Efficient Accommodation of May-Alias Information in SSA Form

Ron Cytron*
Washington University
St. Louis, Missouri 63130

Reid Gershbein†
Oberlin College
Oberlin, Ohio 44074

Abstract

We present an algorithm for incrementally including may-alias information into Static Single Assignment (SSA) form by computing a sequence of increasingly precise (and correspondingly larger) partial SSA forms. Our experiments show significant speedup of our method over exhaustive use of may-alias information, as optimizattion problems converge well before most may-aliases are needed.

1 Introduction

Consider the source program and its SSA representation [CFR+91] shown in Figure 1. The storage referenced by a program P is logically partitioned into disjoint *storage names*, which correspond approximately to the variable names in P. The "store" and "load" instructions of P are statically labelled as distinct *def* and *use* sites, respectively. For example, step $\boxed{2}$ contributes one use (of the pointer p) and one def (of location $\star p$).

In SSA form, the storage names for P are augmented so that the same storage name never appears twice among the def sites. For example, the defs of u at step $\boxed{1}$ and $\boxed{3}$ are renamed to u_1 at step $\boxed{5}$ and u_2 at $\boxed{8}$, which are regarded as *distinct* storage names. A ϕ-function represents the merge of distinct names, and the value returned by the ϕ-function is itself retained as a distinct name. For example, step $\boxed{9}$ combines the aforementioned renames of u. As a result, each use can be associated (renamed to correspond) with a unique def. The u of step $\boxed{4}$ is renamed to u_3 at $\boxed{11}$.

Moreover, *any* potential modification of a storage name v is represented explicitly in SSA form. Where uncertainty exists concerning a dereferenced pointer p's effect on v at step S, we write

$$\star p \overset{S}{\sim} v$$

and say that $\star p$ *may alias* v at S. Generally, we define the set $MayAlias(p, S)$ to contain those names potentially aliased with $\star p$ at S. For example, if the pointer p contains the address of v at step $\boxed{2}$, then the value of v is changed as of that step; otherwise the value of v is unchanged. The $IsAlias()$ function is inserted into a program P to represent this duality. The predicate at step $\boxed{12}$ determines if the alias

Source Program	Full SSA Representation
$v \leftarrow 3$	$v_0 \leftarrow 3$
if (b) **then**	**if** (b) **then**
$u \leftarrow v \quad \Leftarrow \boxed{1}$	$u_1 \leftarrow v_0 \quad \Leftarrow \boxed{5}$
else	**else**
$\star p \leftarrow 6 \quad \Leftarrow \boxed{2}$	$\star p \leftarrow 6 \quad \Leftarrow \boxed{6}$
	$v_1 \leftarrow IsAlias(p, \&v_0) \quad \Leftarrow \boxed{7}$
$u \leftarrow v \quad \Leftarrow \boxed{3}$	$u_2 \leftarrow v_1 \quad \Leftarrow \boxed{8}$
fi	**fi**
	$u_3 \leftarrow \phi(u_1, u_2) \quad \Leftarrow \boxed{9}$
	$v_2 \leftarrow \phi(v_0, v_1) \quad \Leftarrow \boxed{10}$
$c \leftarrow v + u \quad \Leftarrow \boxed{4}$	$c \leftarrow v_2 + u_3 \quad \Leftarrow \boxed{11}$
	Function $IsAlias(w, x)$ **:** *value*
	if ($w = x$) **then** $\quad \Leftarrow \boxed{12}$
	$ans_1 \leftarrow \star w \quad \Leftarrow \boxed{13}$
	else
	$ans_2 \leftarrow \star x \quad \Leftarrow \boxed{14}$
	fi
	$ans_3 \leftarrow \phi(ans_1, ans_2)$
	return (ans_3)
	end

Figure 1. Example.

holds, and steps $\boxed{13}$ and $\boxed{14}$ cause the appropriate value to be returned.

If storing through a pointer certainly modifies v at S, then we write

$$\star p \overset{S}{\approx} v$$

and say that $\star p$ *must alias* v at S. Correspondingly, we define the set $MustAlias(p, S)$ to contain those names certainly aliased with $\star p$ at S. If $v \in MustAlias(p, \boxed{6})$, then step $\boxed{7}$ should be replaced by the assignment "$v \leftarrow \star p$". If v is unaffected by step $\boxed{6}$, then no assignment to v would be necessary at step $\boxed{7}$.

Transformation of source programs into Static Single Assignment (SSA) form [CFR+91] results in more efficient and effective program optimization [WZ91, AWZ88]. Without SSA form, each program optimizaion would require special-case treatment of constructs that yield implicit storage references. The results of applying constant propagation based on SSA form [WZ91] to our example are dependent on the program's SSA form, which is in turn dependent on the alias information associated with storage assignments. With $\perp$ representing "non-constant", the following table illustrates the effects of alias information at step $\boxed{6}$ on subsequent uses of v:

Alias Info at $\boxed{6}$	Use of v at $\boxed{8}$	Use of v at $\boxed{11}$
$\star p \sim v$	$\perp$	$\perp$
$\star p \approx v$	6	$\perp$
Otherwise	3	3

---

*Currently on leave from IBM T.J. Watson Research Labs; e-mail: cytron@cs.wustl.edu.

†This work was supported by the National Science Foundation, Grant CDA-9123643; e-mail: gershbei@cs.oberlin.edu

While SSA offers a unified treatment of all storage references, the introduction of $IsAlias()$ assignments can expand the program representation size by a factor of $|V|$, where worst-case may-aliasing conditions prevail on a set of variables V. The use of languages whose alias behavior is sufficiently restricted [Bod90] can lessen the impact of explicit alias representation, as can recent advances in obtaining high-quality may-alias information [Lan92, LR92, CBC93]. However, the number of aliased expressions per node can grow quite large: the `make.c` program serves as a worst-case in [Lan92], with 675 may-alias relations per node on average, and one node containing some 2000 relations. Research is underway to determine how relations translate into effects on storage names [Lan93], but currently the impact of so many alias relations on SSA form cannot be quantified.

Our experiments show that the amount of alias information represented in SSA form can far exceed the amount of such information *necessary* for solving a given optimization problem: our algorithm typically stops well short of computing full SSA form, without compromising the effectiveness of program optimization while dramatically improving its efficiency. An earlier method for accommodating alias information in SSA form [CWZ90] gains efficiency by directly propagating information to affected program sites. Our approach is complementary, since we avoid computing the bulk of may-alias information; an actual optimization problem motivates and terminates the search for alias information.

In the following sections we outline our approach (Section 2), present our algorithm (Section 3) and correctness proofs (Section 4), and describe the results of our experiments (Section 5).

2 Approach

Essentially, our method constructs a sequence of approximating, *partial* SSA forms of increasingly sharper precision and correspondingly larger size:

$$SSA_0, SSA_1, \ldots, SSA_\Omega, SSA_{\Omega+1}, \ldots SSA_\infty$$

The limit of this sequence, SSA_∞, is full SSA form. Suppose an optimization based on SSA form were applied to each element of this sequence. We define the corresponding solution sequence as

$$Soln_0, Soln_1, \ldots, Soln_\Omega, Soln_{\Omega+1}, \ldots Soln_\infty$$

Since the explicit representation of may-alias information is primarily responsible for size excesses in SSA form, we carefully titrate may-alias information into our sequence of SSA forms. However, must-alias information is introduced initially (into SSA_0) for the following reasons:

- Although they supply very strong information, must-aliases of pointer dereferences are all too rare in programs. We expect relatively little growth in program size for representing such information.

- As discussed in Section 3.1, incremental update of the optimization solution between partial SSA forms is easily accomplished if the optimization solution in SSA_{i+1} cannot be "better" (in the sense of the data flow lattice) than the solution for SSA_i. This requirement precludes arbitrary injection of must-alias information into our sequence.

Partial form SSA_0 therefore contains all must-alias information, but no information due to *may* aliases, while SSA_∞ contains all such information. We define SSA_Ω as the earliest element of the partial SSA form sequence whose precision suffices for solving the optimization problem:

$$\Omega = \min_i \left(Soln_i \equiv Soln_\infty \right)$$

We say that, with respect to a given data flow solution obtained using SSA form, *convergence* is reached at approximation SSA_Ω. A diagram of our approach is shown in Figure 2. Because we expect SSA_Ω to be much smaller than

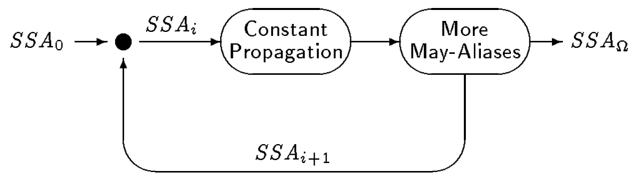

Figure 2. Diagram of our approach.

SSA_∞, we expect a corresponding reduction in the time taken for program optimization. The above diagram raises the following questions, which we answer in the ensuing sections of this paper:

1. Must the solution for constant propagation be computed "from scratch", as we move from SSA_i to SSA_{i+1}?

2. Must SSA_{i+1} be generated "from scratch"?

3. When is SSA_Ω achieved?

4. Which may-alias information should be included in moving from SSA_i to SSA_{i+1}?

5. How efficient is our incremental approach compared to solving the optimization problem using SSA_∞?

We assume that a program P is represented by the following structures:

Flow graph: The nodes of this directed graph represent executable text and the edges represent potential transfer of control between the nodes. Operations within a node are assumed to be totally ordered, particularly those that reference storage.

Storage names: Storage for P is partitioned into a set of static *storage names*. Where the structure of dynamically allocated objects cannot be analyzed at greater precision [LH88, CWZ90, HN90], a single name (such as "heap") can represent all such objects.

References: Each reference (*ref*) r represents a store (*def*) or load (*use*) instruction of P. To simplify the discussion, we assume that all references to storage are through a primary or dereferenced identifier (such as v or $\star p$); temporary names can be inserted into the source program to achieve this property.

We extend our earlier notation for aliases by allowing $MayAlias(r)$ and $MustAlias(r)$ to represent the sets of symbols may- and must-aliased for reference r, respectively. The following notation is also useful:

$DefSites$ is the set of all def sites in P.

Symbol(r) represents the textual "string" associated with reference r. As discussed above, such strings are limited to primary and dereferenced identifiers.

Rdef(u) is the unique def of *Symbol*(u) for use u in SSA form, where *Symbol*(u) is a primary (non-dereferenced) identifier; for a use u of a derferenced identifier, $Symbol(u) = \bot$. In moving from one partial SSA form to another, $Rdef(u)$ may change for any given use u.

Node(r) is the flow graph node containing r.

FirstDef(X) is the first def site in node X. Our algorithm assumes that each node begins with a def site; if node X otherwise lacks an initial def site, then an "empty" def site d ($Symbol(d) = \bot$) can be inserted at the beginning of node X with empty may- and must-alias sets.

LastDef(X) is the last def site in node X.

DomDef(r) is the def site that immediately dominates reference r. Note that r may be a def site or use site, and *Symbol*(r) may be a different symbol than $Symbol(DomDef(r))$.
Within a node, $DomDef(r)$ is simply the def site that occurs just prior to reference r among the references in $Node(r)$. Where none exists within $Node(r)$, then node $idom(Node(r))$ is consulted, where $idom(X)$ is the node immediately dominating X in the flow graph [LT79]. Since each node is assumed to have at least one def site, $DomDef(r)$ is well-defined.

Aliases: As discussed above, two lists of storage names are associated with each reference r, representing *MayAlias*(r) and *MustAlias*(r). Because we expect *MustAlias*(r) to be a rather small set, these symbols can be elaborated directly at the reference without compromising the asymptotic behavior of our approach. On the other hand, worst-case assumptions for *MayAlias*(r) result in relatively large ($O(|V|)$) sets, which if stored directly at each reference, would cause the program representation to suffer the $O(|V|)$ growth that we seek to avoid. Since we don't expect the *MayAlias*() sets to change drastically from reference to reference, we construct a global set of *may-alias relations*, where each relation is a subset of the program's storage names. We then encode *MayAlias*(r) as the the name (i.e., index) of the appropriate subset of storage names.

ϕ-functions: SSA form introduces a ϕ-function for variable v at node Z if multiple definitions of v reach Z. If Z has k predecessors, $Y_1, Y_2, \ldots, Y_k$, then the ϕ-function is of the form

$$v_Z = \phi(v_{Y_1}, v_{Y_2}, \ldots, v_{Y_k})$$

where v_{Y_i} represents the name for v transmitted by node Y_i. The above ϕ-function introduces one def and k uses of v into the program representation.

In the traditional SSA representation [CFR+91], each use of v in the ϕ-function is associated with node Z, where the ϕ-function textually appears. In a departure convenient for our algorithm, we instead associate each use v_{Yi} with node Yi. While the ϕ-function appears at node Z, its uses (arguments) are actually associated with the predecessors of Z. For example, in Figure 1, the uses v_0 and v_1 appearing in the ϕ-function at statement $\boxed{10}$ are actually associated with (the end of) statements $\boxed{5}$ and $\boxed{8}$, respectively.

Throughout this paper, we assume that programs are comprised of a single procedure, and we concentrate primarily on the effects of may-alias information on the quality and speed of program optimization. The effects of procedure calls on storage names, while outside the scope of this paper, can be treated analogously.

Our algorithm as presented does not attempt to discover constants for any use u for which $Symbol(u) = \bot$. Although our approach is easily generalized to handle such cases, we concentrate instead on the impact of may-alias information on constant propagation for readily identified uses.

3 Algorithm

In this section, we describe how to incrementalize an SSA-based optimization problem to accommodate *IsAlias*() functions missing from SSA_i but present in SSA_{i+1}. We then describe how to compute SSA_{i+1} from SSA_i by including relevant may-alias information.

The algorithm is shown in Figure 3. The main procedure contains the interaction of constant propagation with the computed sequence of partial SSA forms. The incrementalized constant propagator (described in Section 3.1) is invoked on:

Form, which represents the current partial SSA form, and

DList, which represents those definition sites that have changed since the previous invocation of *IncCProp*().

and returns:

Soln, which contains the constant value attributed to each definition site (where $\bot$ represents non-constant), and

UList, which contains a list of *suspicious* uses: incorporation of additional may-alias information could adversely affect their solution.

The following properties, formally proven in Section 4, are essential to our algorithm's performance:

1. For each use $u \in ULIST$, *Update*() discovers the definition d that would reach u in SSA_∞.
2. When *IncCProp*() returns an empty *UList*, constant propagation convergence has been achieved.

3.1 Incrementalizing the Optimization

A generally incremental constant propagation algorithm must address arbitrary modifications of a program, including those that substantially alter program flow or the contents of the associated flow graph's nodes. We shall see that the changes introduced by step $\boxed{18}$ are of three restrictive forms:

1. When $v \in MayAlias(d)$ is accommodated at step $\boxed{27}$, a statement of the form

$$v = IsAlias(Symbol(d), v)$$

is inserted into $Node(d)$ by step $\boxed{28}$.

2. Insertion of a definition for v at node X may introduce a ϕ-function for v at node Z, where X dominates some predecessor Y_m of Z, but X doesn't strictly dominate Z. If Z has k predecessors, then a statement of the form

$$v = \phi(v^1, v^2, \ldots, v^k)$$

```
Algorithm[1] Constant Propagation
    foreach (d ∈ DefSites) do
        Visited(d) ← 0
    od
    VisitStamp ← 0
    Form ← SSA_0(Prog)
    DList ← DefSites                                  ⇐ [15]
    [Soln, UList] ← IncCProp(Form, DList)             ⇐ [16]
    while (UList ≠ ∅) do                              ⇐ [17]
        VisitStamp ← VisitStamp + 1
        [Form, DList] ← Update(Form, UList)           ⇐ [18]
        [Soln, UList] ← IncCProp ( Form,
                                   DList              ⇐ [19]
                                 )
    od
end
```

```
Function Update(Form, SuspiciousUses) : [Form, DList]
    DList ← ⊥
    foreach (u ∈ SuspiciousUses) do                   ⇐ [20]
        if (∃ d | Symbol(u) ∈ MayAlias(d)) then       ⇐ [21]
            NewDef ← Snoop ( DomDef(u),
                             Rdef(u),                 ⇐ [22]
                             Rdef(u)
                           )
            if (NewDef ≠ ⊥) then
                Rdef(u) ← NewDef                      ⇐ [23]
            fi
        fi
    od
end
```

```
Function Snoop(StartDef, StopDef, Rdef) : Def
    d ← StartDef
    while (d ≠ StopDef) do                            ⇐ [24]
        Visited(d) ← VisitStamp                       ⇐ [25]
        if (Symbol(d) = Symbol(Rdef)) then
            DList ← DList ∪ { d }                     ⇐ [26]
            return (d)
        fi
        if (Symbol(Rdef) ∈ MayAlias(d)) then          ⇐ [27]
            NewDef ← CreateIsAlias(d, Rdef)           ⇐ [28]
            DList ← DList ∪ { NewDef }                ⇐ [29]
            return (NewDef)
        fi
        if (d = FirstRef(node(d))) then
            IDef ← ⊥
            foreach (m ∈ Preds(node(d))) do           ⇐ [30]
                if (Visited(LastRef(m)) ≠ VisitStamp) then   ⇐ [31]
                    IDef ← Snoop ( LastRef(m),
                                   LastRef(idom(node(d))),   ⇐ [32]
                                   Rdef
                                 )
                fi
                if (IDef ≠ ⊥) then
                    NewDef ← CreatePhi(d, Rdef, m, IDef)     ⇐ [33]
                    DList ← DList ∪ { NewDef }               ⇐ [34]
                    return (NewDef)
                fi
            od
        fi
        d ← DomDef(d)
    od
    return (⊥)
end
```

In the following, r represents any superscripted reference.

$CreateIsAlias(d, Rdef)$: inserts the statement

$$d^1 = IsAlias(u^1, u^2)$$

after def d, with

$$\begin{aligned} Symbol(d^1) &= Symbol(Rdef) \\ Symbol(u^1) &= Symbol(d) \\ Symbol(u^2) &= Symbol(Rdef) \\ Rdef(u^1) &= d \\ Rdef(u^2) &= Rdef \\ DomDef(r) &= d \\ Node(r) &= Node(d) \end{aligned}$$

$CreatePhi(d, Rdef, m, IDef)$: inserts the statement

$$d^1 = \phi(u^1, \ldots, u^k)$$

at the beginning of node $Node(d)$ with indegree k, setting

$$\begin{aligned} DomDef(d) &= d^1 \\ Symbol(r) &= Symbol(Rdef) \\ Node(d^1) &= Node(d) \end{aligned}$$

$$Node(u^i) = \text{Pred } i \text{ of } Node(d)$$

$$DomDef(u^i) = LastDef(Node(u^i))$$

$$Rdef(u^i) = \begin{cases} Rdef & i \neq m \\ IDef & i = m \end{cases}$$

Figure 3. Algorithm.

is inserted at the beginning of node Z by step $\boxed{33}$. [1] As discussed earlier, each use v^i is actually associated with predecessor Y_i of Z.

3. The SSA name associated with a use u, maintained as $Rdef(u)$, changes at step $\boxed{23}$ because of a def inserted by either of the above modifications.

The above modifications occur in response to accommodation of may-alias information. We prove in Section 4 that such modifications cannot "better" the data flow solution for any variable at any node. Thus, an incremental constant propagator can assimilate the definitions provided by step $\boxed{18}$ without restarting any of its computations.

In summary, an SSA-based constant propagator [WZ91] is incrementalized by:

exposing its definition worklist: Step $\boxed{15}$ provides the initial set of definitions, and a new list is developed each time step $\boxed{18}$ executes;

flagging its suspcious uses: Any use u can be returned to $Update()$ as *suspicious*, in which case $Update()$ will invoke $Snoop()$ to incorporate relevant may-alias information.

A sufficient condition for flagging a use u suspicious exists when u's constant propagation solution is non-$\perp$.

The correctness and termination conditions for our algorithm place the following demands on the incrementalized constant propagator:

1. Any use u that contributes to a non-$\perp$ solution for some def must be flagged as suspicious during some invocation of $IncCProp()$.

2. A use u is flagged as susicious at most once. By maintaining a single bit with each use, the constant propagator can "remember" whether it has ever flagged a given use as suspicious, and avoid repeatedly placing the same use on $UList$.

3.2 Update and Snoop

The $Update()$ procedure shown in Figure 3 considers each suspicious use u at step $\boxed{20}$, and finds (through $Snoop()$) the definition of $Symbol(u)$ associated with u in SSA_∞. If a suitable definition is found, then step $\boxed{23}$ modifies $Rdef(u)$.

Though absent from our initial implementation, we soon discovered the value of step $\boxed{21}$, which determines if *any* may-aliases are associated with $Symbol(u)$. This step precludes from consideration those uses whose reaching definition in SSA_∞ was correctly determined in SSA_0. The formidable predicate at step $\boxed{21}$ is easily established by maintaining a single bit with each storage name, indicating whether the symbol is mentioned in any of the program's may-alias relations.

In any partial SSA form, $Rdef(u)$ contains the definition of $Symbol(u)$ whose "name" is currently associated with u, and $Node(Rdef(u))$ necessarily dominates $node(u)$. [2] Any change in $Rdef(u)$ must occur because some definition of $Symbol(u)$ intercedes between u and $Rdef(u)$ as additional may-alias information is accommodated. In Figure 3, such a definition might be introduced by $Snoop()$ as the target of an $IsAlias()$ function (at step $\boxed{29}$), or as the target of a related ϕ-function (introduced by step $\boxed{34}$).

Definition 1 *Consider two definitions d and D, of potentially differing storage names, such that D dominates d. We define $DomStretch(d, D)$ as the list of dominating definition sites from d to D, excluding D:*

$$(d, DomDef(d), DomDef(DomDef(d)), \ldots, \Delta)$$

where $DomDef(\Delta) = D$.

Thus, $Snoop()$ is invoked at step $\boxed{22}$ to process defs in $DomStretch(DomDef(u), Rdef(u))$.

The function $Snoop()$ proceeds up the dominator tree in search of alias information that directly (via $IsAlias()$) or indirectly (via a ϕ-function) induces a definition of $Symbol(Rdef)$ symbol between $StartDef$ and $StopDef$ (excluding $StopDef$). Although the function is invoked recursively, each invocation returns the appropriate def in $DomStretch$ (or $\perp$ if none exists). The $Visited()$ attribute makes certain that no def is visited twice per suspicious use.

As step $\boxed{24}$ iterates over each def site $d \in DomStretch(StartDef, StopDef)$, one of the following conditions concludes the search:

1. A def of $Symbol(Rdef)$ is encountered in the current SSA form. Because the search excludes $Rdef$, d dominates the suspicious use u, and d is strictly dominated by $Rdef$; the next partial SSA form must reflect that $Rdef(u) = d$. $Snoop()$ therefore adds d to the constant propagator worklist (because its set of reached uses changes), and returns d; step $\boxed{23}$ then updates $Rdef(u)$.

2. A def is encountered whose may-aliases included $Symbol(Rdef)$. Step $\boxed{28}$ inserts the appropriate definition of $Symbol(Rdef)$ through an $IsAlias()$ function, step $\boxed{29}$ adds the new def to the constant propagator worklist, and $Snoop()$ returns this new def.

3. A recursive invocation of $Snoop()$ from step $\boxed{32}$ has established a def that requires insertion of a ϕ-function. Step $\boxed{33}$ creates the appropriate ϕ-function, step $\boxed{34}$ adds the new def to the worklist, and $Snoop()$ returns the new def.

3.3 Example

We illustrate the workings of our algorithm using the example shown in Figure 4. The SSA_0 program is a slight variant of our early example in Figure 1. Each dereferenced pointer serves as a may-alias for v only.

In the initial round of constant propagation, the use of v_0 at $\boxed{38}$ is found to be 3, since its reaching definition at $\boxed{35}$ has that value. As a result, the use of v at $\boxed{38}$ is flagged as suspicious by the constant propagator and placed on $UList$. Recalling that a def site is inserted at the beginning of nodes that otherwise lack an initial def site, $Snoop()$ is called on $DomStretch(\boxed{37}, \boxed{35})$. Because $\boxed{37}$ is the first def in its node, step $\boxed{30}$ explores predecessors of $\boxed{37}$. Accordingly, $Snoop()$ is called recursively for $DomStretch(\boxed{36}, \boxed{35})$. Statement $\boxed{36}$ is then discovered as a may-alias for v, adding the $IsAlias()$ at $\boxed{41}$ and, upon return from recursion, the ϕ at $\boxed{42}$. The suspicious use of v at $\boxed{38}$ is renamed v_2—its reaching definition in SSA_∞.

The next round of constant propagation will accommodate the new defs v_1 and v_2, but each is found to be the constant 3. However, three suspicious uses are added to $UList$ for the next partial SSA form: the two uses of v at $\boxed{42}$ (actually associated with predecessors $\boxed{41}$ and $\boxed{40}$) and the use at $\boxed{41}$.

---

[1] We use superscripts to represent the i-th argument of the ϕ-function, as subscripts might connote a *name* for the i-th argument.

[2] In traditional SSA form, this property would be true only for ordinary uses. Because we push ϕ-uses into predecessors of the ϕ-function, this property is true for all uses in our representation.

SSA_0	SSA_1	SSA_2

$v_0 \leftarrow 3$ $\Leftarrow \boxed{35}$
$\star r \leftarrow 3$
if (b) **then**
  $\star q \leftarrow 3$
else
  $\star p \leftarrow 6$
  $\star s \leftarrow 3$ $\Leftarrow \boxed{36}$
fi
$Null \leftarrow$ $\Leftarrow \boxed{37}$
$c \leftarrow v_0$ $\Leftarrow \boxed{38}$
	$v_0 \leftarrow 3$ $\Leftarrow \boxed{39}$
	$\star r \leftarrow 3$
	if (b) **then**
	  $\star q \leftarrow 3$ $\Leftarrow \boxed{40}$
	else
	  $\star p \leftarrow 6$
	  $\star s \leftarrow 3$
	  $v_1 \leftarrow IsAlias(s, \&v_0)$ $\Leftarrow \boxed{41}$
	fi
	$v_2 \leftarrow \phi(v_0, v_1)$ $\Leftarrow \boxed{42}$
	$Null \leftarrow$
	$c \leftarrow v_2$
		$v_0 \leftarrow 3$
		$\star r \leftarrow 3$ $\Leftarrow \boxed{43}$
		if (b) **then**
		  $\star q \leftarrow 3$
		  $v_3 \leftarrow IsAlias(q, \&v_0)$ $\Leftarrow \boxed{44}$
		else
		  $\star p \leftarrow 6$
		  $v_4 \leftarrow IsAlias(p, \&v_0)$ $\Leftarrow \boxed{45}$
		  $\star s \leftarrow 3$
		  $v_1 \leftarrow IsAlias(s, \&v_4)$ $\Leftarrow \boxed{46}$
		fi
		$v_2 \leftarrow \phi(v_3, v_1)$ $\Leftarrow \boxed{47}$
		$Null \leftarrow$ $\Leftarrow \boxed{48}$
		$c \leftarrow v_2$

Figure 4. Example.

1. Processing the use at $\boxed{41}$ (not shown—associated with the ϕ at $\boxed{42}$) invokes $Snoop()$ on $DomStretch(\boxed{41}, \boxed{41})$ and so no work is done.
2. Processing the use at $\boxed{40}$ (not shown—associated with the ϕ at $\boxed{42}$) invokes $Snoop()$ on $DomStretch(\boxed{40}, \boxed{39})$, and the $IsAlias()$ at $\boxed{44}$ is added, with the use of v changed at $\boxed{47}$.
3. Processing the use at $\boxed{41}$ introduces the $IsAlias()$ at $\boxed{45}$, which changes the use of v at $\boxed{46}$.

When constant propagation is again executed, the definition at $\boxed{45}$ becomes $\perp$, which makes the definition at $\boxed{46}$ become $\perp$, which makes the definition at $\boxed{47}$ become $\perp$, which makes the use of v at $\boxed{48}$ $\perp$.

Constant propagation then returns an empty list of suspicious uses, so the solution has converged. Accommodation (i.e., explicit representation) of the may-alias for v at $\boxed{43}$ is unnecessary with respect to constant propagation.

4 Analysis

REALLY NEW STUFF Begins 3/24/93

We begin our analysis by proving that covergence is reached in our sequence of partial SSA forms. This analysis will show that the optimization solution in SSA_{i+1} cannot be "better" (in terms of data flow lattice) than the solutions for SSA_i.

Lemma 1 *If $v_u \in Refs(SSA_i)$ and $Rdefs_{SSA_i}(v_u) \neq Rdefs_{SSA_{i+1}}(v_u)$, then $Rdefs_{SSA_{i+1}}(v_u)$ is in one of the following forms:*

1. *$Rdefs_{SSA_{i+1}}(v_u) = IsAlias(*p, V)$*
2. *$Rdefs_{SSA_{i+1}}(v_u) = \phi(v_1, v_2, \ldots, V, \ldots, v_n)$*

Where $Lattice(V) \leq Lattice(Rdefs_{SSA_i}(v_u))$.

Proof: Any definition added to SSA_0 will in one of these forms. Only newly added definitions will become new reaching definitions.

Lemma 2 *If $v_u \in Refs(SSA_i)$ and $Rdefs_{SSA_i}(v_u) \neq Rdefs_{SSA_{i+1}}(v_u)$, then $Lattice(Rdefs_{SSA_{i+1}}(v_u)) \leq Lattice(Rdefs_{SSA_i}(v_u))$.*

Proof: This follows directly from Lemma 1 and the properties of the IsAlias and *phi* functions.

Lemma 3 *If $v_d \in Defs(SSA_i)$, then $Lattice_{SSA_{i+1}}(v_d) \leq Lattice_{SSA_i}(v_d)$.*

Proof: By Lemma 2 the lattice values of the uses that define definition, v_d will not become "better" in successive SSA forms. Therefore, the lattice value of v_d will not become "better" in successive SSA forms.

Theorem 1 *For all references $v \in Form(SSA_{i+1})$ and $Form(SSA_i)$, the $Lattice_{SSA_{i+1}}(v) \leq Lattice_{SSA_i}(v)$.*

Proof: This follows directly from Lemmas 1, 2, and 3.

To compute the SSA_∞ lattice value of use u, the $Rdef_{SSA_\infty}(u)$ must be determined. For definition v, the $Lattice_{SSA_\infty}(v)$ is derived from the $Lattice_{SSA_\infty}$s of its contributing uses.

The placement of ϕ-function uses to the nodes from which they came is necessary for the following lemmas.

Lemma 4 *For use u, Rdef(u) will be in the DomStretch(DomDef(u), FirstDef(Entry)).*

Proof: Node(Rdef(u)) must be on a path from ENTRY to Node(u). If Node(Rdef(u)) does not dominate Node(u), then $\exists$ node X such that X $\in$ IteratedDominanceFrontier(Node(Rdef(u))) and X dominate Node(u). Since X exists, there is a ϕ-function of Symbol(u) in node X. The shortest path from X to Node(u) is shorter than the shortest path from Node(Rdef(u)) to Node(u). Therefore, Node(Rdef(u)) will dominate Node(u).

THAT PROOF NEEDS HELP

Lemma 5 *A ϕ-function for variable V in node N will occur if and only if there is a definition of v in node M where N $\in$ IteratedDominanceFrontier(M) and Idom(N) strictly dominates M.*

Lemma 6 *If $Rdefs_{SSA_i}(v_u) \neq Rdefs_{SSA_\infty}(v_u)$, then the following properties of $Rdefs_{SSA_\infty}(v_u)$ are true:*

1. *Node($Rdefs_{SSA_\infty}$) dominates Node(v_u). This follows from lemma 4.*
2. *If $Rdefs_{SSA_\infty}(v_u)$ is in the form: $Rdefs_{SSA_\infty} = IsAlias($ *p, Symbol(v_u)) *p may alias symbol(v_u) and a definition of *p occurs in node N that dominates Node(v_u).*
3. *$Rdefs_{SSA_\infty}(v_u)$ is in the form: $Rdefs_{SSA_\infty}(v_u) = \phi(v_1, \ldots, v_n)$ Node($Rdefs_{SSA_\infty}(v_u)$) $\in$ IteratedDominanceFrontier(Node(a definition of Symbol(v_u).*

Lemma 7 *Given use u, Snoop() will find* $Rdefs_{SSA_\infty}(u)$*.*

Proof: Snoop() goes up the DomStretch(StartDef, StopDef) in while loop $\boxed{24}$. It determines Rdefs in lemma 6 part 2 in $\boxed{27}$. It determines Refs of lemma 6 part 3 in $\boxed{30}$.

Lemma 8 *The* $Lattice_{SSA_\infty}$ *of definition* v_d *can be determined if all uses that define it have their* $Lattice_{SSA_\infty}$*s.*

Lemma 9 *The* $Lattice_{SSA_\infty}$ *of use* v_u *can be determined if* $Rdef(v_u) = Rdefs_{SSA_\infty}(v_u)$ *and* $Rdefs_{SSA_\infty}$ *has its* $Lattice_{SSA_\infty}$*.*

Theorem 2 $SSA_{i+1} = SSA_\Omega$ *when:*

1. *For all definitions* $v_d \in (DefSites(SSA_0) and DefSites(SSA_{i+1}))$*, the lattice value of* v_d *is equal in* SSA_{i+1} *and* SSA_∞*.*

2. $\exists definition v_d \in (DefSites(SSA_0) and DefSites(SSA_\Omega))$*, for which the lattice values* v_d *is not equal in* SSA_i *and* SSA_∞*.*

Lemma 10 *For all uses u, if the Form Rdef(u))* $\neq SSA_\infty$ *Rdef(u), u defines definition v, and v does not have its* SSA_∞ *lattice value, then IncCProp() will put u on UList.*

Theorem 3 *When the condition at* $\boxed{17}$ *is fulfilled, Soln will be* SSA_Ω*.*

REALLY NEW STUFF ENDS 3/24/93 Other stuff about how the algorithm works.

Theorem 4 *With respect to a given use* v_U*, the program changes described above either leave the data flow solution for* v_U *unchanged, or else drive the solution* lower *in a data flow lattice.*

Proof: **In full paper.** □

Each element in our sequence of partial SSA forms is *partial* in two senses:

1. Each may lack the full $IsAlias()$ information present in SSA_∞. Correspondingly, certain ϕ-functions may also be absent.

2. The "anonymous" uses introduced by $Snoop()$ may lack the "reaching definition" information associated with SSA form.

The first point should be expected, since we wish to avoid the space (and associated time) expense of computing full SSA form. The second point is necessary for our time bounds, in that we desire performance related to the amount of "change" necessary to accommodate freshly-incorporated may-alias information. If our algorithms spent time determining the reaching definition of every use introduced by $Snoop()$, then the resulting partial SSA form would be more complete, but we could not guarantee the usefulness of such work. We therefore wait until a use appears on the suspicious $UseList$ before we determine its reaching definition.

In the full paper, we offer proofs of our algorithm's correctness. Our algorithm's worst-case performance is easily analyzed, though the result is not enlightening in an incremental setting. Let U be the number of suspicious uses identified during the optimization problem (including all iterations of loop $\boxed{17}$). In the worst-case, every edge in the flow graph (at step $\boxed{30}$) must be considered. Our algorithm's performance is thus $O(UE)$, where E is the number of edges in the flow graph. The term U is a function of the number of uses in the original SSA_0 text, as well as the number of $IsAlias()$ and ϕ functions inserted by our algorithm. **In the full paper** we present empirical measurements of these quantities.

As should be noted from the example presented in Figure 4, further investigation of suspicious uses occurs only when incorporation of may-alias information does not sufficiently degrade the optimization solution. It's very unlikely that may-aliases of a name are all assigned the same constant value. We hypothesize that programs contain either:

- a great deal of may-aliases, in which case the optimization problem converges quickly to $\perp$ around such aliases; or,

- scant may-alias inforamation, which can be included with a small number of iterations of our algorithm (the loop at step $\boxed{17}$).

While our experience with the algorithm indicates very good performance in practice (Section 5), the asymptotic performance can be improved by introducing *path-compression* [Tar75] into $Snoop()$. As an associated expense, space would have to be allocated to maintain such information per-variable, or the $UseList$ would have to be sorted by symbol name.

5 Experiment

We performed the following experiment on 139 Fortran programs taken from the Perfect [BCK+88] (*Ocean*, *Spice*, *QCD*) benchmark suite and from the Eispack [SBD+76] and Linpack [DBMS79] subroutine library. Since Fortran (77) has no pointer dereferencing capability, we randomly associated may-alias information with each program using the following two parameters:

AliasRatio R**:** the probability that a given definition site is may-aliased with any symbols.

AliasWidth W**:** the probability that a given symbol is included in a definition site's may-aliases.

The pair (R, W) can then model a wide variety of alias behavior:

(0,0) no definition site has any may-aliases. With this setting, the exhaustive algorithm should outperform our incremental algorithm.

(1,1) each definition site is may-aliased with every symbol. With this setting, our incremental algorithm should outperform the exhaustive algorithm.

(.25,.5) on average, one out of four definition sites has may-aliases; when this occurs, half of the symbols are involved, on average.

The following table shows the median speedup of our method over the exhaustive method:

Alias Ratio	Alias Width		
	25%	50%	100%
0%	1.0		
25%	2.6	4.3	8.1
50%	3.4	6.0	11.6
100%	5.2	9.9	20.7

Figure 5 plots the speedups we obtained when, on average, one-fourth of a procedure's definition sites alias one-half of the variable names. Program size is measured by the number of nodes in the associated flow graph. Our method shows significant speedup, especially on the larger programs.

- In the no-aliasing situation, the exhaustive algorithm did outperform the incremental, though the median was 1.0. The only significant cases were two programs of 20 or fewer statements, where our speedup was 0.7, but these programs took scant time anyway. For one program of 132 statements, our speedup was 0.86. For all others, we were close to 1.0.

- In the completely-aliased situation, the space requirements for exhaustive SSA exceeded the 156M virtual storage capacity of the host machine for 12 programs. [3] In our best showing, we obtained a speedup of 60 for three programs of 128 to 263 flow graph nodes (none of these aborted).

6 Conclusions and open questions

In this abstract, we focussed on the optimization of constant propagation. However, the mechanism by which the optimization was coupled to our algorithm involves the definition and use sites of a partial SSA form. Any optimization based on SSA form (value numbering, code motion, etc.) must be concerned with the interplay between definitions and uses. The introduction of monotonic information into a subsequent partial SSA form does not require "restarting" a data flow problem based on that information. Thus, data flow optimizations based on SSA form are easily incrementalized in the style by which we incrementalized constant propagation. We expound on this issue **in the full paper**.

In developing optimization algorithms for programs with pointers, one must either special-case the behavior of pointers, or else fully and accurately model the effects of pointer dereferencing. While the latter approach allows for a cleaner optimization algorithms—including the results of sophisticated (may-)alias analysis—representing all effects of all stores through pointers can significantly increase the size of a program's representation. Processing that representation will necessarily incur greater expense during each phase of optimization.

Our experiments show that accommodating may-alias information incrementally can significantly improve optimization-time performance while still allowing optimizations to be formulated without special-cases for pointers. Even where may-aliases are rare, our incremental algorithm does not perform poorly when compared with its exhaustive counterpart.

We suggest the following avenues of further research on this problem:

- Our work to date addresses only "flat" name spaces, so we hope to extend the work to accommodate information developed for structure references [LH88, CWZ90, HN90].

- Our algorithm may be suitable for including information similar to may-aliases incrementally into sparse data flow evaluation graphs [CCF91].

- Our experiments are based on simulated (probabilistic) may-alias patterns. We would like to try our algorithm in a system that actually analyzes C programs for may-aliases.

Acknowledgements

We are grateful to Barbara Ryder and Bill Landi for inspiring this work through their ambitious alias analysis, and we thank them for many enlightening conversations during our work on this problem. We thank the PTRAN and RISC compiler groups at IBM Research for their help and comments at the early stages of this work. We thank Ron Loui, administrator of the SURA NSF-funded program at Washington University, for sponsoring Reid Gershbein's work on this topic. We thank Will Gillett, Vivek Sarkar, and Mark Wegman for their careful reading of this paper and for their helpful comments. The final formulation of this paper was guided by suggestions from the program committee members, and we are grateful for their assistance.

[3] Timings for these 12 runs were taken at the abort point.

References

[AWZ88] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of values in programs. *Conf. Rec. Fifteenth ACM Symp. on Principles of Programming Languages*, pages 1–11, January 1988.

[BCK+88] M. Berry, D. Chen, P. Koss, D. Kuck, S. Lo, Y. Pang, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsiung, J. Schwarzmeier, K. Lue, S. Orszag, F. Seidl, O. Johnson, G. Swanson, R. Goodrum, and J. Martin. The perfect club benchmarks: Effective performance evaluation of supercomputers the performance evaluation club (perfect). Technical report, U. of Ill–Center for Supercomputing Research and Development, November 1988.

[Bod90] F. Bodin. Preliminary report – Data structure analysis in C programs. Indiana University, Bloomington, March 1990.

[CBC93] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. *Conference Record of the Twentieth Annual ACM SIGACT–SIGPLAN Symposium on Principles of Programming Languages*, January 1993.

[CCF91] J. Choi, R. Cytron, and J. Ferrante. Automatic construction of sparse data flow evaluation graphs. *Conf. Rec. Eighteenth ACM Symp. on Principles of Programming Languages*, pages 55–66, January 1991.

[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*, October 1991.

[CWZ90] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. *Proc. SIGPLAN'90 Symp. on Compiler Construction*, pages 296–310, June 1990. Published as *SIGPLAN Notices* Vol. 25, No. 6.

[DBMS79] J. J. Dongarra, J. R. Bunch, C. B. Moler, and G. W. Stewart. *Linpack Users' Guide*. SIAM Press, 1979.

[HN90] L. J. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Trans. on Parallel and Distributed Systems*, 1(1):35–47, 1990.

[Lan92] William A. Landi. *Interprocedural Aliasing in the Presence of Pointers*. PhD thesis, Rutgers, The State University of New Jersey, 1992.

[Lan93] William A. Landi. personal communication, 1993.

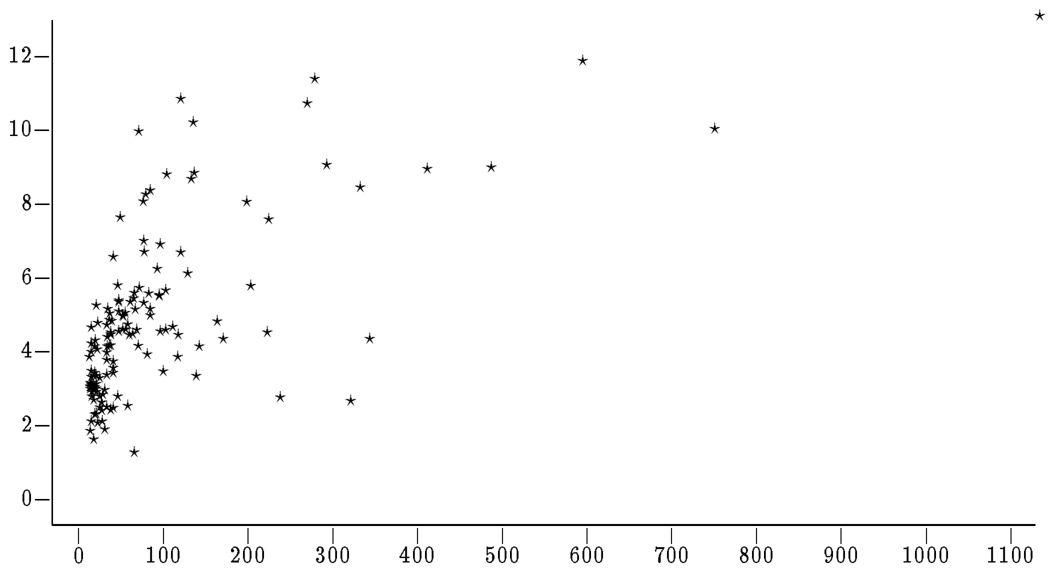

Figure 5. Plot of speedup vs. program size for $(R, W) = (.25, .5)$.

[LH88] J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. *Proc. SIGPLAN'88 Symp. on Compiler Construction*, pages 21–34, July 1988. Published as *SIGPLAN Notices* Vol. 23, No. 7.

[LR92] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 235–248, June 1992.

[LT79] T. Lengauer and Robert Tarjan. A fast algorithm for finding dominators in a flowgraph. *TOPLAS*, July 1979.

[SBD$^+$76] B. T. Smith, J. M. Boyle, J. J. Dongarra, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler. *Matrix Eigensystem Routines – Eispack Guide*. Springer-Verlag, 1976.

[Tar75] Robert Tarjan. Efficiency of a good but not linear set union algorithm. *JACM*, 22:215–225, 1975.

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