but does not dominate any predecessor of Z, so $Z \in \mathrm{RF}(X)$, but $Z \notin \mathrm{DF}(X)$.

Next, we show that the iterated dominance frontier is a superset of the iterated reaching frontier on all graphs.

Theorem 7 *$DF^+(\mathcal{S}) \supseteq RF^+(\mathcal{S})$*

Proof:

It has been shown [5, Lemma 4] that for any node Z that X reaches, some node $Y \in \{X \cup \mathrm{DF}^+(X)\}$ dominates Z. Now, for any node Z that X reaches, if Z is in $\mathrm{RF}(X)$, then Z is in $\mathrm{DF}^+(X)$; this is because some node in $\mathrm{DF}^+(X)$ must dominate Z. Choose a path p from X to Z. Let Y be the last node on p in $\{X \cup \mathrm{DF}^+(X)\}$; Y must dominate Z. If Y is not Z, then Y dominates all predecessors of Z, so there is a path from Y to all predecessors of Z; thus there is a path from X to all predecessors of Z, and Z is not in $\mathrm{RF}(X)$.

Thus, $\mathrm{DF}^+(X) \supseteq \mathrm{RF}(X)$. Thus, $\mathrm{DF}^+(\mathcal{S}) \supseteq \mathrm{RF}(\mathcal{S})$. $\mathrm{RF}^2(\mathcal{S}) = \mathrm{RF}(\mathcal{S} \cup \mathrm{RF}(\mathcal{S})) \subseteq \mathrm{RF}(\mathcal{S} \cup \mathrm{DF}^+(\mathcal{S})) \subseteq \mathrm{DF}^+(\mathcal{S} \cup \mathrm{DF}^+(\mathcal{S})) = \mathrm{DF}^+(\mathcal{S})$. By induction, $\mathrm{DF}^+(\mathcal{S}) \supseteq \mathrm{RF}^+(\mathcal{S})$. ∎

In general $\mathrm{RF}^+(X) \neq \mathrm{DF}^+(X)$. Although $\mathrm{DF}^+(X) \supseteq \mathrm{RF}^+(X)$, the converse is not necessarily true. Consider Figure 7. $\mathrm{DF}(X) = \{B,Z\}$, and $\mathrm{DF}^2(X) = \mathrm{DF}^+(X) = \{B,Z,X\}$. However, $\mathrm{RF}(X) = \{B\}$, and $\mathrm{RF}^2(X) = \mathrm{RF}^+(X) = \{B,X\}$. Thus, by counterexample, $\mathrm{RF}^+(X) \not\supseteq \mathrm{DF}^+(X)$.

But, we note that the above example contains a cycle. We are interested in placing ψ-functions in a PG, which we know to be acyclic. And, in a DAG, we next show that $\mathrm{RF}^+(\mathcal{S}) = \mathrm{DF}^+(\mathcal{S})$.

Theorem 8 *In a DAG, $RF^+(\mathcal{S}) = DF^+(\mathcal{S})$*

Proof: Given a DAG, we demonstrate two preliminary lemmas.

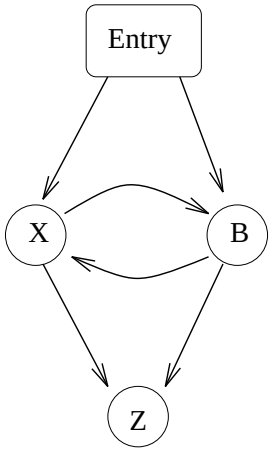


Figure 7

- **Lemma 1** *$RF^+(\mathcal{S}) \supseteq DF(\mathcal{S})$.*

  Let $X \in \mathcal{S}$ and let Z be in DF(X). Then X dom A, a predecessor of Z. X $\overline{\text{dom}}$ B, some other predecessor of Z, since X $\overline{\text{dom}}$ Z. If X does not reach B, then Z is in RF(X). So assume that X reaches B.

  We now show that on some path from X to B, there exists a C such that C is in RF(X). Since X $\overline{\text{dom}}$ B, consider a path from entry to B such that X is not on the path (there must be at least one such path). Let C be the first node on this path that X can reach (C may be B). Then since X can reach C, but not the predecessor of C on this path, C is in RF(X).

  Next, note that C cannot reach A. Else, we would have the path Entry $\rightarrow C \rightarrow A$ (which cannot go through X since the graph is acyclic) which does not pass through X, contradicting the fact that X dom A.

  But, this means that Z is in RF(C), since C reaches Z through B, but cannot reach A. We already know that C is in RF(X), so we have shown that Z is in $\text{RF}^+(X)$. ∎

- **Lemma 2** *$RF^+(\mathcal{S}) \supseteq DF^+(\mathcal{S})$.*

  Given Lemma 1, we know that $\text{RF}^+(X) \supseteq \text{DF}(X)$. So, $\text{RF}^+(\mathcal{S}) \supseteq \text{DF}(\mathcal{S})$. $\text{DF}^2(\mathcal{S}) = \text{DF}(\mathcal{S} \cup \text{DF}(\mathcal{S})) \subseteq \text{RF}^+(\mathcal{S} \cup \text{RF}^+(\mathcal{S})\ ) = \text{RF}^+(\mathcal{S})$. By induction, $\text{DF}^+(\mathcal{S}) \subseteq \text{RF}^+(\mathcal{S})$. ∎

Lemma 2 together with Theorem 7 gives us our result. ∎

Since $\text{M}^+(\mathcal{S}) \subseteq \text{RF}^+(\mathcal{S}) \subseteq \text{DF}^+(\mathcal{S})$ (with $\text{RF}^+(\mathcal{S}) = \text{DF}^+(\mathcal{S})$ in a DAG), we have shown that merging information within a PG at the $\text{DF}^+(\mathcal{S})$ is a safe approximation for the somewhat smaller set of $\text{M}^+(\mathcal{S})$. However, for the common depth-first implementations which use renaming, such as ψ-functions for reaching definitions within PGs, placing merge operators at DF^+(S) may well be necessary for correct propagation of information, as discussed above. Another paper [14] has shown that placing ψ-functions at these points does correctly propagate reaching definitions, albeit through placeholder ψ-functions at times.

How conservative is the use of $\mathrm{DF}^+(\mathcal{S})$ as an approximation for $\mathrm{M}^+(\mathcal{S})$? First, if there is only one member of $\mathcal{S}$, then $\mathrm{M}^+(\mathcal{S})$ will be empty, while $\mathrm{DF}^+(\mathcal{S})$ will usually not be empty. Second, $\mathrm{DF}^+(\mathcal{S})$ assumes a definition lies along all possible paths. Thus, in the case of Figure 8 where $\mathcal{S}= \{A, C\}$, $\mathrm{M}(\mathcal{S}) = \mathrm{M}^+(\mathcal{S}) = \{E\}$, while $\mathrm{DF}(\mathcal{S}) =$ includes D. Third, $\mathrm{M}^+(\mathcal{S})$ is insensitive to transitive edges, while $\mathrm{DF}^+(\mathcal{S})$ is not. Again, examine Figure 8, where $\mathrm{DF}^+(\mathcal{S}) = \{D, E, F\}$. A ψ-function is only needed at E, but the insensitivity to transitive edges of $\mathrm{DF}^+(\mathcal{S})$ adds node F to its set.

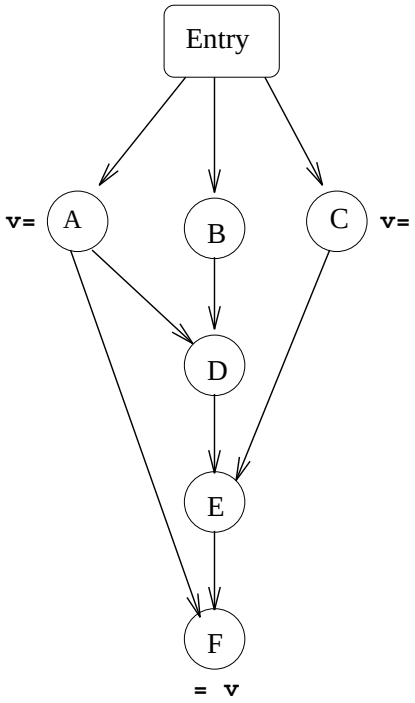


Figure 8

However, extra ψ-functions are safe, since they only pass along the information collected at those points. Thus, merging information at the DF^+ of the set of interesting nodes within a PG has been shown to be a safe method, and is relatively efficient since it can be performed with the same complexity as that for ϕ-function placement and the techniques employed for sparse evaluation graphs [3].

5 Conclusions and Extensions

In this paper we have contrasted the order of execution within a DAG parallel environment which uses precedence graphs to that of control flow graphs for sequential programs. Further, we have discovered the minimal set of points in precedence graphs to merge information. However, this set is currently infeasible to implement since depth-first renaming schemes, such as those used for implementing SSA form, require a larger set of confluence nodes, notably the iterated dominance frontier of a set of nodes. We have also shown that using the iterated dominance frontier is a safe and reliable alternative.

Future work includes looking more closely at the ideas of shielding as it relates to sequential control flow graphs, and studying other problems which must merge the flow of information within parallel precedence graphs.

Finally, we mention the open question: are there other renaming schemes, not depth-first based, which could efficiently make use of the iterated meet, if it could be effectively calculated?

References

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

[2] Per Brinch Hansen. *Operating Systems Principles*. Automatic Computation. Prentice-Hall, 1973.

[3] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Conf. Record 18th Annual ACM Symp. Principles of Programming Languages*, pages 55–66, Orlando, Florida, January 1991.

[4] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. On the efficient engineering of ambitious program analysis. *IEEE Transactions on Software Engineering*, 20(2):105–114, February 1994.

[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. on Programming Languages and Systems*, 13(4):451–490, October 1991.

[6] Ron Cytron, Michael Hind, and Wilson Hsieh. Automatic generation of DAG parallelism. In *Proc. ACM SIGPLAN '89 Conf. on Programming Language Design and Implementation*, pages 54–68, Portland, OR, June 1989.

[7] Fox *et al.* Common runtime support for high-performance parallel languages. In *Proceedings of Supercomputing 93*, pages 752–757, Portland, OR, November 1994.

[8] Dirk Grunwald and Harini Srinivasan. Data flow equations for explicitly parallel programs. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 159–168, San Diego, California, May 1993. ACM Press.

[9] Matthew S. Hecht. *Flow Analysis of Computer Programs*. North Holland, New York, 1977.

[10] Steven S. Muchnick and Neil D. Jones, editors. *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.

[11] Parallel Computing Forum. PCF Parallel Fortran extensions. *Fortran Forum*, 10(3), September 1991. (special issue).

[12] Vivek Sarkar and Barbara Simons. Parallel program graphs and their classification. In Utpal Banerjee, David Gelernter, Alexandru Nicolau, and David A. Padua, editors, *Languages and Compilers for Parallel Computing*, number 768 in Lecture Notes in Computer Science, pages 633 – 655. Springer-Verlag, 1993.

[13] Harini Srinivasan, James Hook, and Michael Wolfe. Static single assignment for explicitly parallel programs. In *Conf. Record 20th Annual ACM Symp. Principles of Programming Languages*, pages 16–28, Charleston, SC, January 1993.

[14] Eric Stoltz, Harini Srinivasan, James Hook, and Michael Wolfe. Static Single Assignment form for explicitly parallel programs: Theory and practice. *submitted for publication*, August 1994.

[15] Michael Wolfe. $J^+ = J$. *ACM Sigplan Notices*, 29(7):51–53, July 1994.