On the Importance of Program Representations in Static Analysis

Xiao Xiao

Supervisor: Charles Zhang

Department of Computer Science and Engineering

The Hong Kong University of Science and Technology

January 28, 2013

Abstract

Static analysis is a set of automatic or semi-automatic techniques to gather the semantic knowledge of a program without executing it. An important kind of semantic knowledge is the control flow and data flow information, which characterizes the instruction execution paths and the data generation/consumption paths, respectively. The obtained information is encoded in various graphical data structures or in the program itself directly for subsequent use. These data structures or instrumented programs are called program representations. In this survey, we first study four data structures that can help us dissect the code structure and dig deep structural properties. Then, we discuss the popular representations used in contemporary compiler, the single static assignment (SSA) form and its variants, which enable efficient program analysis algorithms by exploiting the sparsity of the data flow problem. We believe our surveyed knowledge can shed light to our future work.

1 Introduction

Software engineering and programming language research focuses on improving the programmers' productivity and the program code efficiency, which significantly benefit from the advances in the way of gathering the semantic knowledge of a program. *Static analysis* is a category of techniques to explore a program without executing it. The main theme of static analysis is to infer the behaviors of a program and answer the questions such as, which statements *must* be executed after statement A? Which variables will be modified by statement A? What is the relationship in value of the variables a and b? These questions are often addressed by the *control flow* and *data flow* analysis, and of course, by many other methods, which are out the scope of this survey. The control flow analysis characterizes how the code is executed, such as computing the dominator tree of a procedure [1], testing the reducibility of a program [44], computing the control dependencies for all the statements [32], computing the single entry single exit regions of a procedure [25], etc. The data flow analysis estimates the values of the variables, such as tracking the possible targets for pointers [18], computing the def-use chains [1], detecting induction variables [47]. The collected semantic information can provide vision for subsequent algorithms to comprehend and manipulate code.

In a code analysis tool such as compiler, static analysis algorithms are usually performed one by one, where the subsequent algorithms require the outcome of the former algorithms as input. By this reason, it is important to encode the analysis results of an algorithm in some way for future use.

The encoded form of static analysis results is called *program representation*. Usually, we either build additional graphical data structures to encode the analysis results, such as we use the dominator tree to query the dominance information [1]. Or, we directly inject the semantic information into the code and construct an *intermediate representation* (IR). Both solutions have their advantages. Adding a new data structure for a piece of information does not require the modification to the existing structures and algorithms, and most often, it is easy to implement. In contrast, employing a new IR sometimes needs to change all the algorithms that are built on the particular IR syntax. Moreover, it is hard to encode rich information with succinct syntax. However, using a proper IR, the algorithms built on can often be greatly simplified. For example, with SSA IR [12], a simple flow insensitive points-to analysis can transparently obtain certain degree of flow sensitive precision [20].

In the case more than one tools are used to process the code, using IR to exchange the information among these tools is a good choice, since these tools may be implemented by different vendors, they cannot negotiate in advance to use the same data structure to encode the same information.

1.1 Paper Organization

This survey is not a complete summary of the program representations developed in use today. Instead, we selectively introduce several important representations, each of which forms the basis for a wide spectrum of applications. In summary, we will examine the following graphical data structures:

- Loop nesting forest, a data structure that represents the loops in a control flow graph and the containment relation between them [33]. This structure can be used to perform the elimination based data flow analysis [36], and also, it can be used to construct the SSA IR [12] and sparse evaluation graph [10].

- DJ-graph, a data structure that combines dominator tree and control flow graph. It is very useful in both exhaustive and incremental data flow analysis [41]. And recently, it is applied to answer the liveness queries in a demand-driven way [13].

- Program structure tree, a hierarchical representation of program structure based on single entry single exit (SESE) regions of control flow graph [25]. This decomposition of program can be exploited to speed up data flow analysis in divide-and-conquer fashion and compute data flow analysis in parallel.

- Program dependence graph, a graph that only encodes the control and data dependence relations between the statements in the same procedure [14]. This is graph is intensively used in software engineering tools, such as test case generation [17], slicing [24], and code clone detection [15].

For the intermediate representations, we will study the single static assignment (SSA) form and its variants. SSA encodes the def-use chain information into the code, hence, we can quickly discover the definitions of a variable used by a statement without executing the reaching definition analysis. Many well known compilers such as GCC, LLVM, and open64

use SSA as its primary IR for machine independent code optimization. SSA transparently lends certain degree of flow sensitive precision to flow insensitive algorithms and exposes the opportunity for sparse data flow analysis [29], many algorithms, such as pointer analysis [19], greatly benefits from these features.

However, the original SSA does not consider the impact of the aliased pointers, the interprocedural sides effects, the predicate expressions, and the chance to perform backward data flow analysis sparsely, which limited the use of SSA in sophisticated analysis setting. Fortunately, progress has been made to address these limitations. The aliasing SSA form called Hashed SSA (HSSA) proposed by Chow *et.al* [11] models the def-use impacts of the pointer dereference site. Calman *et.al* [8] and Livshits *et.al* [27] generalizes the HSSA to track the interprocedural side effects. The single static information (SSI) form also considers the variable use sites, other than the definition places, as the information sources. Hence, the predicate information such as $x == 2$ can be used to enrich the subsequent analysis result. All of these extensions greatly simplify the design and implementation of the algorithms that need corresponding information.

2 Terminologies

We repeat the commonly used definitions for several important concepts here. A good summary of the graph terminologies for PL research is given by Offner [30].

Definition 1 *A* **control flow graph** *(CFG)* $G(V, E)$ *is a directed graph. A node* $v \in V$ *is called* **basic block***, which contains a sequence of statements that will be executed once the first statement in the sequence is executed. An edge* $u \to v$ *represents possible flow of control from* u *to* v*. The graph contains two distinguished nodes:* START *has no predecessors and it reaches every node.* End *has no successors and it is reachable from every node.*

CFG is the simplest way to layout the program and encode the program execution flows. Most of the time, CFG is the start of a static analysis algorithm. The **abstract syntax tree** (AST) [1] is another common start abstraction of a program. However, AST does not explicitly model the flow of control between statements, it emphasizes more on the syntax structure of the source code. In contrast, the statements in a CFG are usually transformed

into triple code format [1], *i.e.* a statement involves at most three operands, which is suitable for machine analysis rather than studying the programmer's code style.

However, it is not obvious to discover any interesting results for the program execution flow encoded in CFG. The primary methodology to study the CFG is using *depth first search* (DFS). DFS, as well as the DFS decomposition of a graph by **tree edges**, **forward edges**, **back edges**, and **cross edges**, are well defined concepts in graph theory text book [30]. In addition, we define a family of dominance relations to characterize the execution flow:

Definition 2 *A node or edge x is said to* **dominate** *node or edge y in a CFG if every path from* START *to y includes x. We use x **dom** y to record the dominance relation. If $x \neq y$, x* **strictly dominates** *y and this relation is recorded as x **sdom** y. Finally, x might* **immediately dominate** *y if x **sdom** y and there is no other node z such that x **sdom** z **sdom** y. We write x **idom** y to record the immediate dominance relation.*

Definition 3 *A node or edge x is said to* **postdominate** *node or edge y if every path from y to* END *includes x.*

Definition 4 *A* **dominator tree** *is a graphical representation of the* **idom** *relations. There is an edge $x \rightarrow y$ in the tree iff x **idom** y.*

Definition 5 *The dominance frontier $DF(x)$ of node x is the set of all nodes y such that x dominates a predecessor of y but does not strictly dominate y.*

We extend the definition of dominance frontier to a set S of nodes: $DF(S) = \bigcup_{x \in S} DF(x)$. Then, we define the **iterated dominance frontier** $IDF(S)$ for a set S of nodes as the limit of the following sequence:

$$\begin{aligned} IDF(S) &= DF(S) \\ IDF(S) &= DF(S \bigcup IDF(S)) \end{aligned}$$

Based on the postdominate relation, we define a more involved but still a fundamental concept, **control dependence**:

Definition 6 *A node w is said to be* **control dependent** *on edge $u \rightarrow v$ if:*

1. *w postdominates v;*
2. *if $w \neq u$, then w does not postdominate u.*

Intuitively, control dependence means that if the execution flows along $u \to v$, the control will eventually reach w.

The last fundamental view of the execution flow is loop. A **loop** is a strongly connected component (SCC) [30]. A loop can interfere with another loop, resulting in nested loops. For a well structured program, the program written without **goto**s, a loop is either contained in or disjoint to another loop. This perfect situation is called **reducible** program, which is defined as follows:

Definition 7 *A program is* **reducible** *iff for every back edge $x \to y$ in any DFS decomposition, y **dom** x.*

More properties of reducible graph other than its definition can be found in Hecht *et.al*'s paper [22]. A loop in a reducible CFG is a group of nodes S that form a SCC. There is a unique node $x \in S$ that x dominates all other nodes in S. The node x is called **loop header** and we use a tuple (S, x) to record a loop. The analysis of loops in irreducible graphs is given in next section.

3 Graphical Representations

We present a set of data structures that would be very helpful for researchers to look inside the code structure and design analysis algorithm.

3.1 Loop Nesting Forest

The **loop nesting forest**, informally, is a tree representation for the loop containment relations. For two loops in a reducible program, as we learned, are either disjoint or nested. However, two loops can be overlapped in a irreducible program. As shown in Figure 1, the loop $\{u, w\}$ and the loop $\{w, x\}$ overlap at w. In such a CFG, the concepts outer loop and inner loop are undefined. Therefore, we could draw many different loop nesting forests, which identify 1, 2, 3 loops as shown in Figure 1.

Although we cannot give a precise definition of what is loop and how the loops are nested in a general program, we can define a parametric framework to model all of the loop definitions so far developed by different researchers [33]. The key is to consider a loop in a

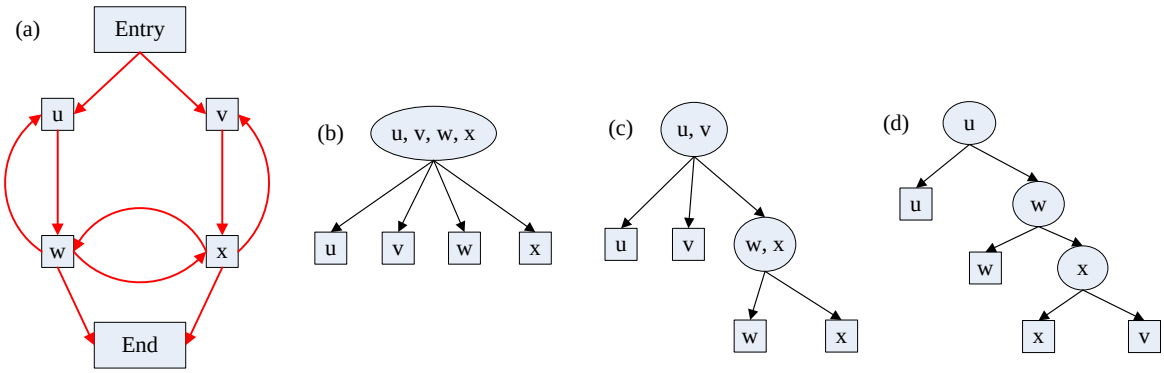


Figure 1: Illustration of different loop nesting forests. (a). A sample CFG. (b). Screedhar-Gao-Lee forest. The loop headers are shown in the ellipse, none cycle members of the loop are shown in the rectangle. (c). Steensgard's forest. (d). Havlak's forest.

graph $G(V, E)$ to be not a set of vertices, but a pair (B, H) of non-empty sets of vertices B and H, with $H \subseteq B$, where B is the **body** of the loop and H is the set of **headers** of the loop. Thus, a loop is an element of $2^V \times 2^V$, where 2^V denotes the powerset of V. A loop nesting forest is *a set of loops*, *i.e.*, a subset of $2^V \times 2^V$. We represent a loop nesting forest as a pair $(\mathbb{B}, \mathcal{H})$, where $\mathbb{B} \subseteq 2^V$ is the set of loop bodies, and $\mathcal{H} \in \mathbb{B} \to 2^V$ is a function that maps each loop body to the set consisting of its loop headers. Thus, $(\mathbb{B}, \mathcal{H})$ is an abuse notation for the set $\{(\mathbb{B}, \mathcal{H}(\mathbb{B}))|B \in \mathbb{B}\}$.

Of course, not every $(\mathbb{B}, \mathcal{H})$ pair is a loop nesting forest, a valid forest should conform to the following properties:

Definition 8 *1. $\forall B \in \mathbb{B}$, B is a non-trivial SCC;*

2. Every non-trivial SCC X in the graph is covered by $(\mathbb{B}, \mathcal{H})$ for some $B \in \mathbb{B}$;

3. The set of loops have the proper nesting property: $\forall B_1, B_2 \in B$, where $B_1 \cap B_2 = \emptyset$ or $B_1 \subseteq B_2$ or $B_2 \subseteq B_1$;

4. A header of the loop is not dominated by any other vertex in the loop.

The parameter in the framework is the way to pick the loop headers. Suppose you have a way to pick the headers, the loop nesting forest $\mathcal{L}$ can be generated as follows. We denote the original CFG as G_0. The first step is identifying all the maximal SCCs in G_0 as the outermost loops. Consider any SCC X, an edge from a vertex inside X to one of its headers

is referred to as a **loopback edge** of X. Then, we delete all the loopback edges from G_0 and obtain G_1. If G_1 is not cycle free, we search the SCCs in G_1 and delete the loopback edges again, until we reach a acyclic graph G_i, with $i \geq 0$.

The various published loop nesting forests are only different in the way of choosing headers, and all of them can be derived from the framework above. The Steensgard's forest [42] picks all the entry vertices of a loop to be the headers. An **entry vertex** is a vertex in a loop that has a predecessor outside the loop. The Havlak's forest [21] each time picks only one header from an SCC, which is the vertex having the least pre-order timestamp in the DFS traversal of G. The last, Sreedhar-Gao-Lee's forest [40], picks all the vertices that are not dominated by other vertices in the same SCC as the headers. These forests for our example are visualized in Figure 1, from pictures (b) to (d).

The loop nesting forest is useful because *certain arbitrary problem instances (including those on irreducible graphs) can be transformed into equivalent problem instances based on acyclic graphs*. We illustrate an application of computing dominator tree with loop nesting forest. The dominator tree for a directed acyclic graph (DAG) is easy to compute due to the following lemma:

Lemma 3.1 *On a DAG, the immediate dominator of node w is the least common ancestor of w's predecessors on the dominator tree.*

Therefore, we can process the vertices in topological order and maintain a dynamic structure for querying the least common ancestor on tree [16]. This algorithm is described by Ramalingam *et.al* [35].

The acyclic algorithm can be easily generalized to reducible cyclic graphs. The key is *the loopback edges in a reducible graph do not impact the dominance relations for any nodes*. Therefore, we remove the loopback edges and apply the acyclic algorithm, then we obtain a dominator tree for reducible graph.

Extending the acyclic algorithm to irreducible graphs requires loop nesting forest, and, suppose we use Steensgard's. In an irreducible graph, the dominator tree for the graphs before and after removing all the loopback edges are not identical. The reasons is *a irreducible loop has multiple entries*. Thus, directly deleting all the loopback edges loses reachability information. However, this information loss can be compensated by adding edges from every pre-entry vertex of the loop to every header of the loop, where a pre-entry vertex is an vertex

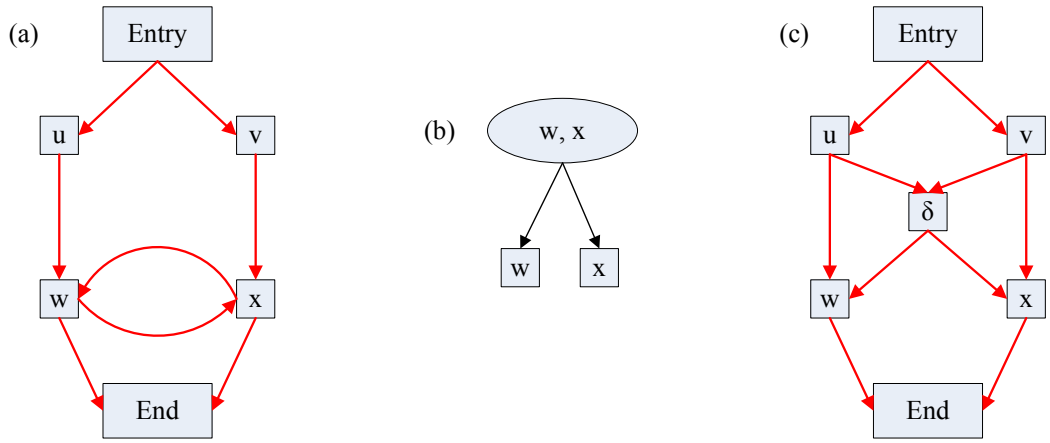


Figure 2: Transformation that preserves the dominance relations.

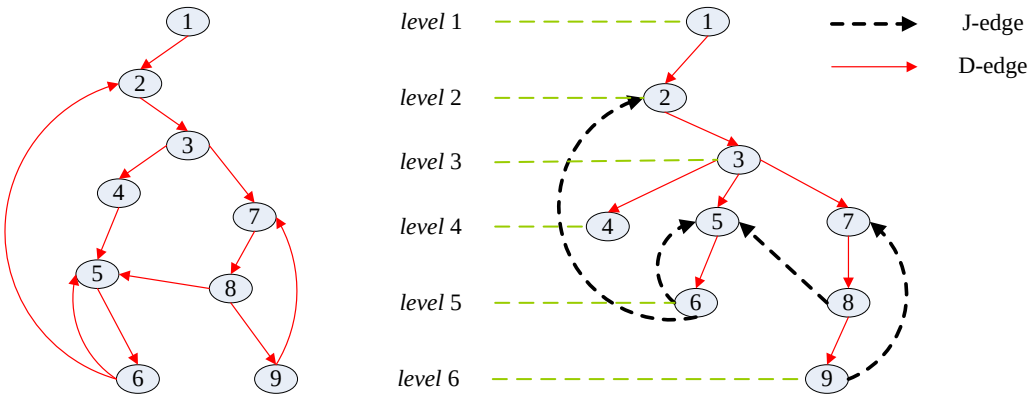
outside loop but it has a successor in the loop. Formally, we transform every loop L from top down in the loop nesting forest as follows:

$$\begin{aligned} V\prime &= V \cup \{\delta_L\} \\ E\prime &= E - \text{Loopback}(L) \\ &\quad \bigcup \{p \to \delta_L \mid p \in \text{PreEntries}(L)\} \\ &\quad \bigcup \{\delta_L \to h \mid h \in \mathcal{H}(L)\} \end{aligned}$$

In the formula above, Loopback(L) denotes the loopback edges in L, and the PreEntries(L) denotes the pre-entry vertices for L. To prevent blowup in the number of newly added edges, we create a hop node δ_L: all the pre-entry vertices connect to δ_L and it connects to all the headers in L. A sample transformation is shown in Figure 2. Let us represent the graph G after applying the transformation to all its loops to be $\Psi(G)$, the set of nodes δ_L to be Δ_L, we can obtain the dominator tree of G with the following formula:

Theorem 3.1 $DomTree(G) = DomTree(\Psi(G)) - \Delta_L$

A more detailed analysis of this algorithm can be found in [33].



(a). a control flow graph (b). DJ graph

Figure 3: DJ graph.

3.2 DJ Graph

A **DJ graph** is just the dominator tree of a CFG augmented with *join* edges [39]. A CFG edge is a join edge (J-edge) if it has no corresponding edge on the dominator tree. A dominator tree edge is called a D-edge. The construction of DJ graph is only a matter of constructing the dominator tree, which can be done in linear time.

A sample DJ graph is given in Figure 3. An interesting part of DJ graph is that it gives a simple way to compute the dominance frontier set $DF(x)$ for any vertex x: We first assign each vertex in G a *level* label, which is the depth from root to that node on the dominator tree. To compute $DF(x)$, we simply visit all the vertices y in the subtree of x. For every J-edge $y \rightarrow z$, $z \in DF(x)$ iff $z.level \leq x.level$. For example, in Figure 3, we want to know $DF(7)$. There are two J-edges in the subtree of 7: $9 \rightarrow 7$ and $8 \rightarrow 5$. Both 7 and 5 have the same level label with 7, hence, $DF(7) = \{5, 7\}$.

In fact, we can bottom-up traverse the dominator tree to compute the dominance frontiers for all vertices in a simple manner. Specifically, $DF(x)$ can be obtained with the following recurrence formula:

$$\begin{aligned} DF(x) \quad = & \bigcup_{y \in subtree(x)} \{w \mid w \in DF(y) \wedge w.level \leq x.level\} \\ & \bigcup \{t | x \xrightarrow{J} t \wedge t.level \leq x.level\} \end{aligned}$$

In the formula, $x \xrightarrow{J} t$ means the J-edge $x \rightarrow t$. The recurrence works in $O(n^2)$ to

compute the dominance frontier for all vertices, however, we can compute the iterative dominance frontier $IDF(S)$ for a set of vertices S in $O(n)$ time. This is because, *although a vertex z can appear in $IDF(x)$, $\forall x \in S$, we only need to add z once to $IDF(S)$ and avoid the revisit of z multiple times.* The pseudo-code of computing $IDF(S)$ is given in Algorithm 1 and it works as follows. First, we put all nodes of S in a container, which supports insertion and return of the node with deepest level label in $O(1)$ time. Then, we repeatedly retrieve the deepest node from the container. For every retrieved vertex x, we call the *SearchIDF(x)* procedure, which first visits the J-edges $x \xrightarrow{J} y$ of x and inserts y to $IDF(S)$ if $x.level \leq y.level$. Meanwhile, if y is never processed by *SearchIDF()* before, we put y into the container. For the D-edges $x \rightarrow t$, if t is not processed by *SearchIDF()*, we directly call *SearchIDF(t)* recursively. Because we only visit every vertex in G once and put every candidate vertex to $IDF(S)$ once, the total running time is $O(|V| + |E|)$.

Algorithm 1: `IDFmain(`S`)`

Input: S: the seed nodes for IDF computation

Output: $IDF(S)$: the iterative dominance frontier for S

```
1  IDF = ∅;
2  foreach x ∈ S do
3  |   InsertNode (x);
4  |   visited(x) = false;
5  end
6  while x = GetDeepestNode () do
7  |   currentRoot = x;
8  |   visited(x) = true;
9  |   SearchIDF (x);
10 end
```

3.3 Program Structure Tree

The **program structure tree** (PST) is a hierarchical representation of the control structure of a program [25]. A sample PST is given in Figure 4. Nodes in this tree represent **single entry single exit** (SESE) regions of the program, while edges represent nesting of edges.

Algorithm 2: `SearchIDF(`x`)`

Input: x: a DJ graph node x

```
1  foreach node y of x's successor do
2      if x → y is a J-edge then
3          if y.level ≤ currentRoot.level then
4              IDF(S) = IDF(S) ∪ y;
5              if y ∉ S then
6                  InsertNode (y);
7              end
8          end
9      else
10         if visited(y) == false then
11             visited(y) = true;
12             SearchIDF (y);
13         end
14     end
15 end
```

The SESE region is defined on the dominance and postdominance relations of edges, which are general extension of dominance on nodes:

Definition 9 *A SESE region in a graph G is an ordered edge pair (a, b) of distinct control flow edges a and b where:*

1. *a dominates b;*
2. *b postdominates a;*
3. *Every cycle containing a also contains b and vice versa.*

However, the above definition along cannot help us partition a program because, two SESE regions can overlap but not always be nested or tandem. For example, (a, b) and (c, d) regions can both be SESE regions and c is contained in (a, b). Complete enumeration of the SESE regions is of no use. We need a hierarchical layout of the SESE regions similar to the loop nesting forest. To address this problem, we require every SESE region recorded in PST to be *canonical*:

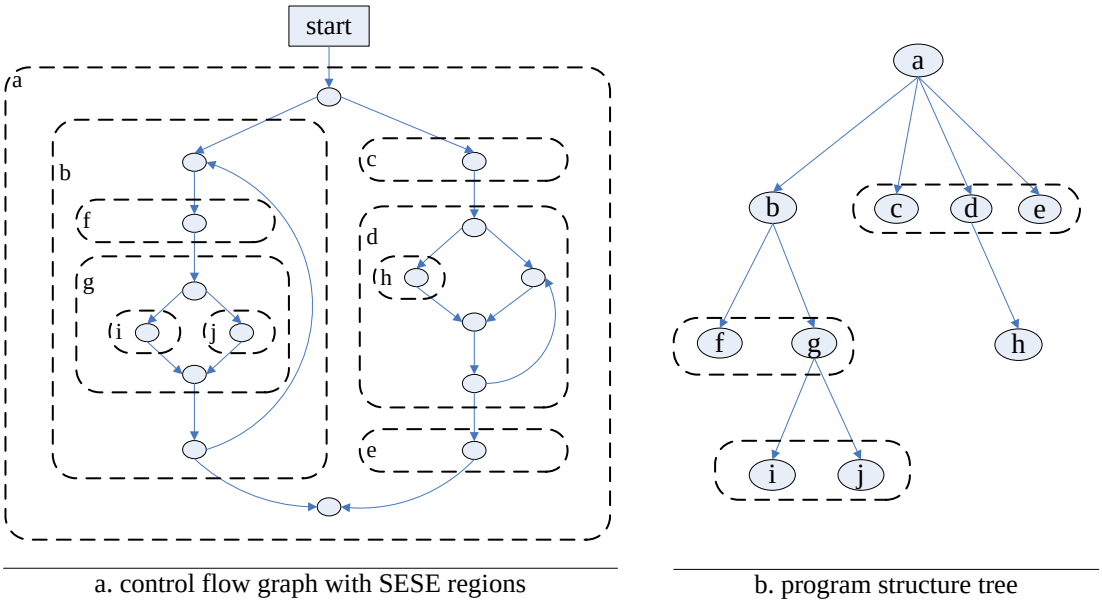


Figure 4: The program structure tree.

Definition 10 *A SESE region (a, b) is canonical if:*

1. *b dominates b′ for any SESE region* (a, b') *and*
2. *a postdominates a′ for any SESE region* (a', b).

In another way, for every edge e in the CFG, the canonical SESE region associated to e is the smallest region that uses e as entry or exit edge. Therefore, every edge only has one chance to be an entry or exit edge for a SESE region, which eliminates the possibility we enumerate two overlapped regions in the PST. The construction algorithm of PST is similar to the Tarjan interval finding algorithm [44]. We do not repeat it here and interested readers to refer to Johnson *et.al*'s original paper [25].

A first look at the definition of SESE region is strange. However, think about the CFG itself, which is of course a SESE region. Therefore, *PST is looking for self-similarity structures in the CFG*. More specifically, PST can enhance the performance of program analysis algorithms by exploiting the flow graph properties of global structure, local structure, and sparsity:

1. *Global structure*: As we said, each SESE region is a control flow graph in its own right. Therefore, any procedural analysis algorithm can be applied unchanged to each SESE region,

and the partial results can be combined using the PST to give the global result, resulting in a divide-and-conquer fashion algorithm.

2. *Local structure*: Algorithms for processing irreducible procedure is always slower than the reducible counterpart. However, irreducible procedures are rare and most of the places in the irreducible procedures are reducible. PST lets us to localize the effect of irreducible structure to several SESE regions. For reducible regions, we can apply reducible algorithms directly, which avoids using the slow irreducible algorithms globally.

3. *Sparsity*: Many analysis algorithms only manipulate a small subset of SESE regions. If the summarized transfer function for a SESE region is only an identity function w.r.t a data flow analysis, such transparent region can be directly bypassed, resulting in quick propagation of data flow facts. In fact, the sparse flow graph constructed in this fashion is called *quick propagation graph*, which is denser but easier to construct than Choi *et.al*'s *sparse evaluation graph* [34].

SSA construction is an application of PST that exploits the global structure and sparsity properties of the problem. ϕ-function placement for a variable x can be solved completely by analyzing only those regions that contain an assignment to x. More concretely, if a merge node needs a ϕ-function for variable x, this merge node must be in the iterated dominance frontier of some assignment to x in the same SESE region as the merge node. Therefore, we only mark those SESE regions that contain assignments to x. Then, we perform any SSA construction algorithm separately to these regions, the SSA representation for the whole procedure is constructed.

3.4 Program Dependence Graph

The **program dependence graph** (PDG) is a combination of the **control dependence graph** (CDG) and the **data dependence graph** (DDG) [14]. The DDG subgraph is exactly the graphical representation of the def-use chains. The CDG subgraph, informally, answers the question that under what branch conditions, a statement will be executed. Formally, the CDG is a graphical representation of the control dependence relations defined in Section 2. A sample PDG is given in Figure 5, where the rounded rectangle represents a control statement (*e.g.* *if* statement), the circle is a non-control statement, and the hexagon is a region node.

The **region node** summarizes the set of control dependence relations for its children

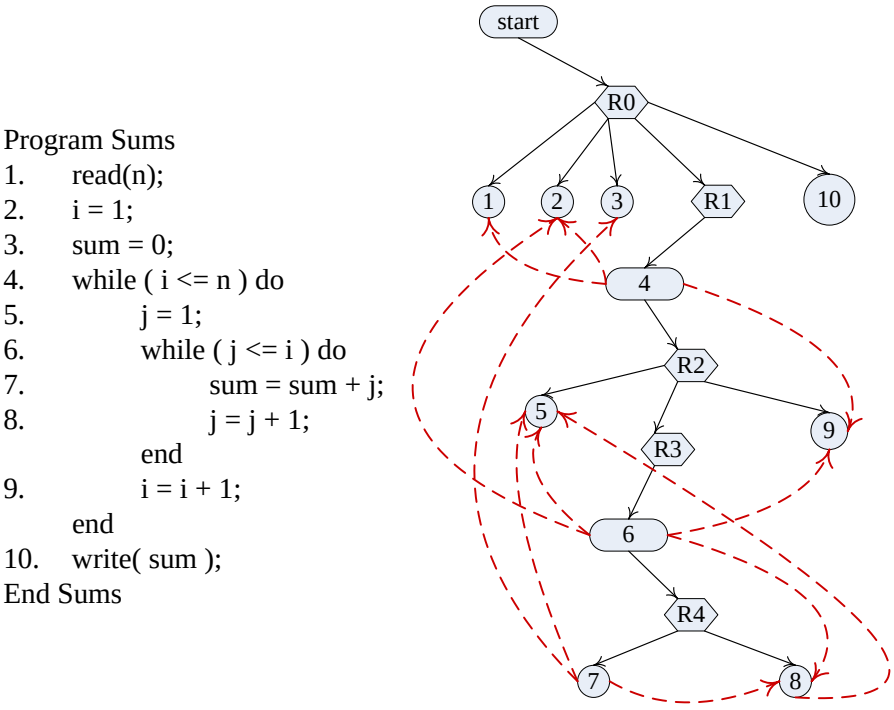


Figure 5: The program dependence graph. Solid arrow is control dependence, dotted arrow is data dependence.

statements (the circles) of the region node. We call the circles under the same region node **control equivalent statements**. Note that, the control equivalent statements may not be consecutive in the program. For example, in Figure 5, the statements 5 and 9 are separated by a *while loop*. However, they are placed under the same region node R2.

With the region nodes, it is clear that the CDG subgraph is another program structure tree that partitions the program into components: the children of a region, both the normal statements and region nodes, are threaded in originally program. Intuitively, these children form a super basic block that executes only once, from the first statement to the last one. For example, under the region $R2$, if we treat the inner *while* loop represented by $R3$ as a super statement, the statements 5 – 9 are executed one by one from top down. The CDG decomposition of program is very similar to the interval decomposition of program, which is, therefore, very suitable for elimination based data flow analysis [36].

However, performing iterative data flow analysis on PDG is harder than on CFG because

the execution flow between two statements is lost [1]. However, since PDG has a DDG subgraph, performing sparse analysis for non-distributive data flow problems [2] on PDG is easy. However, PDG is not suitable for the distributive problems because these problems require the decompression of the sparse analysis result.

One feature of PDG is that it makes backward slicing very simple [45]: Slicing is as easy as performing a backward reachability analysis on the PDG, walking along both the control dependence and data dependence edges. An extension of PDG is the **system dependence graph** (SDG) that integrates the interprocedural side-effects into consideration [23]. Therefore, we can directly perform the interprocedural backward slicing with SDG.

4 Single Static Assignment Form and Its Usage

The Single Static Assignment (SSA) intermediate representation (IR) gains popularity in recent years because, it greatly simplifies the design and improves the speed of many program analysis algorithms. Transforming an arbitrary program to an SSA program has two steps: The first is inserting ϕ-functions into proper positions in the program to ensure that every use of a variable x has only one reaching definition. We call this property *single reaching definition*. The second is renaming every assignment $x = \cdots$ to a new name $x_i = \cdots$ and changing $\cdots = x$ that uses x_i to $\cdots = x_i$. The term "static" means the we only give a new name to a variable for every assignment statement but not for every possible value. For example, whatever the value x would be after the assignment $x = y + 2$, we only give a new name x_1 to x and change the assignment to $x_1 = y + 2$. This single name x_1 represents all possible values 3, 4, 5, $\cdots$, for variable x in real execution.

Since every assignment introduces a new name to a variable, all the variables are **referential transparent**, *i.e.* we can locate a variable globally without name confusion. This feature lets us easily disambiguate two expressions with different values and implements the common subexpression removal optimization. For example, in the code of Table 1, we can easily figure out the two appearances of $x + 1$ expression compute different values. Because, in SSA form, they are syntactically different.

[1] Solely from PDG, we do not know statement 5 is executed before statement 9 or not.

[2] Non-distributive data flow problem has monotone but non-distributive transfer functions, such as constant propagation, points-to analysis.

$x = 1$	$x_1 = 1$
$y = x + 3$	$y_1 = x_1 + 3$
$x = 2$	$x_2 = 2$
$y = x + 2$	$y_2 = x_2 + 2$

Table 1: Original code fragment (left) and its SSA form (right).

The ϕ functions for a set S of definition statements are placed in the nodes $IDF(S)$. IDF can be computed by Cytron *et.al*'s algorithm [12] or the DJ graph based algorithm. SSA constructed by filling every ϕ function is called **minimal SSA**. The term *minimal* indicates that we insert least ϕ-functions into the program to ensure the single reaching definition property. However, the minimal SSA also contains many *useless phi*-functions, which define never used new variables. We can eliminate these dead *phi*-functions through liveness analysis [12]. The minimal SSA after the dead *phi* removal is called **pruned SSA**. Of course, other than first constructing minimal SSA, we can directly construct the pruned SSA [10].

The third flavor, other than the minimal SSA and the pruned SSA, is the **semi-pruned SSA** [6], which only eliminates the ϕ-function defined variables that are used wholly in the same basic block with its definition. The amount of such *local* names is huge because compilers often generate temporary names to hold intermediate steps in a non-trivial computation. From the experiment we know that the semi-pruned SSA eliminates 80% dead ϕ-functions of those removed by pruned SSA. Identifying the variables referenced across the basic block boundary is simply a linear scan over the statements, which is more efficient and simpler than implementing a liveness analysis.

The importance of SSA is that it enables efficient program analysis or greatly simplifies the analysis algorithm design. Specifically, SSA provides the opportunity to perform *sparse* data flow analysis. In the compiler backend, SSA can help achieve better register allocation. Next, we introduce them respectively.

4.1 Sparse Data Flow Analysis

Sparse analysis is a data flow information compaction and analysis statements minimization

strategy [29]. Informally, for a variable x, sparse analysis only considers the definition and use sites of x, and ignores analyzing other statements that do not influence the value of x. Traditional data flow analysis associates information to pairs formed by a variable and a program point. This results in inefficiency because the information bound to the same variable at a set of different program points may be the same (those points do not change the information). In this case, redundant information is stored and evaluated multiple times, which is a waste of both space and time.

However, in SSA form, the data flow information for variable x in its live range is never changed. This is because the live range for a x is those program points dominated by the definition point of x, and, these points can only see this definition due to SSA semantics. Therefore, there is no statement in the live range can change the information of x. Due to this reason, we do not need to associate the information to variable and program point pair. Instead, we only associate information to the variable and we decompress the information for every program points when queried. This kind of redundancy removal in fact saves quite a lot time and memory. For example, Hardekopf *et.al* observe 175x speedup for pointer analysis [19].

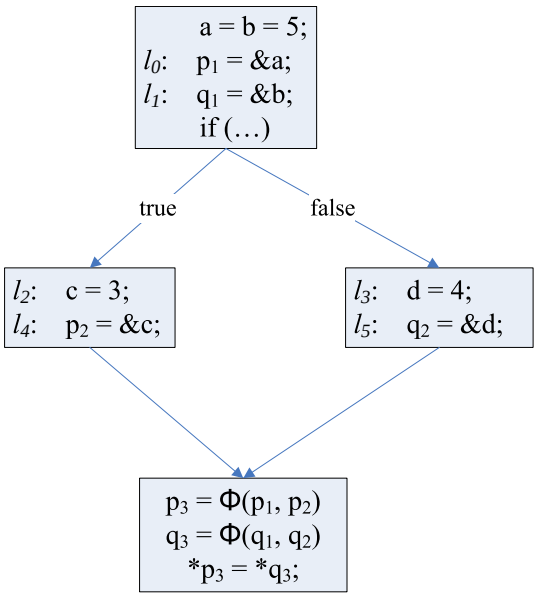


Figure 6: Example of sparse pointer analysis.

Another nice feature of SSA based sparse analysis is that flow insensitive analysis automatically obtains certain degree of flow sensitivity. It gains flow sensitivity because the information obtained at different definition sites is kept separated. We say certain degree because the ϕ-function is place at the beginning of a basic block, it merges information before the subsequent statements use it. For the non-distributive data flow problem such as constant propagation, merging before use would degrade precision [28].

We use a pointer analysis to illustrate the two features mentioned above. The code is shown in Figure 6. The points-to information of p_1 at the points l_3 and l_5 is the same as l_0, because those places do not change the points-to value of p_1. This is the same situation to

q_1 at l_2 and l_4. Therefore, we only maintain points-to information for p_1 and q_1 at l_0 and l_1. Since at l_3, the only visible definition of variable p and q are p_1 and q_1, we can easily deduce that p points-to a at l_3 at query time.

The precision degeneration comes from the two ϕ functions. In pointer analysis, the ϕ function merges the points-to information for its parameter variables. Therefore, p_3 points to a and c, q_3 points to b and d. The assignment $*p_3 = *q_3$ becomes four assignments: $a = b$, $a = d$, $c = b$, $c = d$. However, the assignments $a = b$ and $c = d$ are fake. In traditional data flow analysis, we analyze the *true* and *false* branches separately and we will not suffer from the spuriousness. Because, only $p_3 = c$ and $q_3 = b$ co-exist in the *true* branch, $p_3 = a$ and $q_3 = d$ co-exist in the *false* branch. Of course, the reason for spuriousness in SSA is that the transfer function of pointer analysis is non-distributive.

4.2 Register Allocation

Register allocation is a big topic that we only give a overview here. Interested readers can read Pereira's survey [31] for more details, which is also our major reference.

4.2.1 Traditional Register Allocation

Register allocation is the problem of mapping program variables to either machine registers or memory addresses. Since the number of registers is limited, not all the variables can be assigned to registers at any time. Therefore, resolving the *live range interferences* is the core topic for a register allocator. Specifically, the collection of program points where a variable is alive is called its **live range**. Two variables **interfere** if the intersection of their live ranges is non-empty. In this case, we also say that their live ranges overlap. We can safely assign two variables that do not interfere to the same register.

Due to the number limitation of registers, there may exist some program points, a register value must be written back to the main memory and that register is assigned to other variables to use. This operation is called **spilling**. Since memory store and load are expensive, hence, a register allocator should minimize the spilled variables. A spilling is always preceded by **live range splitting**, which divides the live ranges of variables by adding copies to the program and renaming the variables. For example, if we try to split x at point P, we insert a copy $x\prime = x$ and rename all the subsequent use of x to $x\prime$. After the splitting, spilling the original x becomes spilling the current version of x or $x\prime$, both require placing

less store/load statements. The inverse operation of live range splitting is **coalescing**, which is used eliminate useless copies. Specifically, if two variables v_1 and v_2 do not interfere, and they are related by a copy instruction $v_1 = v_2$, it is desirable that these variables should be allocated to the same register.

The most used approach for register allocation is based on graph coloring. The first step is building an **interference graph**. That is, given a program, its interference graph $G = (V, E)$ contains a vertex for each variable v . A undirected edge (u, v) is in E if u and v do not interfere. Thus, the problem of assigning registers to variables can thus be approximated by coloring the interference graph, where each color corresponds to a register.

The most cited graph coloring based approach is described by Chaitin [9]. This is an iterative algorithm that performs graph coloring, spilling, and coalescing interchangeably, until the interference graph is K-colorable, where K is the number of registers. Specifically, this algorithm executes the following operations [7]:

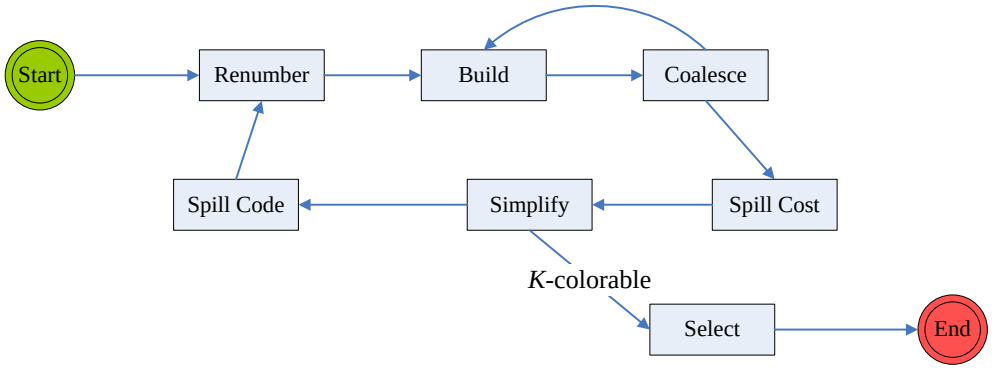


Figure 7: Chaintin's register allocator.

1. **Renumber**: discover live range information in the source program.
2. **Build**: build the interference graph.
3. **Coalesce**: merge the live ranges of non-interfering variables related by copy instructions.
4. **Spill cost**: estimate the spill cost of each live range. The cost is computed by computing the number of loads and stores that would be required to spill the live range, with each live range weighted by $c \times 10^d$, where c is the operation's cost on the target architecture and d is the instruction's loop nesting depth.

5. **Simplify**: constructs an ordering of the nodes. It uses an stack and repeats the following two steps until the graph is empty:

- If there exits a node x with degree less than K, choose x. Otherwise, choose a node x with smallest estimated spill cost for spill. Mark x for spill later;
- Remove x and all its edges from the graph. Push x onto the stack.

6. **Spill Code**: spill the marked nodes. Each spilled live range is converted to a collection of tiny ranges by inserting loads before uses and stores after definitions.

7. **Select**: assign colors to nodes.

Figure 7 gives a flowchart of Chaintin's algorithm. This algorithm has to be iterative because Chaintin shows that spill free register allocation is NP-Complete. However, if the input program is not so general, perhaps we can have more efficient and effective register allocation algorithm. Fortunately, the SSA program is one of such special form.

4.2.2 SSA based Register Allocation

The specialty of SSA program is that its interference graph is a *chordal graph*. A graph is **chordal** if every cycle with four or more edges has a chord, that is, an edge which is not part of the cycle but which connects two vertices on the cycle. You can say a chordal graph is triangulated. Chordal graph is a kind of perfect graph, which can be colored optimally in polynomial time. More concretely, coloring a chordal graph is in $O(E + V)$ time.

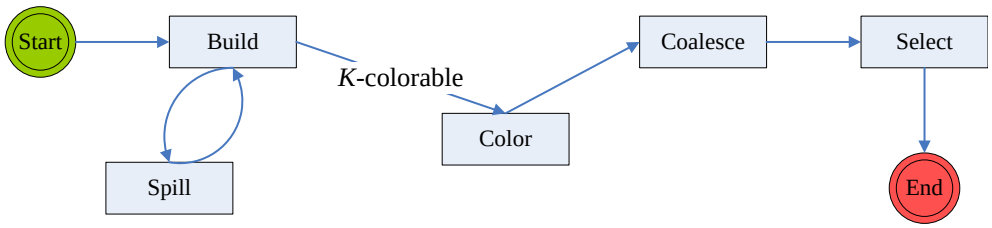


Figure 8: SSA based Chaintin's register allocator.

Not only the graph coloring process is much faster, the large iteration cycle involving spilling, coloring, and coalescing is also simplified. This is because we can determine the minimal number of colors for coloring the chordal graph in linear time. Therefore, we spill

code until the graph is K-colorable. Then, we coalesce and color the spilled code, and back translate the SSA program to original program. This process is depicted in Figure 8.

4.3 Translation Out of SSA

The ϕ-function is supported only by a few processors, hence, after the code optimization on SSA, we should eliminate the ϕ-functions in order to generate the assembly code. The first solution proposed in Cytron *et.al*'s seminal paper [12] is: A k-input ϕ-function at entrance of a node X is replaced by k ordinary assignments, one at the end of each control flow predecessor of X. Then, replacing the names x_1, x_2, $\cdots$, x_k to a single name x.

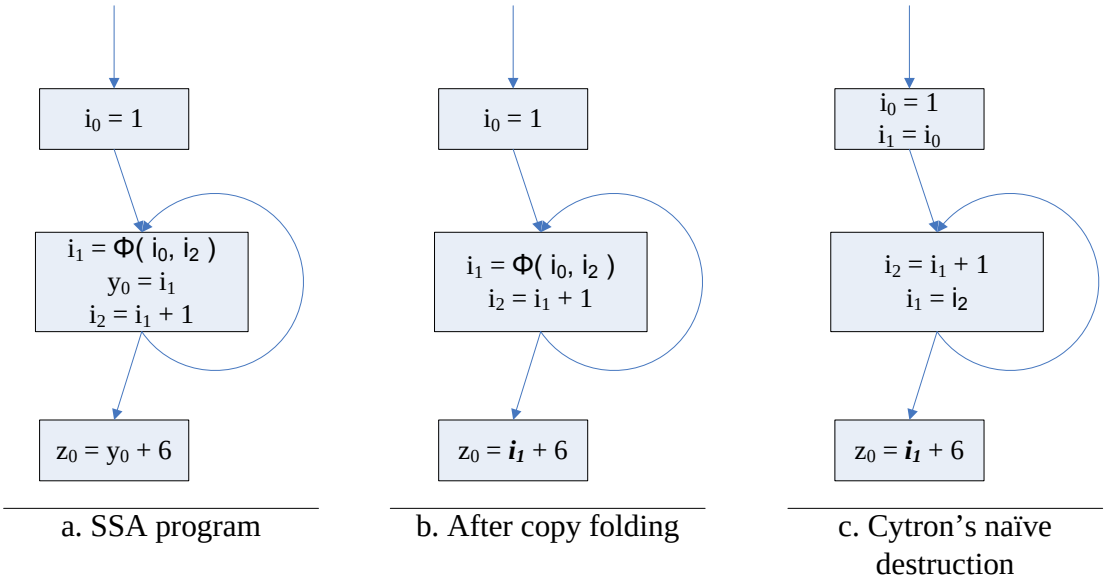


Figure 9: The lost copy problem. In picture (c), z_0 received wrong value.

Some of the inserted assignments are redundant, and part of them are removed by coalescing in register allocation. The most important problem is that this simple translation algorithm is incorrect. Briggs *et.al* point out two subtle errors [6]. The two errors are called *lost copy problem* and *swap problem*, which are illustrated in Figure 9 and Figure 10.

Although Briggs *et.al* give the remedy, their algorithm is very complex that includes liveness analysis and a preorder walk over the dominator tree. More importantly, they do not show their new algorithm is correct, they only say they cure the problems they encountered.

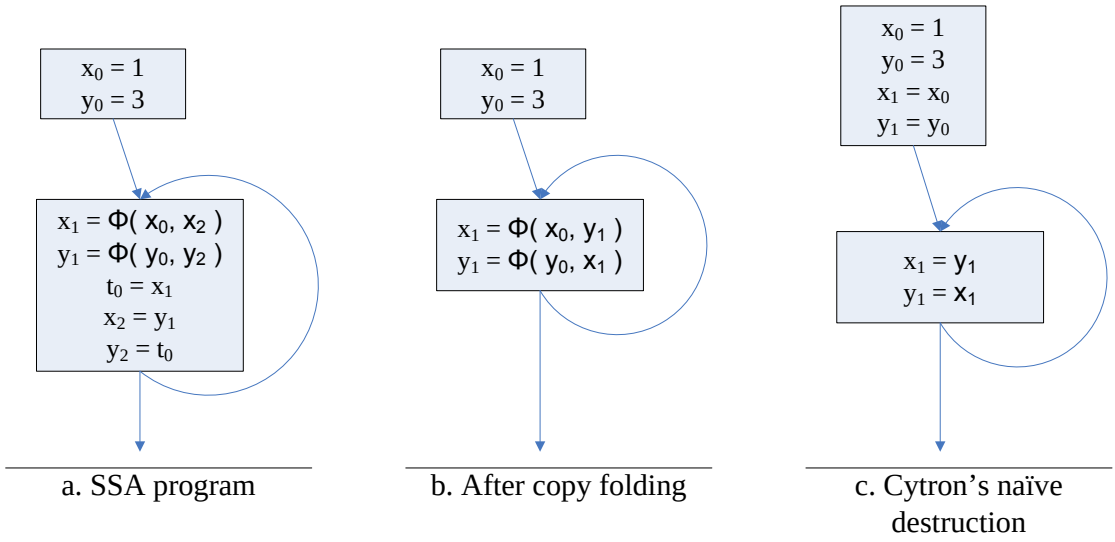


Figure 10: The swap problem. In picture (c), $x_1 = y_1 = 3$.

The first both simple and correct solution is given by Screedhar *et.al* [38]. Their core idea is *treating the translation out of SSA problem as a coalescing problem.* The observation is that replacing the names x_1, x_2, $\cdots$, x_n to a single name x is very similar to assigning the register x to these names, in the context of register allocation. As we learned from the register allocation chapter, two variables can be coalesced iff their live ranges are disjoint. Reconsider the lost copy problem and the swap problem, the errors occur because the live ranges of the parameters of the ϕ-function have non-empty intersection.

Knowing the reason of the naive algorithm, Screedhar *et.al* give a simple solution. Consider a ϕ-function $x_0 = \phi(x_1, \cdots, x_n)$ placed at entry of a block B_0: x_0 takes the value of x_i if the control flow comes from the i^{th} predecessor block of B_0. If x_0, $\cdots$, x_k can be given the same name without changing semantics of the program, the ϕ-function can be eliminated. When this property is true for all the ϕ-functions in the program, the SSA form is called conventional (CSSA) [38]. The conventional property may be invalided after code optimization, especially the copy folding and code motion. The naive solution introduced by Cytron *et.al* only work correctly for the CSSA program. Screedhar *et.al*'s method tries to recover the conventional property as follows:

1. It creates $n + 1$ new variables a_0, a_1, $\cdots$, a_n;
2. A copy $a_i = x_i$ is placed at the end of B_i, the i^{th} predecessor block of B_0;

3. The ϕ-function $x_0 = \phi(x_1, \cdots, x_n)$ is replaced by $a_0 = \phi(a_1, \cdots, a_n)$;

4. A copy $x_0 = a_0$ is placed at entry of the block B_0 (after ϕ-functions).

Now, folding the variables $a_0, \cdots, a_n$ can eliminate all the ϕ functions. Apparently, this algorithm (Method I called by Screedhar *et.al*) is simple to implement. However, it introduces many copies that may significantly slowdown the program. Method II is proposed to remove some useless copies by coalescing the variables. Other than post-processing these copies that already take many resources to construct, Method III inserts the copies in Step 2 of Method I on-the-fly based on the live range interference information. Screedhar *et.al* conjecture that the number of inserted copies in Method III is minimal [38]. However, Boissinot *et.al* figure out Screedhar *et.al*'s Method III does not produce minimal copies and, Method III may not work correctly for embeded software where branch instruction can define variables [5]. By this reason, Boissinot *et.al*'s approach is built on Method I to guarantee the correctness, and improve Method I in efficiency and effectiveness by:

1. Avoiding the interference graph and developing a lightweight query structure for liveness checks;

2. Testing variable classes congruence in linear time rather than quadratic time;

3. Developing more general coalescing mechanism for more aggressive coalescing.

5 Extentions to SSA

During the development of SSA, a lot of SSA variants are introduced to extend the capability of SSA. The most important extensions are aliasing annotations for handling indirect memory side effects (HSSA), and the variable use annotations for enabling both forward and backward sparse analysis. We will go through these extensions in this section.

5.1 HSSA: Resolving Aliasing Information

The original SSA is proposed for Fortran programs. Hence, it cannot handle aliased memory access due to memory overlap such as the "union" structure in C and pointer induced aliasing. We can treat the names representing overlap memory addresses, such as the fields members in a union structure, as a large monolithic variable to take account the memory effects induced by writing to these different names. For pointer based indirect access, one way is using SSA

only for top level variables, *i.e.*, the variables that cannot be modified through pointers. The produced IR is called **partial SSA**, which is adopted by LLVM compiler. Another way is using pointer analysis to resolve the pointers, then we instrument the program with side effects annotations before inserting ϕ-functions.

The most popular way to annotate the pointer induced side effects is developed by Chow *et.al* [11]. They call it **Hashed SSA** (HSSA) form. HSSA distinguishes the pointer induced scalar variable definition as **MayDef** and annotates its side effects with χ-function. The semantics of χ is that it potentially defines a variable, which means the reaching definition of the potentially defined variable should be preserved. For example, a statement $*p = 3$ where p points to variable a is annotated as $a = \chi(a)$. The annotation is put *immediately after* the statement $*p = 3$. For an indirectly use of a variable such as $x = *q$, where q points to b, an annotation $\mu(b)$ is placed *immediately before* the statement to describe the **MayUse** information. The χ and μ functions can also be used to annotate the interprocedural side effects at the callsites in the same way.

The heap variable can be processed similarly as the scalar variable. In Chow *et.al*'s original paper, it defines the concept *virtual variable*, which is a representative conceptual variable of a group of heap variables. The virtual variable has the same aliasing effects as the its group members, which means, every indirect write to a group member, the algorithm annotates the write effects to its virtual variable. Therefore, we can recognize the virtual variable as to aggregate the indirect effects of its members. The use of virtual variable of course degrades the precision of the subsequent algorithms based on HSSA. But simultaneously, it greatly reduces the number of annotations.

After the annotation, the program can be processed by standard SSA transformation algorithm without loss of information. However, the resulting SSA program may be terribly inefficient because the explosion of the χ-functions and the chain reactions to the ϕ-functions blow up. To reduce the number of new variable versions created, Chow *et.al* introduce the **zero version** variable to represent a set of variables, similar to using the virtual variable for a group of heap variables. Specifically, they first define the occurrences of variables in the original program before conversion to SSA form *real occurrences*. Thus, the variable occurrences in ϕ, μ, and χ are not real occurrences. Based on the real occurrence concept, the zero versions are defined recursively as follows:

1. The left hand side of a χ is zero version if it has no real occurrence.
2. If an operand of a ϕ is zero version, the result of the ϕ is zero version if it has no real

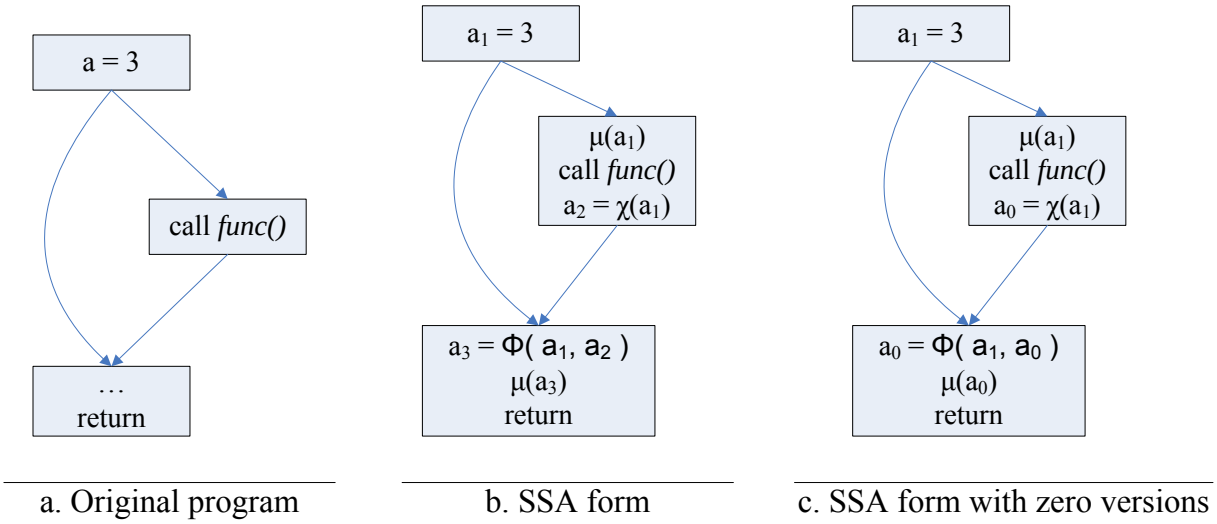


Figure 11: Example of using zero versions.

occurrence.

From the definition we know, zero versioning characterizes versions with no real occurrences and their values are affected by aliased stores. A sample zero versioned SSA is illustrated in Figure 11. After the zero versioning, Chow *et.al* also perform a global value numbering (GVN) to identify equivalent memory expressions, hence they further compact the SSA program. Due to the use of hashing in the GVN, the resulting SSA form is called HSSA.

The application of HSSA form is broad, especially in the defects analysis field, such as Memory leak detection [43] and security checking [27]. In modern design, zero versioning and GVN optimizations usually are not performed, resulting least compactness but best precision for subsequent use.

5.2 SSI: Splitting All Information Definition Points

SSA only splits the information generated by explicit assignments. However, variable use can also be an information source. An example is the classical backward data flow analysis, liveness analysis, the variable use sites are collected and propagated to determine the liveness range of each variable. Another case is the predicate expression such as *if* $(p > 0)$, which implies the value range information ($p > 0$ and $p \leq 0$) that can be used in analyzing

corresponding *true* and *false* branch code. The predicate information is very important for path sensitive defect analysis such as array out of bounds checking [3]. Based on these requirements, Ananian *et.al* develop the **static single information** (SSI) form [2].

Building SSI form involves adding pseudo assignments for a variable v:

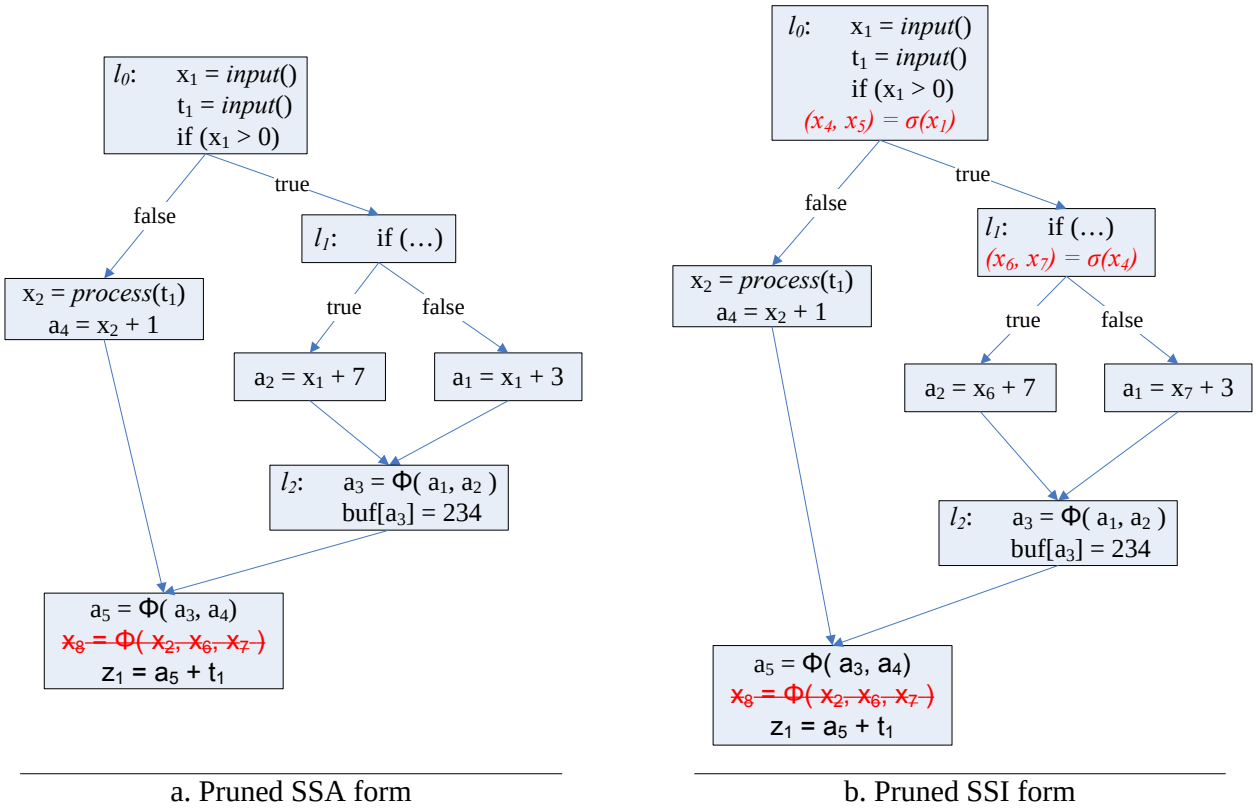
1. ϕ: at control flow merges, exactly the SSA ϕ-function;

2. σ: at locations where control flow splits and at least one of the disjoint paths from the split uses the value of v.

Constructing the SSI form is not as easy as placing the ϕ and σ functions independently. Since they mutually interfere, a fixed point computation is required. Recently, the construction algorithm proposed by the originator of SSI, Ananian *et.al* [2], is proven to be wrong [4]. Fortunately, Singer gives a correct algorithm in his PhD thesis [37]. Singer's algorithm to place σ-functions is in the same shape as Cytron *et.al*'s algorithm to place the ϕ-functions: It first computes the *iterative post-dominance frontier* (IPDF) set, seeded by the concerned variable use sites. IPDF is a parallel concept to the iterative dominance frontier (IDF) mentioned in Section 2. Then, a σ-function is placed at each member of the IPDF set. The SSI construction algorithm is simply iteratively placing the σ and ϕ functions until convergence. The resulting SSI form is called *miniaml* SSI.

However, from Singer's empirical study we find that, σ functions will significantly increase program size: the SSI form has 6 times more pseudo assignments as the corresponding SSA form [37]. Because of its large size, the *pruned* or *semi-pruned* SSI from, which are defined similarly to their SSA counterparts, are always constructed in practice. Of course, the key point is to use liveness analysis to eliminate unnecessary pseudo assignments. Figure 12 shows the pruned SSA and corresponding pruned SSI forms.

SSI has two important features. The first is enabling sparse analysis for backward data flow problems. This is unachievable by SSA and confirmed by a number of researchers [10, 26, 46]. The reason is that *the information of unanalyzed basic blocks cannot be recovered in SSA form.* We illustrate this problem through the *very busy variable* analysis. A variable v is very busy at a program point P iff it is always subsequently used before killed. Let us first consider the SSA program in Figure 12 (a), we want to know if x_1 is very busy at statement l_1? From the sparse analysis, we can conclude that x_1 is not very busy at its definition point l_0, because x_2 kills x_1 before use. However, from the information at l_0, we cannot decide the busy information at l_1. This because the busy information is propagated backwardly from l_1 to l_0, it is changed along the way from l_1 to l_0. As we discussed, sparsity



a. Pruned SSA form

b. Pruned SSI form

Figure 12: Example of pruned SSA and pruned SSI form. The ϕ-function for x_8 is deleted because it is no longer live after its definition.

is a compression strategy that the program points dominated by point P all have the same information with P. However, in our case, l_1 is dominated by l_0 but l_1 has different flow information with l_0, this is a violation of the sparsity requirement.

The second feature of SSI is it can integrate the predicate information into sparse analysis. In Figure 12 (b), the predicate $x_1 < 4$ yields the opportunity to obtain that $a_3 < 11$ at location l_2, which in turn helps us check if the visit of array $buf[a_3]$ is out of boundary. If the predicate is $x_1 == 0$, we can even use it for constant propagation because now we know $a_1 = 3$ and $a_2 = 7$. More importantly, this more powerful version constant propagation still stays in sparse form.

Last, SSI again simplifies the register allocation algorithm, because the interference graph build in SSI form is *interval graph* [4], a more restricted type of graph than chordal graph.

Based on interval graph, the maximum K-colorable subgraph problem can be solved in polynomial time. Therefore, we can choose the least variables to spill and generate a K-colorable graph, which is still a NP-Complete problem in chordal graph. The second benefit is, with SSI, the linear scan register allocator never face to the *lifetime holes* problem. More details about linear scan algorithm can be found in Pereira's survey [31], which is out the scope of this survey.

6 Conclusion

In this survey, we examine several important program representations both in graphical form and linear form, which we believe, are very helpful for program analysis researchers. Of course, we still uncover some useful representations here, but our survey is already comprehensive. We will keep updating this survey in the future.

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers principles, techniques, and tools*. Addison-Wesley, Reading, MA, 1986.

[2] C. Scott Ananian and Martin Rinard. Static single information form. Technical report, Master's thesis, Massachussets Institute of Technology, 1999.

[3] Rastislav Bodík, Rajiv Gupta, and Vivek Sarkar. Abcd: eliminating array bounds checks on demand. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 321–333, New York, NY, USA, 2000. ACM.

[4] Benoit Boissinot, Philip Brisk, Alain Darte, and Fabrice Rastello. Ssi properties revisited. *ACM Trans. Embed. Comput. Syst.*, 11S(1):21:1–21:23, June 2012.

[5] Benoit Boissinot, Alain Darte, Fabrice Rastello, Benoit Dupont de Dinechin, and Christophe Guillon. Revisiting out-of-ssa translation for correctness, code quality and efficiency. In *Proceedings of the 7th annual IEEE/ACM International Symposium on*

Code Generation and Optimization, CGO '09, pages 114–125, Washington, DC, USA, 2009. IEEE Computer Society.

[6] Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. Practical improvements to the construction and destruction of static single assignment form. *SOFTWARE – PRACTICE AND EXPERIENCE*, 1998.

[7] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Trans. Program. Lang. Syst.*, 16(3):428–455, May 1994.

[8] Silvian Calman and Jianwen Zhu. Increasing the scope and resolution of interprocedural static single assignment. In *Proceedings of the 16th International Symposium on Static Analysis*, SAS '09, pages 154–170, Berlin, Heidelberg, 2009. Springer-Verlag.

[9] G. J. Chaitin. Register allocation & spilling via graph coloring. In *Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, SIGPLAN '82, pages 98–105, New York, NY, USA, 1982. ACM.

[10] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '91, pages 55–66, New York, NY, USA, 1991. ACM.

[11] Fred C. Chow, Sun Chan, Shin-Ming Liu, Raymond Lo, and Mark Streich. Effective representation of aliases and indirect memory operations in ssa form. In *Proceedings of the 6th International Conference on Compiler Construction*, CC '96, pages 253–267, London, UK, UK, 1996. Springer-Verlag.

[12] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991.

[13] Dibyendu Das, B. Dupont De Dinechin, and Ramakrishna Upadrasta. Efficient liveness computation using merge sets and dj-graphs. *ACM Trans. Archit. Code Optim.*, 8(4):27:1–27:18, January 2012.

[14] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[15] Mark Gabel, Lingxiao Jiang, and Zhendong Su. Scalable detection of semantic clones. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 321–330, New York, NY, USA, 2008. ACM.

[16] Harold N. Gabow. Data structures for weighted matching and nearest common ancestors with linking. In *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, SODA '90, pages 434–443, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics.

[17] Rajiv Gupta and Mary Lou Soffa. Employing static information in the generation of test cases. *Softw. Test., Verif. Reliab.*, 3(1):29–48, 1993.

[18] Ben Hardekopf and Calvin Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 290–299, New York, NY, USA, 2007. ACM.

[19] Ben Hardekopf and Calvin Lin. Semi-sparse flow-sensitive pointer analysis. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '09, pages 226–238, New York, NY, USA, 2009. ACM.

[20] Rebecca Hasti and Susan Horwitz. Using static single assignment form to improve flow-insensitive pointer analysis. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, pages 97–105, New York, NY, USA, 1998. ACM.

[21] Paul Havlak. Nesting of reducible and irreducible loops. *ACM Trans. Program. Lang. Syst.*, 19(4):557–567, July 1997.

[22] M. S. Hecht and J. D. Ullman. Characterizations of reducible flow graphs. *J. ACM*, 21(3):367–375, July 1974.

[23] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, PLDI '88, pages 35–46, New York, NY, USA, 1988. ACM.

[24] Susan Horwitz and Thomas Reps. The use of program dependence graphs in software engineering. In *Proceedings of the 14th international conference on Software engineering*, ICSE '92, pages 392–411, New York, NY, USA, 1992. ACM.

[25] Richard Johnson, David Pearson, and Keshav Pingali. The program structure tree: computing control regions in linear time. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, PLDI '94, pages 171–185, New York, NY, USA, 1994. ACM.

[26] Richard Johnson and Keshav Pingali. Dependence-based program analysis. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, PLDI '93, pages 78–89, New York, NY, USA, 1993. ACM.

[27] V. Benjamin Livshits and Monica S. Lam. Tracking pointers with path and context sensitivity for bug detection in c programs. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-11, pages 317–326, New York, NY, USA, 2003. ACM.

[28] Nomair A. Naeem, Ondřej Lhoták, and Jonathan Rodriguez. Practical extensions to the ifds algorithm. In *Proceedings of the 19th joint European conference on Theory and Practice of Software, international conference on Compiler Construction*, CC'10/ETAPS'10, pages 124–144, Berlin, Heidelberg, 2010. Springer-Verlag.

[29] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[30] Carl D. Offner. Notes on graph algorithms used in optimizing compilers. 2011.

[31] FMQ Pereira. a survey on register allocation. Technical report, UCLA, 2008.

[32] Keshav Pingali and Gianfranco Bilardi. Optimal control dependence computation and the roman chariots problem. *ACM Trans. Program. Lang. Syst.*, 19(3):462–491, May 1997.

[33] G. Ramalingam. On loops, dominators, and dominance frontier. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 233–241, New York, NY, USA, 2000. ACM.

[34] G. Ramalingam. On sparse evaluation representations. *Theor. Comput. Sci.*, 277(1-2):119–147, April 2002.

[35] G. Ramalingam and Thomas Reps. An incremental algorithm for maintaining the dominator tree of a reducible flowgraph. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '94, pages 287–296, New York, NY, USA, 1994. ACM.

[36] Barbara G. Ryder and Marvin C. Paull. Elimination algorithms for data flow analysis. *ACM Comput. Surv.*, 18(3):277–316, September 1986.

[37] Jeremy Singer. Static program analysis based on virtual register renaming. Technical report, University of Cambridge, 2006.

[38] Vugranam C. Sreedhar, Roy Dz ching Ju, David M. Gillies, and Vatsa Santhanam. Translating out of static single assignment form. In *In Static Analysis Symposium, Italy*, pages 194–210. Springer Verlag, 1999.

[39] Vugranam C. Sreedhar and Guang R. Gao. A linear time algorithm for placing &phgr;-nodes. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '95, pages 62–73, New York, NY, USA, 1995. ACM.

[40] Vugranam C. Sreedhar, Guang R. Gao, and Yong-Fong Lee. Identifying loops using dj graphs. *ACM Trans. Program. Lang. Syst.*, 18(6):649–658, November 1996.

[41] Vugranam C. Sreedhar, Guang R. Gao, and Yong-Fong Lee. A new framework for exhaustive and incremental data flow analysis using dj graphs. In *Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, PLDI '96, pages 278–290, New York, NY, USA, 1996. ACM.

[42] Bjarne Steensgaard. Sequentializing program dependence graphs for irreducible programs. Technical report, 1993.

[43] Yulei Sui, Ding Ye, and Jingling Xue. Static memory leak detection using full-sparse value-flow analysis. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 254–264, New York, NY, USA, 2012. ACM.

[44] Robert Tarjan. Testing flow graph reducibility. In *Proceedings of the fifth annual ACM symposium on Theory of computing*, STOC '73, pages 96–107, New York, NY, USA, 1973. ACM.

[45] Frank Tip. A survey of program slicing techniques. Technical report, IBM T.J. Watson, Amsterdam, The Netherlands, The Netherlands, 1994.

[46] Daniel Weise, Roger F. Crew, Michael Ernst, and Bjarne Steensgaard. Value dependence graphs: representation without taxation. In *Proceedings of the 21st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '94, pages 297–310, New York, NY, USA, 1994. ACM.

[47] Michael Wolfe. Beyond induction variables. In *Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, PLDI '92, pages 162–174, New York, NY, USA, 1992. ACM.