École normale supérieure de Lyon
Laboratoire de l'Informatique du Parallélisme
École Doctorale Informatique et Mathematiques

Doctorat

Informatique

Benoit BOISSINOT

Towards an SSA based compiler back-end: some interesting properties of SSA and its extensions

Thèse dirigée par Fabrice RASTELLO.

Soutenue le 30 septembre 2010.

Jury:

Albert	Cohen	(Rapporteur)
Anton	Ertl	(Examinateur)
David	Monniaux	(Rapporteur)
Fabrice	Rastello	(Directeur)
Yves	Robert	(Examinateur)

Contents

Chapter 1

Introduction

Compilation of programs, that is the transformation of human generated source code into binary code executable by a processor has always been a very active topic in computer science. First the goal was to get rid of the burden of hand-compiling, letting programs themselves take care of this tedious task. The initial focus was purely on getting a correct translation from textual source into binary code: an important area of research was lexing (separating the input into tokens) and parsing (interpreting the tokens). The automation brought by the compilers can be used to optimize programs too: detecting non-optimal structure, removing unneeded code and in general generating the best possible assembly for a given source code.

One of the recent trend in the compiler community is the area of virtualization. Virtualizing the program, for example by using a machine-independent bytecode to represent the compiled program, helps in several areas: portability between architectures, security, productivity, ... While this process has been used for a long time in the desktop and server computing field, it has only since recently been brought forward on embedded platforms. Indeed, the diversity of the architectures is a nightmare when it comes to distributing the executable, shipping a single binary that is compiled on the host processor (ahead-of-time or just-in-time compilation) helps solve this problem. But as the host processor has limited resources, we need to design compiler algorithms which aggressively optimize but still run relatively fast and consume little memory.

Our goal is to split the compilation process, a first architecture-independent pass compiles the source code into bytecode, optimizing as much as possible. A second pass, on the host processor, then compiles the bytecode down to the

instructions available for the specific architecture. This second pass is severely constrained by the computing power available on the embedded platform, this is the part our thesis improves by bringing new fast algorithms for crucial parts of the compiling process.

All along this process, several intermediate representation of the code can be used. One of them is the Static Single Assignment (SSA) form, which facilitates some optimizations or analyses.

Under the SSA form, the program is transformed into a semantically equivalent program where every variable is defined only once textually. Currently, the majority of compilers use the SSA form: LLVM, GCC, Open64, ...

Our work in this thesis is articulated around three contributions.

In the first chapter, we define the ground material used in the thesis. First we define the data structure used to represent the program, the control flow graph, furthermore we describe some useful graph properties. Then, in the context of control flow graphs, we give the definition of loops, one of the most essential structure of a computer program. Finally we define the SSA form, which is the intermediate representation at the center of this thesis.

When compiling on a resource-constrained platform, the compiling process is simplified as much as possible. Liveness analysis then becomes one of the costliest analyses found during code generation. The second chapter presents our fast algorithm for the liveness analysis suitable for programs in SSA form. We present several approaches, some based on path exploration, and our approach based on the structure of the loops. Since our algorithm only exploits the shape of the control flow graph, it does not require any complex re-computation after small modifications of the program (for example due to an optimization).

We then present a variant of the SSA form, the Static Single Information (SSI) form. In a first part, we clarify the various definitions found in the literature, and show the differences between them. Then we prove that the intersection graph of the live-ranges of variables under SSI form is an interval graph. Additionally we exhibit an order of the control flow graph such that every live-range is an interval of the program.

Our last contribution is a clean way to go out of the SSA form. The SSA form introduces ϕ-functions which do not map into generated code. In order to produce machine code, those special instructions need to be removed, by introducing new copies, as few as possible. We present a clean process for this transformation, clearly separating the different phases of the algorithm: first the introduction of new copies in order to reach a variant of the SSA

form where the ϕ-functions can be naturally removed, and then minimize the number of copies, while keeping the code under this SSA form variant. This approach allows us to get a provably correct transformation, contrary to some of the previous approaches. Furthermore, while simpler to implement, our approach can achieve results comparable to some of the complex previous techniques.

Chapter 2

Control flow graph, loops, and SSA

Throughout this thesis, we will use different concepts and properties. In this chapter, we lay out the groundwork needed for our contributions. We first describe the internal representation of the compiler we will manipulate all along, a graph based representation of the program: the control flow graph. Based on this representation, we define several graph relations: dominance and post-dominance, additionally we state several useful theorems related to those relations. In the context of the control flow graph, we then give a general definition of loops, later on we define our own more specific definition that will prove useful in the following chapters. Finally, we define the static single assignment form, a commonly used intermediate representation.

Since we only define the structures and properties which are useful for this thesis, the interested reader can find an in-depth introduction to this topics in most compiler textbooks. In particular, we recommend the *Modern Compiler Implementation* books [3] by A. Appel and J. Palsberg.

2.1 Control flow graph (CFG)

2.1.1 Definitions and properties

Control-flow graph

A procedure is represented as a **control-flow graph** (CFG), which is a directed graph $G = (V, E, r, t)$, with set of nodes V, set of edges E, and two

specific nodes r and t: r is the **entry** node, with no incoming edge, and t is the **exit** node, with no outgoing edge.

A **path** P of length $k \geq 0$ from a node u to a node v in V is a non-empty sequence $(v_0, v_1, \ldots, v_k)$ of nodes such that $u = v_0$, $v = v_k$, and $(v_{i-1}, v_i) \in E$ for $i \in [1..k]$. With this definition, a path of length 0 is a path with one node and no edge. If the CFG contains a self-edge, i.e., an edge of the form (u, u), then there is also a path of length 1 from u to itself. Node v is **reachable** from u if there is a path from u to v in the CFG. A node u part of a path P, will be denoted $u \in P$. Our notion of reachability is purely static and only depends on the shape of the control flow graph, it does not take into account the execution of the program: even if a node is reachable from the start of the program, it does not mean there exists an input where the node is executed.

Using this purely static definition of reachability, we assume that every node is reachable from r, this means that unreachable basic blocks have been pruned from the program. Every time the post-dominance property, as defined below, will be used, we will assume that every basic block can reach the exit of the procedure. This is not always the case, for example "noreturn" procedures, which contain an infinite loop, do not fulfill this property. But, in order to fulfill this property, we can always add artificial edges to the CFG, even if in practice they will not be traversed during the execution.

Usually, each node in the CFG represents a **basic block**, i.e., a sequence of successive instructions in the program with no branches or branch targets interleaved. In order to simplify definitions and proofs, we will sometimes assume that every node consists of a single instruction.

Dominance and post-dominance property

A node u in a CFG **dominates** a node v, denoted u dom v, if every path from the entry node r to v contains u. If u dom v and $u \neq v$, then u **strictly dominates** v, denoted u sdom v. The node u is the **immediate dominator** of v, denoted idom(v), if u sdom v and there exists no node w such that u sdom w and w sdom v. Every CFG node other than r has a unique immediate dominator. The directed graph whose nodes are the nodes of the CFG and in which each node other than r is pointed to by its immediate dominator is a tree rooted at r, called the **dominator tree**. The **dominance frontier** of a set of nodes S, denoted $DF(S)$ is the set of nodes v such that there exists $u \in S$, u does not strictly dominate v but dominates a predecessor of v. Figure 2.1 provides an example for the dominance frontier of a node

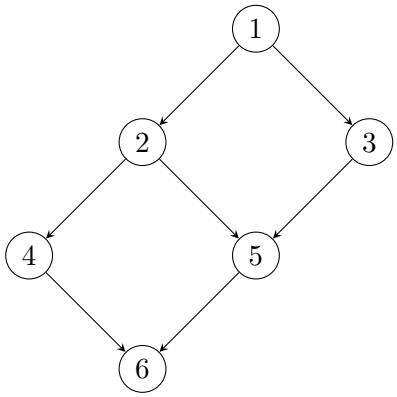


Figure 2.1: Node 2 dominates node 4, but 4 does not strictly dominates one of its successor: 6. Hence 6 is in the dominance frontier of 2. Similarly 5 is in the dominance frontier of 2.

in a simple directed graph. Less formally, for a given node u, this is the set of nodes where u stops dominating other nodes. The iterated dominance frontier is the limit of the following sequence:

$$DF_1(S) = DF(S)$$

$$DF_{i+1}(S) = DF(S \cup DF_i(S))$$

The post-dominance relationship is defined similarly in a CFG. A node v **post-dominates** a node u, denoted v pdom u, if every path from u to the exit node t contains v. The **strict post-dominance** (v spdom u), the **immediate post-dominator** (ipdom(u)), the **post-dominator tree** rooted at t, and the post-dominance frontier are defined analogously. Notice that the post-dominance information is equivalent to the dominance information if the direction of every edge of the CFG is reversed.

Figure 4.3 provides an example of a control-flow graph, and its dominator and post-dominator trees.

2.1.2 Traversal and order of CFG

Many algorithms are based on particular walk of a control flow graph, the depth first search. Starting at the root node, every children is recursively visited in the following way:

Algorithm 1 Depth-first search of a control flow graph.

```
1: function DFSearch(CFG (V, E, r))
2:     for each n ∈ V do
3:         State(n) ← unvisited
4:     DFS(r)
5: function DFS(node n)
6:     State(n) ← visited
7:     for each s ∈ succ(n) do
8:         if State(s) = unvisited then
9:             DFS(s)
10:    State(n) ← finished
```

During the walk every node can be in several states:

- unvisited, when the node has not yet been visited,
- processing, DFS(n) has been called but the recursive call did not return yet
- finished, DFS(n) has been called and returned, all of n children have been processed

From those state transition we can define an ordering of the control flow graph nodes. Before every call to the DFS function, we increment a global counter, the *preorder* number of a node will be the time when DFS(n) is called, and (transition from unvisited to processing), and the *postorder* of a node will be the time when DFS(n) returns (transition from visited to finished).

Those order have some classic properties:

- the preorder number of a node is always smaller than its postorder;
- in an acyclic graph, the reverse postorder of a graph is a topological order (that is the source of an edge always has a greater postorder number than the target);
- for every node, if its preorder number is included between the preorder and the postorder number of another, then it is a descendant in the spanning tree as defined below.

A DFS walk of the control flow graph defines a spanning tree of the control flow graph, if we only keep the edges which are followed. From this spanning tree, we can classify the edges in three categories:

- tree edge, going from a node in the spanning tree, to one of its successor in the spanning tree,
- forward edge, going from a node in the spanning tree, to one of its descendant in the spanning tree,
- cross edge, going from a node in the spanning tree, to another branch,
- back-edge, from a node to one of its ancestor in the spanning tree.

The graph obtained by removing the back-edges from a control flow graph is an acyclic graph. Furthermore, as we will see later, those back-edges play a role in loop structure.

Adding the preorder and postorder numbering to the DFS algorithm yields algorithm 2.

Algorithm 2 Depth first search walk, numbering nodes with pre- and postorder numbering

```
1: function DFSearch(CFG (V, E, r))
2:     time ← 0
3:     for each n ∈ V do
4:         State(n) ← unvisited
5:     DFS(r)
6: function DFS(node n)
7:     pre[n] ← time
8:     time ← time + 1
9:     State(n) ← visited
10:    for each s ∈ succ(n) do
11:        if State(s) = unvisited then
12:            Add arc (n, s) to the spanning tree
13:            DFS(s)
14:    post[n] ← time
```

2.2 Loops

Intuitively, a cycle in the control flow graph represents a loop, a sequence of instructions which can be repeated during the execution of the program.

2.2.1 Definition

A control flow graph can be more or less structured, for example the use of `goto` in the source language can create arbitrary control flow, while if only `if`, `for` or `while` are used to drive the control flow of the program, the resulting control flow graph is more *structured*. More precisely, a set of nodes X is said to be strongly connected if there exists a path, with only nodes from X, between any two nodes from X. A more formal definition of *structured* is that a control flow graph is said to be *reducible* if every strongly connected component possess a node that dominates every node from the component.

While the definition for loops in reducible control flow graphs is unique among the literature (since we can uniquely identify the entry of the loop), multiple definitions have been proposed to define loops in non-reducible CFGs. Ramalingam in [27] presents a definition for loops which generalizes the previous definitions of loops in the non-reducible case. We will present and use this more general definition.

A loop, denoted (B, H), consists of a strongly connected component: the loop body B, and a non-empty set of distinguished nodes from the body: the headers H. The loop will need to satisfy other properties, the first is the nesting property, two loops from a program should either be disjoint or be nested. This allows us to define relations between loops, every nested loop have a unique parents, and the set of loops from the program form a forest. For the header of the loop, if the loop is reducible (intuitively, if there is a unique entry node for the loop), all definitions agree: there is a unique header, the entry node. In other cases, different choices are possible, Ramalingam proposes a definition that covers all previously proposed possibilities.

Let $Entries(X)$, denote the entry nodes from X, that is the nodes from X which have at least one predecessor not in X. And let $UnDominated(X)$, denote the set of nodes from X, such that no node from X strictly dominates them. Obviously $Entries(X) \subset UnDominated(X)$. In general, the headers will be picked among the un-dominated nodes from the loop body. For reasons which will become clearer in 2.2.2, if instead of picking the headers from $UnDominated$, they are chosen out of $Entries$, we will say that the loop

forest is *connected*.

The set of loops we choose must cover every possible strongly connected component from the graph. First we define the notion of cover: a loop (B, H) covers a set of nodes X, iff $X \subset B$ and $B \cap H \neq \emptyset$. We then add the following requirement: decomposition of a CFG in loops, for every strongly connect component of the CFG, there exists a loop that covers it.

Ramalingam [27] gives a constructive proof of existence of minimal loop forest, which holds as long as the function used to select the headers from a strongly connected component satisfies the property that the sets are non-empty, and are a subset of $UnDominated$. The construction works in the following way: each step of the decomposition computes a set of SCCs, which form a new level of loops in the loop nesting forest. From this loops, headers are chosen according to the cover and $Undominated$ property. Then, for each such loop L, the loop-edges, i.e., the edges from a node in L to a loop-header of L, are removed. This process iterates until no strongly component is left.

Since for every strongly connected component, $Entries \subset UnDominated$ and $Entries \neq \emptyset$, a connected loop forest is also a valid loop forest.

We previously defined the back-edge, in term of a spanning tree (implied by a depth-first search) of a control flow graph. Given a set of loops, we define loop-edges as the set of edges (a, b), such that there exists a loop (B, H) with $a \in B$ and $b \in H$, in other words, an edge from a node in the loop to one of its header. Obviously, for reducible graphs, back-edges and loop-edges designate the same thing. But for non-reducible graphs they can be different.

A minimal loop nesting forest is a loop nesting forest such that no loop body from the forest is covered by another loop. Such loop nesting forests holds interesting properties, for example every two sets of headers are disjoint (a node can only be a header of one loop).

Figure 4.4 provides an example of a control-flow graph and its loop nesting forest.

2.2.2 Minimal connected loop nesting forest

In a minimal loop nesting forest denoted by $\mathcal{L}$, let $\mathcal{F}_{\mathcal{L}}(G)$ be the graph obtained after removing all loop-edges from G. As proved in [27, Theorem 2], this graph is acyclic. We will sometimes call this graph the *reduced graph* of G. It has a topological order that respects the nesting of loops, which means that all nodes of a given loop can be visited before visiting any other disjoint loop [27, Theorem 4]. To see this, we can order the nodes of the loop-tree

during its construction: at each level of the decomposition, the children of a loop are sorted according to a topological order of the DAG obtained by removing all loop-edges and considering each resulting strongly connected component (SCC) as a single node.

Furthermore, a topological order of $\mathcal{F}_{\mathcal{L}}(G)$ respects the dominance relation if it is connected, i.e., if there is a path from the root r to any other node u. Indeed, if v dominates u, then any path from r to u contains v. Since the graph is connected, at least such a path exists and v is processed before u in any topological order. In theorem 1, we show that this occurs when loop headers are entry nodes, i.e., when the minimal loop forest is **connected** as defined previously. Additionally, we prove the existence of such a loop nesting forest.

Theorem 1. *Consider a CFG with root r from which there is a path to any other node, and r is not part of any strongly connected component. Then, there exists a minimal connected loop forest $\mathcal{L}$, and for every such loop forest there is a path from r to any other node in $\mathcal{F}_{\mathcal{L}}(G)$.*

Proof. Let us recall the construction of a loop nesting forest: each step of the decomposition computes a set of SCCs, which form new loops in the loop nesting forest. Then, for each such loop L, the loop-edges, i.e., the edges from a node in L to a loop-header of L, are removed. By induction, we prove that there still exists a path from r, the CFG entry node, to any other node in the CFG after the removal of these edges. For the basis, observe that each CFG node is reachable from r in the initial CFG, prior to the identification of the first set of loops.

Then, let L be a loop in the nesting forest. Let G' and G'' denote the CFG before and after the removal of the loop edges of L. By the induction hypothesis, all nodes are reachable, in G', from r. As $r \notin L$, there exists at least one entry node of L, i.e., a node in L with an incoming edge from outside of L. Therefore, it is always possible to select a set of loop-headers that are also entry nodes of L, and the decomposition can continue.

Note that any path ending at an entry node u for L and whose previous node v in the path is not in L cannot contain a node in L, except u, otherwise v would also belong to the SCC L. Thus, in G', there is a path from r to any entry node u of L that does not contain any node in L except u. None of the edges along this path are loop-edges of L, so this path remains present in G''. On the other hand, if u is not an entry node of L, consider a path, in G', from r to u. Let v be the last (if any) entry node of L in P. The sub-path

from v to u does not contain any loop-edges for L and v remains reachable from r in G''. Therefore, concatenating these two paths ensures that a path from r to u exists in G''. □

2.3 Join set

Given a set of nodes S, let $J(S)$ be the set of nodes x such that there exists $y \in S$, and $z \in S$, with a non-empty path from y to x and a non-empty path from z to x, disjoint except for x. Let us define the iterated join set ($J^+(S)$) to be the limit of the following sequence:

$$J_1(S) = J(S)$$

$$J_{i+1}(S) = J(S \cup J_i(S))$$

Michael Wolfe, in [33], extended the results of Michael Weiss [32], and proved the general equivalence between the joint set and the iterated join sets.

Theorem 2. *For any set of nodes S, $J(S) = J^+(S)$.*

Additionally, Cytron et al. [18] implicitly provide the following theorem on the equivalence of join-sets and the iterated dominance frontier. An explicit version can be found in the work of Michael Weiss [32].

Theorem 3. *For any set of nodes S containing the entry, $J^+(S) = DF^+(S)$.*

2.4 Static single assignment (SSA) form

2.4.1 Definitions

The static single assignment (SSA) form was first presented in two papers in POPL'88, focusing on the identification and elimination of redundant computations [1, 12]. Those papers were followed by a journal paper [18] presenting more thoroughly the foundations of the SSA form as well as various construction algorithms.

The main concept behind the SSA form is that every variable satisfies the single definition property. This means that every variable is only being assigned once, textually. The property cannot be achieved with only the

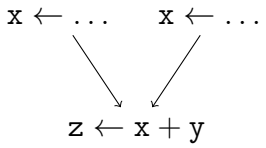
help of renaming, that is choosing a different name for every definition of a variable and renaming every use appropriately. In some cases, two distinct definition reach the same use, depending on the actual execution. To solve this issue, SSA form introduce a new concept, the ϕ-functions.

A ϕ-function can only be inserted at the start of a basic block, and is usually inserted at a join point. It has the same number of arguments as the join point has incoming edges. Every ϕ-functions of a basic block are executed concurrently, the value returned by the function depends on the execution flow. We suppose the incoming edges are ordered, if the instruction is reached via the i-th incoming, then the ϕ-function returns the value of its i-th argument. Another way to view the operation is to imagine copies happening on the edges, the variable at the left-hand side will have as many potential definitions as there are incoming edges, ϕ-functions have the same semantics but allow to have the single definition property.

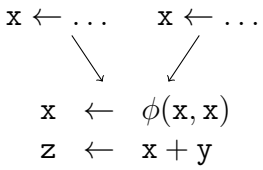
In most cases, to simplify liveness and dominance under SSA, instead of considering that the use associated with a ϕ-function happens on the edge, we can assume it is located at the end of the associated predecessor block.

2.4.2 Minimal SSA

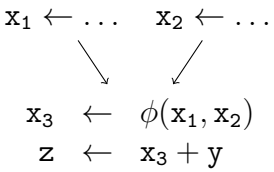
We only defined a property that must be satisfied for a program to be in SSA form and a new instruction allowing us to properly transform a program into SSA form. We now describe how to actually transform a program into SSA form. The textbook algorithm works in two different phases: first, the points where ϕ-functions have to be inserted are computed, then, variables are renamed in order to satisfy the single definition property. For the placement of ϕ-functions, the journal paper of Cytron et al. [18] uses the notion of join points: for a given variable `v`, ϕ-functions are inserted at the iterated join set $J^+(D)$ where D is the set of program points where `v` is defined. It is easier to compute the iterated dominance frontier (DF^+) than the join sets. Since $DF^+(S) = J^+(S)$ if S includes the root of the CFG, the algorithm from Cytron et al. adds a pseudo definition of `v` at the root of the control flow graph to ensure the root node is always included in the set of definitions, this lets them use the iterated dominance frontier instead of the join sets. After inserting the necessary ϕ-functions, but before renaming the variable, we can minimize the number of inserted ϕ-functions such that every use is only reachable from one definition, the resulting SSA form is called the minimal SSA form.



(a) Non-SSA program



(b) With single reaching definitions



(c) SSA program after renaming

Figure 2.2: Placement of ϕ-functions

2.4.3 Liveness and variants of SSA form

The minimal SSA form may insert ϕ-functions and create new variables at merge points where a variable is not live in the original code. Those additional useless variables could increase the runtime of some analysis or optimizations. To avoid this problem, two SSA variants have been introduced: semi-pruned and pruned SSA form.

Many variables are local: they are only used within the basic block where they are defined. The semi-pruned SSA does not create any ϕ-function for those block-local variables.

The pruned SSA is more precise, with the help of a liveness analysis, it avoids creating ϕ-functions if a given variable is not live in the original program. This pruning can be done directly while transforming the program into SSA form, or start from a program in semi-pruned form and prune the dead (useless) ϕ-functions.

Chapter 3

Liveness analysis under SSA

Liveness analysis provides information about the points in a program where a variable holds a value that might still be needed. Thus, liveness information is necessary for most optimizations passes related to storage assignment. For instance optimizations like software pipelining, trace scheduling, and register-sensitive redundancy elimination make use of liveness information. In the code generation part, particularly for register allocation, liveness information is mandatory.

Traditionally, liveness information has been computed with data-flow analysis techniques (e.g. see [16]). But they have major drawbacks since the computation is fairly expensive (several iterations are needed) and the results are easily invalidated by program transformations. Indeed, adding instructions or introducing new variables requires suitable changes in the liveness information: partial re-computation or degradation of its precision. Furthermore one cannot easily limit the data-flow algorithms to compute information only for parts of a procedure. Computing a variable's liveness at a program location generally implies computing its liveness at other locations, too.

In this chapter we describe alternatives algorithms for computing the liveness information. We present our novel algorithm, based on the loop structure of the graph, which builds the liveness information in two-passes: one backward in the control flow graph, and one forward in an order derived from the loop structure. Then we present simpler algorithms solely based on the exploration of paths. Finally we solve a more complex problem: interference between variables. As we will show, this problem not only involves the liveness information related to the variables, but their actual

value as well.

3.1 Definitions

A variable is live at some CFG node if both:

1. its value is available at this node. This can be expressed as the existence of a *reaching definition*, i.e. existence of a directed path from a definition to this node.

2. its value might be used in the future. This can be expressed as the existence of an *upward exposed use*, i.e. existence of a directed path from this node to a use that does not contain any definition of this variable.

In fact, the reaching definition constraint is useful only for non-strict programs. A strict program is a program such that every path from start node to a use of a variable contains the definition of the variable. An upward exposed use at the entry of the CFG, is a *potential* bug in the program, even if the dynamic execution could be safe, for example if every executed path defines the variable before any use as in figure 3.1. In that case the compiler usually dumps a warning message (use of a potentially undefined variable). From now on, we will assume the program to be in **strict SSA form**, the definition of a variable dominates all its uses (dominance property). To simplify some definitions and proofs, we sometimes consider that every CFG node consists of a single instruction. Liveness can then be defined as follows:

Definition 1. *A variable* `a` *is live-in at a node q if there exists a directed path from q to a node u where* `a` *is used and that path does not contain the definition of* `a`*, denoted as* $def_{\texttt{a}}$.

Definition 2. *A variable* `a` *is live-out at a node q if it is live-in at least at one successor of q.*

3.2 Liveness set

Sometimes instead of computing the program points where a variable is live, it is sufficient to compute the set of live variable only at some particular

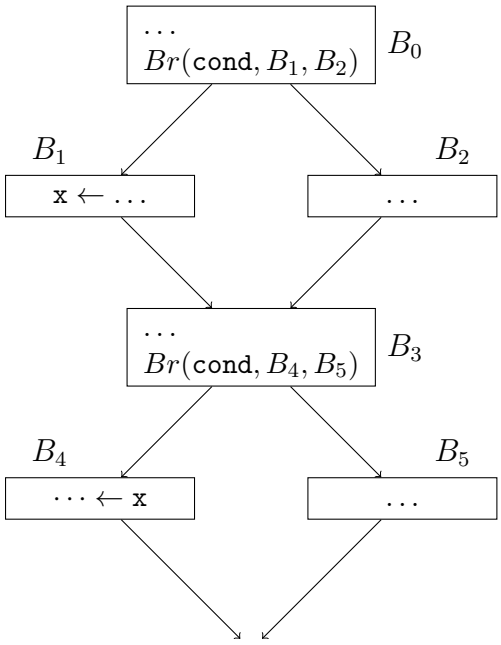


Figure 3.1: There exists a path from B_0 to B_4 that does not contain any definition of `x`. But the program is still correct since, because they are guarded by the same condition `cond`, B_4 is only executed only if B_1 is executed as well, and `x` is never used before being defined.

program point. Typically, live sets are computed at basic blocks boundaries, since the lack of control flow inside basic blocks allows to recompute the precise liveness information easily. We call this analysis the computation of live-in and live-out sets.

3.2.1 Classical liveness set construction

We defined liveness in term of the existence of paths, that is a relation between a node and its successors. Since liveness information evolves around paths, the traditional approach has been to describe the problem using data-flow equations. Indeed, the live-in and live-out definitions are easily mapped into a set of data-flow equations.

We assume we are under strict SSA Form, thus every variable has a unique definition which dominates all its uses. Let us note Killing(B) the set of variables which are defined in the basic block B, and UpwardExposed(B) the set of variable that are used in the basic block and are defined in another basic block. Intuitively, we consider a basic block as a huge instruction, defining some variables (Killing) and using other variables (UpwardExposed), while any local variable that does not escape the basic block is omitted. We can then simply map the definitions to the following equations:

$$\begin{aligned} \text{LiveIn}(B) &= \text{UpwardExposed}(B) \cup (\text{LiveOut}(B) - \text{Killing}(B)) \\ \text{LiveOut}(B) &= \cup_{S \in \text{succs}(B)} \text{LiveIn}(S) \end{aligned}$$

Those equations translate to the data-flow algorithm found in algorithm 3.

As with all data-flow analysis, the order in which the CFG nodes are processed is crucial. If the graph was acyclic, ordering the nodes such that every node is processed after all its successors are processed would avoid doing any iteration. An example of such an order is the postorder of a graph. In practice even if most control flow graphs have loops, thus the postorder will not make the information flow everywhere in one pass, but it will still propagate quite far, that is the reason why the chosen order is usually the postorder.

But even using the postorder, the number of iterations depends on the loop depth. For example in Figure 3.2, we have four loops: $(\{2\}, \{2, 3, 4, 5, 6\})$,

Algorithm 3 Classical liveness algorithm using data flow

```
1: function LIVENESS_DATAFLOW(CFG (V, E))
2:     for each B ∈ V do
3:         LiveIn(B) ← UpwardExposed(B)
4:         LiveOut(B) ← ∅
5:     changed ← true
6:     while changed do
7:         changed ← false
8:         for each B ∈ V, in postorder do
9:             newLiveOut ← ∪_{S∈succs(B)} LiveIn(B)
10:            if newLiveOut ≠ LiveOut(B) then
11:                LiveOut(B) ← newLiveOut
12:                changed ← true
```

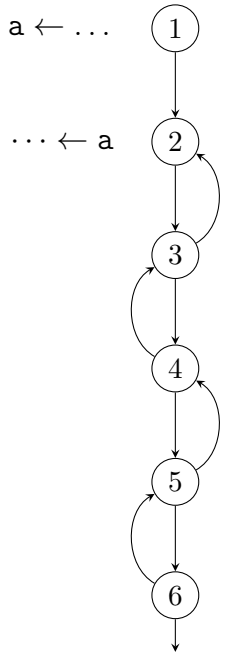


Figure 3.2: A control flow graph where the classical liveness algorithm using backwards data-flow will iterate as many times as the loop depths, when using the postorder to drive the data-flow. At each iteration, the variable a will be marked as live in one more basic block below the only use.

$(\{3\}, \{3, 4, 5, 6\})$, $(\{4\}, \{4, 5, 6\})$ and $(\{5\}, \{5, 6\})$. After each iteration, the information will reach one more loop starting from the outermost one.

We can note that, in the case of forward data-flow, Cooper [17] advocate for the use of a reverse postorder of the control flow graph. For backward-data flow algorithm, most authors [3] simply assume it is the same type of problems with the edges reversed and thus use the reverse postorder of the control flow graph with all edges reversed. But we do not see any compelling reason to choose this order over a simpler postorder of the control flow graph.

Kam et al. [24] explored the complexity of round-robin data-flow algorithms, that is algorithms that iterate over a fixed order until there are no new changes. They show that for forward data-flow problems, if the graph is reducible, there will be at most $d(G) + 3$ passes over the set of nodes, where $d(G)$ is the depth of the most nested loop. To apply this complexity analysis to *backwards* data-flow problems, the *reverse* control flow graph needs to be reducible. However while most control flow graph are reducible, it is not true from a typical reverse control flow graph: many programming languages offer a way to *break* out of a loop from multiple locations. Furthermore some compiler have passes to transform every loop of a control flow graph into a natural loop, but those passes are usually not applicable for the reverse control flow graph. It is an open conjecture to know if the bound proved by Kam et al. [24] is valid for backwards data-flow with postorder. If that conjecture was proven, it would remove any incentive to use the reverse postorder of the reverse graph for round-robin algorithms solving backward data-flow problems.

3.2.2 Two pass data flow

As we have seen with the classical data-flow algorithm, the algorithm needs to iterate in order to propagate the information across loops. With the help of SSA properties, and using the loop structure, can we do better?

The use of SSA properties, single definition and dominance, allows to compute the liveness information in two passes. We show the result first for reducible graphs, and will generalize to non-reducible graphs later on. First, a backward pass propagates the liveness information upwards to the loop headers. Then a second pass propagates the liveness from the header of the loops to their bodies.

The first past consists on a propagation of the liveness, in a way similar to a single pass of a round-robin data-flow algorithm, using a reverse topological

order of the acyclic graph where the loop-edges are removed (the reduced graph, $\mathcal{F}_{\mathcal{L}}(G)$). The second pass descends into the loop tree, propagating the liveness from headers to the loop body for each loop.

We first show that the backward pass on a reverse topological order of the reduced graph correctly propagates the information to the loop headers.

Lemma 1. *In a reducible CFG, given a variable* `v` *and its definition* d*, for every maximal loop* L *with header* h *such that* L *does not contain* d*,* `v` *is live-in at* h *iff there is a path in* $\mathcal{F}_{\mathcal{L}}(G)$*, the reduced graph, from* h *to a use not containing* d*.*

Proof. Given a variable `v` defined in d, and a maximal loop L with header h not containing d. If `v` is live at h, there exists a cycle-free path from h to a use of `v` in the CFG, which does not go through d. Take this path and suppose there is a loop-edge (s, h') in this path, h' being the header of a loop L', and $s \in L'$. $h' \neq h$ otherwise the path would contain a cycle, this means $L \neq L'$.

- $h \in L'$ is not possible, since L was the biggest loop not containing d, it would imply $d \in L'$ and h would dominate d which contradicts the fact that `v` is live-in at h.
- $h \notin L'$, since the graph is reducible, the only way to enter L' is through h', and it would mean there was a previous occurrence of h' in the path, this breaks our hypothesis that the path is cycle-free.

Thus the path does not contain any loop-edges, and it is a valid path from the acyclic graph. Conversely, if there exists a path in the acyclic graph, then `v` is live-in at h, since the acyclic graph is a subgraph of the CFG. □

In the previous lemma, it is not guaranteed there exists a loop L satisfying the conditions. The following lemma covers this case:

Lemma 2. *In a reducible CFG, given a variable* `v` *and its definition* d*, for every program point* p *such that no loop contains* p *but not* d*,* `v` *is live-in at* p *iff there is a path in the reduced graph* $\mathcal{F}_{\mathcal{L}}(G)$ *from* p *to a use not containing* d*.*

Proof. Given a variable `v` defined in d, and a program point p such that `v` is live at p and no loop contains p but not d. Since `v` is live at p, there exists a cycle-free path from p to a use of `v` in the CFG, that does not go through d.

Take this path and suppose there is a loop-edge (s, h) in this path, h being the header of a loop L, and $s \in L$:

- $p \in L$ is not possible, since it would imply $d \in L$ and h would dominate d.
- $p \notin L$, since the graph is reducible, the only way to enter L is through h, and it would mean there was a previous occurrence of h in the path, this breaks our hypothesis that the path is cycle-free.

We built a path which does not contain any loop-edges, thus a valid path from the acyclic graph. Conversely, if there exists a path in the acyclic graph, then `v` is live-in at p, since the acyclic graph is a subgraph of the CFG. □

Those two lemmas prove that if we propagate the liveness information only along the edges of $\mathcal{F}_{\mathcal{L}}(G)$, the acyclic reduced graph, then for a variable `v`, every program point that is either not part of a loop not containing d, or is the header of the biggest loop not containing d, will have `v` marked as live-in.

Furthermore, the first lemma proves that if after the first pass (the backwards propagation along the reduced graph) a program point's live-in is not accurate, then the missing variable is already in the live-in set of the header of one of the surrounding loops. We now prove that every variable live-in at the header of the loop should also be live-in at every program point of the loop body.

Lemma 3. *If a variable* `v` *is live-in at a header of a loop, then* `v` *is live-in at each node from the body of the loop,*

Proof. Given a loop L with header h, such that the variable `v` defined at d is live-in at the loop (it is live-in at h). Since it is live at h, because of the dominance property, h is strictly dominated by d, this mean d is not contained in L. Furthermore there exists a path from h to a use of `v` which does not go through d. For every node of the loop, p, since the loop is a strongly connected component of the CFG, there exists a path, consisting only of nodes from L from p to h. Concatenating those two paths proves that `v` is live-in and live-out of p. □

This lemma proves the correctness of the second pass, which propagates the liveness information inside loops. Lemma 1 proved that every program point which did not have a correct liveness information will be marked after

the second pass, this lemma proves that every program point marked by the pass is indeed live-in. Overall, this proves the correctness of our algorithm.

While a round-robin data-flow algorithm can, in the worst-case, do as many iteration as the depth of the loop nesting forest, with our loop-based algorithm, the first pass will update the header of the outermost loop, while the second pass will directly update every node part of the loop, including the bottom. No iteration is needed. That is the case in Figure 3.2: the first pass will update the liveness information for the outermost loop header (2), and the second pass will update all its descendants in the loop-nesting forest (3, 4, 5, 6).

In the following algorithms (algorithms 4, 5, and 6), we define PhiDefs(B) as the set of variables defined by a ϕ-functionat the entry of B and PhiUses(B) as the set of variables used as operand in a basic block successor of B.

Algorithm 4 First pass of the loop-based liveness analysis.

```
1:  function DAG_DFS(block B)
2:      for each S ∈ CFG_succs(B) such that (B, S) is not a loop-edge do
3:          if S not processed then DAG_DFS(S)
4:      Live ← PhiUses(B)
5:      for each S ∈ CFG_succs(B) such that (B, S) is not a loop-edge do
6:          Live∪ ← LiveIn(S) − PhiDefs(S)
7:      LiveOut(B) ← Live
8:      for each program point p in B, backward do
9:          remove killing definition at p from Live
10:         add uses at p in Live
11:     LiveIn(B) ← Live ∪ PhiDefs(B)
12:     mark B as processed
```

3.2.3 One variable at a time

Instead of computing the liveness globally, for every variables at the same time, we can update the liveness sets variable per variable.

For a given variable, if the def-use chains (a pointer from every use to the instruction that defines the variable) are available, a simple algorithm can be used to build the live-sets. Starting at the uses, the algorithm follows every edges backwards and stops when visiting the block containing the definition.

Algorithm 5 Second pass of the loop-based liveness analysis.

```
1: function LoopTree_DFS(block B)
2:     if C not a loop header then
3:         return
4:     for each C ∈ loopTree_children(B) do
5:         LiveIn(C)∪ ← LiveIn(B) − PhiDefs(B)
6:         LiveOut(C)∪ ← LiveIn(B) − PhiDefs(B)
7:         LoopTree_DFS(C)
```

Algorithm 6 Loop-based liveness analysis

```
1: function Compute_LiveSets_SSA_Reducible(CFG)
2:     for each basic block B do
3:         unmark B
4:     Let R be the root of the CFG
5:     DAG_DFS(R)
6:     LoopTree_DFS(R)
```

The variable is live on every visited edge. From that we deduce the live-in and live-out sets of the basic blocks we visit.

While this technique is not specific to the SSA form, it is more convenient to use it for variables in SSA Form: def-use chains are easily computed from the use-def chains (which are a by-product of the SSA form, every use is associated with a unique definition) and they can be kept around and maintained as long as the SSA form is used. With this algorithm, every basic block will be visited once for every variable either live-in or live-out.

The principle of algorithm 7 is similar to the algorithm used by Appel to construct the interference graph in [3].

3.2.4 One use at a time

If the def-use chains are not available, the live sets can still be built in an efficient manner, in a way that is not SSA specific, as shown by algorithm 8. First, collect all the definitions for every variable. Then, process every basic-block of the CFG. For each basic block, process each instruction from bottom to top, and collect the local live-in sets (the variables with a use in the basic block, which are not preceded by a definition). After the local live-in sets is computed, for every variable in this set, we visit every ancestor from the

Algorithm 7 Liveness analysis based on building the live-range of every variable one after the other.

```
1: function LIVENESS(variable v)
2:     Seen ← ∅
3:     Visit ← []
4:     D ← basicBlock(d)
5:     for each u ∈ uses(v) do
6:         U ← basicBlock(u)
7:         if U ≠ D then
8:             Visit.append(U)
9:     while Visit is not empty do
10:        C ← Visit.pop()
11:        if C ∈ Seen then
12:            continue
13:        Seen.add(C)
14:        LiveIn(C).add(v)
15:        for each B ∈ predecessor(C) do
16:            LiveOut(B).add(v)
17:            if B ∈ Seen then
18:                continue
19:            LiveOut(U).add(v)
20:            if B ≠ D then
21:                Visit.append(B)
```

flow graph, marking the variable as live-in until a basic block containing a definition is reached.

Algorithm 8 Liveness analysis based on building the live-range of every variable. Def-use chains are computed on demand.

```
1: function LIVENESS(CFG)
2:     for each basicBlock B in postorder do
3:         Live ← LiveOut(B)
4:         for each instruction i in reverse order of B do
5:             for each v used in i do
6:                 Live.add(v)
7:             for each v defined in i do
8:                 Live.remove(v)
9:         for each variable v in Live do
10:            if v ∈ LiveIn(B) then
11:                continue
12:            Seen ← ∅
13:            Visit ← [B]
14:            D ← basicBlock(def(v))
15:            while Visit is not empty do
16:                C ← Visit.pop()
17:                if v ∈ liveOut(C) then
18:                    continue
19:                LiveIn(C).add(v)
20:                for each P ∈ predecessor(C) do
21:                    if v ∈ liveOut(P) then
22:                        continue
23:                    LiveOut(P).add(v)
24:                    if P ≠ D then
25:                        Visit.append(B)
```

This algorithm has some similarities with the algorithm used by the LLVM compiler in order to compute, for each variables, the sets of basic block where they are live.

3.3 Liveness check

Instead of computing live sets, sometimes we only need a simpler information: given a program point and a variable, is the variable live at the program point?

First we can use a very simple method, similar to the one variable at a time algorithm used to compute liveness sets. In this case instead of iterating over all variables, we only explore the blocks where a particular variable is live. Then, if the program point p is visited by the algorithm, `v` is live at this point. This approach is presented in algorithm 9.

Algorithm 9 Liveness-check based on live-range computation.

```
1: function IsLiveIn(variable v, programPoint p)
2:     Seen ← ∅
3:     P ← basicBlock(p)
4:     Visit ← []
5:     D ← basicBlock(d)
6:     if D = P and d after p then
7:         return false
8:     for each u ∈ uses(v) do
9:         U ← basicBlock(u)
10:        if P = U and u after p then
11:            return true
12:        Visit.append(U)
13:    while Visit ≠ [] do
14:        C = Visit.pop()
15:        for each B ∈ predecessor(C) do
16:            if B ∈ Seen then
17:                continue
18:            if B = P then
19:                return true
20:            Seen.add(B)
21:            if B ≠ D then
22:                Visit.append(B)
23:    return false
```

This method does not need any pre-computation or any additional data-structure. In some cases, depending on the number of queries, more efficients

methods can be used. If we pre-compute some data-structures first, the query cost can be lowered.

We now describe such a solution, where we use an auxiliary data structure to lower the query cost. First we examine the case where the graph is reducible. We will later adapt our algorithm to the more general case of potentially irreducible graphs.

Using the results from 3.2.2, we know that a variable defined at a program point d is live at a program point p iff it is live at p or at the header of the biggest loop containing p and not d, using only $\mathcal{F}_{\mathcal{L}}(G)$, the acyclic reduced graph, to test for reachability.

Given the reduced graph and the loop nesting forest, finding out if a variable is live at some program point can be done in two steps, as shown in algorithm 10. First, if there exists a loop containing the program point p and not the definition, pick the header of the biggest such loop instead as the query point. Then check for reachability from this program point to any use of the variable. Correctness is proved from the theorems used for liveness sets.

Finding the biggest loop not containing the definition but containing the query point is a problem similar to finding the least common ancestor (LCA) of two nodes in a tree: the loop in question is the only direct child of the least common ancestor which is an ancestor of the smallest loop containing the query point. A LCA query can be answered in $O(1)$, with a pre-computation of $O(n)$, as described in [4]. The algorithm described by Bender is based on the Euler tour of the tree, that is the sequence of nodes as they are visited by a depth first search. A simple adaptation of their algorithm can be made, based on the insight that the biggest loop containing the query but not the definition is the next node after the last occurrence of the LCA in an Euler tour of the loop-tree.

But since the depth of the tree is usually small, a simpler solution can be used: the naive solution that just walks upward in the tree starting at both nodes and stops when it encounter a common ancestor.

The third alternative method (shown in algorithm 11) is to pre-compute the set of ancestors from the loop-tree for every node. Then a simple set operation can find the node we are looking for: the ancestor of the definition node are removed from the ancestor of the query point. From the remaining ancestors, we pick the shallowest. Using bitsets, indexed with a topological order of the loop tree, this operations are easily implemented. The removal is a `bit-inversion` followed by a `bitwise-and` operation, and the shallowest

node is found by searching for the `first-bit-set` in the bitset. Since the number of loops (and thus the number loop headers) is rather small, the bitsets are themselves small as well and this optimization does not result in much wasted space.

Algorithm 10 Liveness-check based on loop structure.

```
1: function IsLiveIn(variable v, programPoint p)
2:     Seen ← ∅
3:     P ← basicBlock(p)
4:     Visit ← []
5:     D ← basicBlock(d)
6:     if D = P and d after p then
7:         return false
8:     B ← biggestLoop(P, D)
9:     for each u ∈ uses(v) do
10:        if basicBlock(u) ∈ Reachable(B) then
11:            Return true
12:    return false
```

Algorithm 11 Given two basic blocks, find the biggest loop containing the first block but not the second.

```
1: function GreatestNonCommonAncestor(basicBlock A, basicBlock
   B)
2:     loopA ← loopNode(A)
3:     loopB ← loopNode(B)
4:     nonCommonAncestors ← loopTreeAncestors(loopA) −
   loopTreeAncestors(loopB)
5:     if nonCommonAncestors = [] then
6:         return A
7:     return min(nonCommonAncestors)
```

The reachability information can be stored in bitset, in every basic block.

3.4 From reducible CFG to irreducible CFG

The algorithms based on loops that we described above, for both the liveness-set problem and the liveness-query problem, are only valid for reducible

graphs. We can derive an algorithm that works for irreducible graphs as well, in the following way: transform the irreducible graph to a reducible graph while keeping the liveness identical, in practice the graph *is not actually modified*, but the algorithms are changed to simulate the modification of some edges, on the fly.

As hinted by Ramalingam, adding a dummy node to represent the headers can help dealing with irreducible graphs. Contrary to Ramalingam we want to keep the liveness identical as well, not only the dominance information.

We can iteratively construct our modified graph G' from the control flow graph G in the following way:

Given a loop L, we define a new graph Ψ'_L as (V', E'), where:

$$\begin{aligned} V' &= V \cup \{\delta_L\} \\ E' &= E - \mathit{LoopEdges}(L) - \mathit{EntryEdges}(L) \\ &\quad \cup \{(p, \delta_L) | p \in \mathit{PreEntries}(L)\} \\ &\quad \cup \{(p, \delta_L) | \exists x \in \mathit{Headers}(L), (p, x) \in \mathit{LoopEdges}(L)\} \\ &\quad \cup \{(\delta_L, h) | h \in \mathit{Entries}(L) \cup \mathit{Headers}(L)\} \end{aligned}$$

In other words, for a given loop L, add a new node to the graph: δ_L. Replace every entry [1] edge (x, y) by two edges (x, δ_L) and (δ_L, y). Similarly replace every loop-edge (x, y) by two edges (x, δ_L) and (δ_L, y).

Applying the transformation to every loop in the forest, will transform G into G'. Contrary to the graph built by Ramalingam, G' is not acyclic. In fact the loop structure is preserved. Furthermore, since, from the construction, every loop has only one entry node (δ_L), the graph is reducible.

Theorem 4. *Given a loop L, for every two nodes x and y from G, x dom y in G iff x dom y in Ψ'_L.*

Proof. Take two nodes x and y from G, such that x dom y in Ψ'_L. Given p an arbitrary path in G from r to y, we show that x is included in that path. From p, we construct p' a path in Ψ'_L from r to y as follow: for every edge (u, v) from p not present in Ψ'_L (v is a header or an entry node), (u, δ_L) and (δ_L, v) are valid edges from E', and we replace (u, v) by those two edges. Since x dom y in Ψ'_L, and p' is a path from r to y, it must include x. δ_L is

[1] it is important not to restrict to the loop headers

the only node in p' that was not in the original path p, this proves x was in the original path.

Reciprocally, suppose that x dom y in G. We show that any path from Ψ'_L, p, from r to y, goes through x. We construct a path in G using p as a basis. Starting from the end of the path, for every sub-path u, δ_L, v in p:

- if $x \in L$, then there exists a path, p', from r to v in G which might only contain x as the last node (v is not dominated by any other node from the body of L), we replace the sub-path from r to v with p' and stop the transformation.
- else $x \notin L$, then there exists $v' \in L$ such that $(u, v') \in E$ and there is a path in G included in L from v' to v, this path does not contain x ($x \notin L$) and the sub-path can be replaced by the path from L.

Finally we have a path in G, from r to y. Because x dom y, this path contains x. Since our transformation did not add x, x was in p. □

Lemma 4. *Given three nodes x, y and z from G such that x dom z, there exists a path from y to z which does not contain x in G iff there exists a path from y to z in Ψ'_L which does not contain x.*

Proof. Given a path p from y to z in G which does not contain x. Since x dom z, x dominates every node from p. For every edge of p, (u, v), from G which does not exist in Ψ'_L, there exists δ_L and v' such that (u, δ_L) and (δ_L, v') are edges from Ψ'_L. v is either en entry node of the loop, or a loop header, it means it is not dominated by any node of the loop. Since x dominates every node from p, it dominates v, and x is not part of the loop. Hence, there exists a path in L from v' to v which does not contain x. So we can replace the edge (u, v) by the path $u, \delta_L, v', \ldots, v$ which does not contain x. Repeating this process will yield a path from Ψ'_L, from y to z which does not contain x.

Reciprocally, suppose we have a path in Ψ'_L from y to z which does not contain x. For every sub-path u, δ_L, v, since x dom z, x dominates every node from the path, in particular it dominates v. Since no node from the loop dominate v, x is not in L. It means there exists $v' \in L$ such that $(u, v') \in E$ and there is a path included in L from v' to v, this path does not contain x ($x \notin L$) and the sub-path can be replaced by the path from L. □

In the proof, no edges from δ_L to entry points which are not headers are used. We only rely on the fact that an edge from δ_L to every loop headers

exists. This will allow us to omit those edges in the following algorithms. It means that while omitting them affects the equivalence for dominance, they do not play any role for the liveness analysis.

Theorem 5. *For any SSA variable, and a node from G, it is live in G iff it is live in Ψ'_L.*

Proof. Direct from the existence of the path in previous lemma. □

With this equivalence, we use the transformation of the control flow graph to use our previous algorithm with irreducible graphs. The simplest solution is to build the transformed graph, adding the necessary new edges, removing others and adding new nodes for every loop (δ_L nodes). But we would like to emphasize that actually modifying the graph is *not* required.

The modification is quite simple in practice, and even more if we restrict ourself to loop-forest which only have one header per loop (as is the case of the loop forest structure presented by Havlak [21]) In this case, we do not need to insert any δ_L nodes. The header node and δ_L can be merged, since δ_L has only one outgoing edge, and the header has only one incoming edge.

Additionally, the insertion and removal of edges can be simulated as well: we need to redirect any entry- or loop- edge to it's δ_L node. We virtualize the modification of the CFG in the following way: for every visited edge, if the target of the edge is not part of the same loop, we virtually replace the edge by an edge to the δ_L node of the biggest loop containing the target but not the source of the edge. We would then need to add an edge from the δ_L node to every entry or header node. But as shown in our proof for liveness equivalence, the edges to the entry nodes are not necessary.

To summarize, if our loop forest only have one header per loop, the transformation is extremely simple, during the graph traversal, every entry edge is simply redirected to the header of the biggest loop containing the target but not the source (see algorithm 12). This transformation needs to be done for the computation of the reachable set, and for the DAG_DFS pass for liveness set, in both cases, every edge in CFG_succs needs to be transformed before further processing.

Algorithm 12 Edge transformation to make a control flow graph reducible while keeping the liveness identical.

```
1: function TransformEdge(CFGEdge (A, B))
2:     LoopA ← EnclosingLoop(A)
3:     LoopB ← EnclosingLoop(B)
4:     LoopBAncestor ← GreatestNonCommonAncestor(LoopB, LoopA)
5:     Redirect ← B
6:     if LoopA = LoopB then
7:         if isHeader(B) then
8:             Redirect ← Header(LoopB)
9:     else
10:        Redirect ← Header(LoopBAncestor)
11:    return (A, Redirect)
```

3.5 Interference: value and intersection of live-ranges

During register allocation or spilling, the information that is required is the interference between two variables. Is it possible for two different variables to share the same resources (usually a register)? Since computing this information exactly requires knowing the exact dynamic execution of the program, usually only a conservative approximation is computed. The simplest approximation is to only use the liveness information: if two live-ranges intersect, then we consider that the corresponding variables interfere.

Using live-sets, the intersection graph of all variables can be easily computed [3]. Since this construction is expensive, both in time and space, it is worth building it only if the algorithm asking for this information is intensively querying. Furthermore the results of the analysis are easily invalidated by many program transformations (for example value-numbering, live-range splitting, code-motion, etc.).

In practice, under SSA, intersection of the live-ranges of two variables can be expressed as a liveness-query [13].

Theorem 6. *The live-ranges of two different variables intersects if one variable is live at the definition point of the other variable.*

Proof. If the live-range of `a` and the live-range of `b` intersect. There exists p

a program point where a is live and b is live. Since the variables are under SSA Form, the definitions of a and b dominates p. We know that $\text{def}_\texttt{a}$ dom p and $\text{def}_\texttt{b}$ dom p, so one if the definition must dominate the other. Suppose $\text{def}_\texttt{a}$ dom $\text{def}_\texttt{b}$.

So there is a path, from $\text{def}_\texttt{b}$ to p, which does not contain $\text{def}_\texttt{a}$. Since a is live at p, there is a path from p to a use of a which does not contains $\text{def}_\texttt{a}$. This proves that a is live at $\text{def}_\texttt{b}$. □

Instead of building the intersection graph of the variables, we can dynamically answer intersection queries using the liveness-query algorithm shown in the previous section. Since the pre-computation only relies on the shape of the CFG, not on the variables themselves, no updates are necessary when a new variable is introduced.

It is common to find in the literature the following definition of interference "two variables interfere if their live ranges intersect" (e.g. in [20, 13, 30]) or its refinement "two variables interfere if one is live at a definition point of the other" (e.g. in [14]). Actually, a and b interfere only if they cannot be stored in a common register. Chaitin et al. discuss more precisely an "ultimate notion of interference" [15]: a and b cannot be stored in a common register if there exists an execution point where a and b carry *two different values* that are both defined, used in the future, and not redefined between their definition and use.

This definition of interference contains two dynamic (i.e., related to the execution) notions: the notion of liveness and the notion of value. Analyzing statically if a variable is live at a given execution point is not decidable since it can be reduced to the halting problem (given a variable defined at the entry of the program and only used at the exit node, it is live at the exit node if and only if the program terminates). We previously provided a (quite accurate in practice) approximation defined with paths: reaching definition and upward exposed use [3]. In SSA with the dominance property – in which each use is dominated by its unique definition, so it is defined – upward exposed use analysis is sufficient.

The notion of value is even harder and can be approximated using data flow analysis on non trivial lattices (see for example [1, 7]). This has been extensively studied in particular in the context of partial redundancy elimination. The scope of variable coalescing is usually not so large, and Chaitin proposed the following conservative test: two variables interfere if one is live at a definition point of the other and this definition is not a copy between

the two variables.

This interference notion is the most commonly used, see for example how the interference graph is computed in [3].

Chaitin et al. noticed that, with this conservative interference definition, when a and b are coalesced, the set of interferences of the new variable may be strictly smaller than the union of interferences of a and b. Thus, simply merging the two corresponding nodes in the interference graph is an over-approximation with respect to the interference definition.

For example, in a block with two successive copies $b \leftarrow a$ and $c \leftarrow a$ where a is defined before, and b and c (and possibly a) used after, it is considered that b and c interfere but that none of them interfere with a. However, after coalescing a and b, c should not interfere anymore with the coalesced variable.

Hence, the interference graph has to be updated or rebuilt. Chaitin et al. [15] proposed a counting mechanism, rediscovered in [19], to update the interference graph, but it was considered as too space consuming. Recomputing it from time to time was preferred [15, 14]. Since then, most coalescing techniques based on graph coloring use either live-range intersection graph [31, 13] or Chaitin's interference graph with reconstructions [20, 11].

Actually, in SSA with the dominance property, things are simpler. Each variable has, statically, a **unique** value, given by its unique definition. Furthermore, the "has-the-same-value" binary relation defined on variables is an equivalence relation. This property is used in SSA dominance-based copy folding or global value numbering [10]. The **value** of an equivalence class is the variable whose definition dominates the definitions of all other variables in the class.

Hence, using the same scheme as SSA copy folding or constant propagation, finding the value of a variable can be done by a simple topological traversal of the dominance tree: when reaching an assignment of a variable b, if the operation is a copy $b \leftarrow a$, $V(b)$ is set to $V(a)$, otherwise $V(b)$ is set to b.

The interference test is now both simple and accurate (no need to rebuild/update after a coalescing): a interfere with b if live(a) intersects live(b) and $V(a) \neq V(b)$. (The first part reduces to def(a) $\in$ live(b) or def(b) $\in$ live(a) thanks to the dominance property [13].)

3.6 Conclusion

In this chapter we have presented three contributions, based on properties brought by the SSA form. First we described our novel liveness algorithm, which constructs live-sets for each basic block based on the structure of the loops. The more classical liveness algorithm, based on a data-flow approach, visits every edge and iterates multiple times until no further change appears. On the contrary, our algorithm first visits every edge once, and then visits every node a second time.

Then, based on the insight that many modifications of the code invalidates the liveness-sets, we propose an alternative approach: the liveness checking. Instead of building the set of live variables, we pre-compute some data structure, and only answer to the question "is the variable live at this program point?". The precomputed data structure only depends on the shape of the control flow graph, this means that most modifications of the code (inserting new instructions for example) will not invalidate the data.

Finally, liveness is often used to compute the interference between variables. But interference does not consist of checking if the live-ranges intersect, it also depends if the variables hold the same value. Chaitin gave an ultimate notion of interference, and provided an approximation. We give an algorithm which provides a more precise approximation and which does not need complex updates when the code changes.

Chapter 4

SSA extensions: SSI

Since the inception of the SSA form, many variants have been presented, usually in order to facilitate some analysis. In this chapter we present one of this variant the Static Single Information (SSI) form. Different definitions have appeared in the literature since it was first introduced. Our first contribution is to clarify the differences between those definitions. Then we prove a property related to liveness and the SSI form: the intersection graph of variables under SSI (the intersection of the live-range of the variables) is an interval graph. This allows us to build liveness algorithms which make use of this property.

4.1 Definitions and motivations

The **static single information (SSI) form** is an extension of SSA form that treats uses and definitions symmetrically with respect to one another. We now explore the different definitions found in the literature, by Ananian and by Singer.

4.1.1 Ananian's definition of SSI

Ananian introduced the SSI form in a similar way as Cytron et al. did for SSA, i.e., using properties on paths [2]. For this purpose, we use the notions of split set (similar to join set for SSA) and of upward-exposed use (similar to reaching definition for SSA).

The **split** set of two CFG nodes $u \neq v$, denoted $S(\{u, v\})$, is the set of nodes w such that there exist two paths, one from w to u and one from w to

v, with only w in common. A use of a variable `x` is **upward-exposed** at a program point p if there is a path from p to the use that does not go through any other use of `x`. [1] A procedure satisfies the **single upward-exposed-use property** if at most one use of each variable is upward-exposed at each program point p.

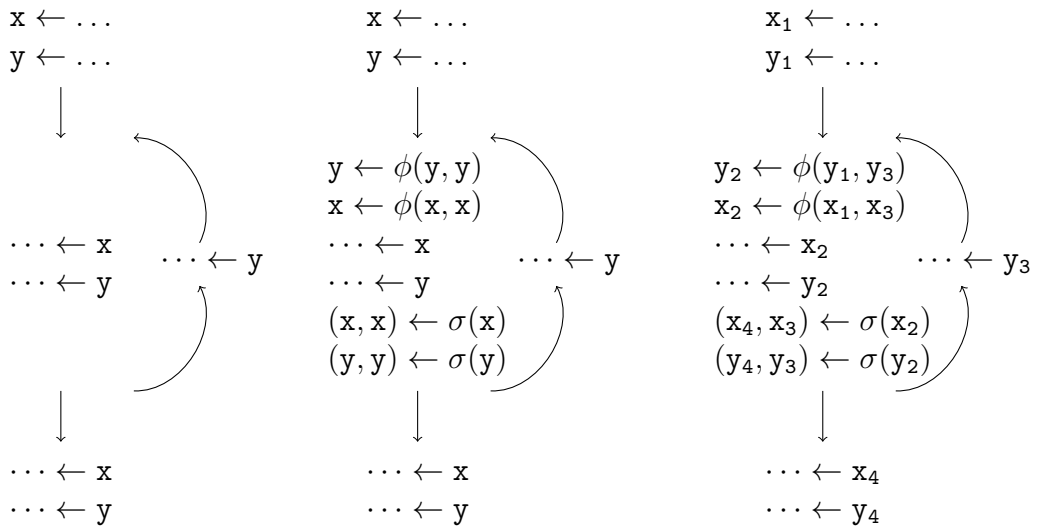
The SSA form construction inserts ϕ-functions at join points to merge multiple variable definitions, thereby satisfying the single reaching-definition property. Similarly, the SSI form construction inserts σ-functions at split points that reach multiple upward-exposed uses, thereby satisfying the single upward-exposed-use property. A σ-function has one argument (a variable use) and it defines as many variables as successors of the split point. Several σ-functions placed at the end of the same block act as parallel statements. To define liveness and dominance, each definition in a σ-function is considered to take place, not at the exit of the block where the flow splits, but on the edge leading to the corresponding successor block, before the any of the ϕ-function related uses. Another simplification would be to assume any critical edge (an edge going from a block with multiple successors, to a block with multiple predecessors) is split, then we could place the definition induced by the σ-function at the entry of the successor basic block, and not on the edge.

Ananian provided a definition of SSI in the spirit of the definition of SSA. Each variable has a **pseudo-use** at the CFG exit node t. A code is in SSI form if it is in minimal SSA form and if it satisfies the single upward-exposed-use property. To satisfy this property for each variable `x`, σ-functions of the form $(\mathtt{x}, \ldots, \mathtt{x}) = \sigma(\mathtt{x})$ can be inserted at the **iterated post-dominance frontier** of the set of uses U_x, denoted $p\mathcal{DF}^+(U_x) = S(U_x \cup \{t\})$. However, as σ-functions create new definitions, more ϕ-functions may be inserted. Then, in a later phase, variables can be renamed so that each variable is defined only once and all uses can be renamed accordingly.

Figure 4.1 illustrates Ananian's SSI definition. The situation is similar for `x` and `y` despite the use of `y` on the back-edge. Two uses are upward-exposed at the end of the central basic block, thus a σ-function is inserted. Now, the central use is reached by two definitions and a ϕ-function is added

[1] Notice that this definition may differ from the one used by liveness analysis. Indeed, for liveness analysis the corresponding path should not contain any definition of the variable. In our context, both definitions of upward-exposed use can be considered, as soon as the code is in SSA form. Any SSI construction algorithm on a non-SSA code may have to cope with subtleties related to this notion of upward-exposed use. This is out of the scope of this paper.

at block entry (see Figure 4.1b). Then, definitions and uses of x and y are renamed, so that each variable has a unique definition and each use refers to the right definition, see the code in Figure 4.1c.



(a) Non-SSI program

(b) With single reaching definitions and single upward-exposed uses

(c) Strong SSI after renaming

Figure 4.1: Placement of ϕ-functions and σ-functions for strong SSI

Ananian's algorithm for SSI construction, however, is not based directly on the iterated dominance and post-dominance frontiers: instead, it uses a data structure called the program structure tree (PST). As the soundness of the PST is questionable (see Section 4.5), the correctness of his construction algorithm is arguable. In this thesis, however, we *do not* question the soundness of Ananian's definition of SSI form. To summarize, a code is in SSI form according to Ananian's definition if it satisfies, before final renaming, the **single reaching-definition** and **single upward-exposed-use** properties, assuming that the CFG has a **pseudo-definition** at the entry node r and a **pseudo-use** at the exit node t for every variable.

4.1.2 Singer's definition of SSI

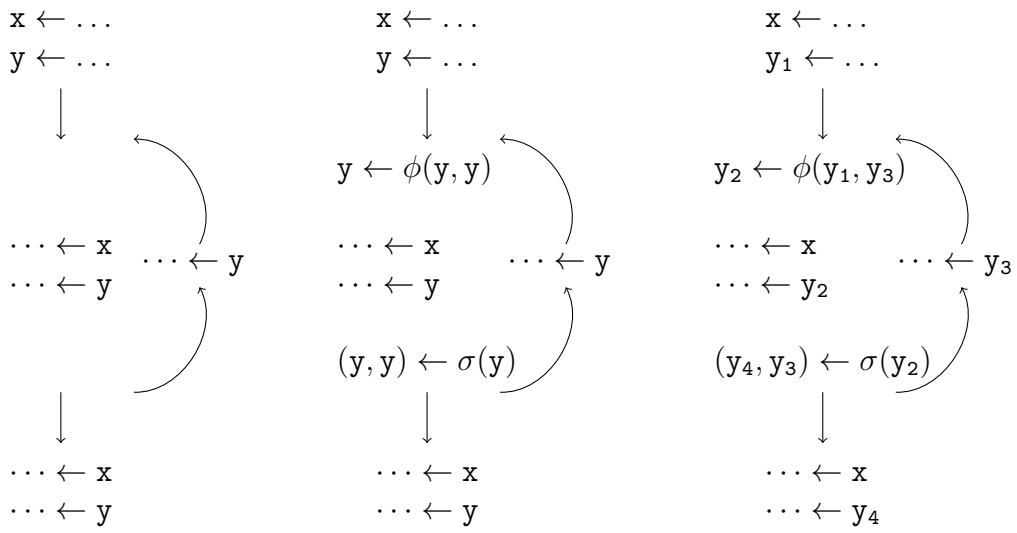
Singer, in his Ph.D. thesis [29], proposed an alternative definition of SSI form: a program is in SSI form if, before renaming (and *a fortiori* after renaming), it satisfies both the **dominance property** and the **post-dominance property**. As we explained in chapter 2, the post-dominance property is the symmetric of the dominance property, inverting the role of uses and definitions. It means that, for each variable, any use that is upward-exposed at a definition of the variable post-dominates this definition (such a use is thus unique). After renaming, as there is a single definition per variable, the (unique) definition of each variable is post-dominated by all its uses, and only one is upward-exposed at the definition.

Singer assumed (incorrectly, as we will show) that this definition was equivalent to Ananian's. It is true that Ananian's definition does satisfy the post-dominance property; however, Singer's definition does *not* imply the single upward-exposed use property for all program points, only for definition points as we explain in Section 4.2.1. Therefore, situations exist where Ananian's definition of SSI form leads to the instantiation of more σ-functions than with Singer's definition. An example is given in Figure 4.2. Now, with Singer's definition, the situation is different for `x` and `y`. The two uses of `x` post-dominate its unique definition, so no σ-function is needed and, consequently, no ϕ-function either. For `y`, the use on the back-edge does not post-dominate the definition of `y`, thus a σ-function is added at the end of the central block, then a ϕ-function at the entry of the basic block (see Figure 4.2b). Then, definitions and uses of `y` are renamed, leading to the code of Figure 4.2c.

4.1.3 Semi-pruned and pruned SSI form

Similar in principle to SSA form, semi-pruned, and pruned variants of SSI form can be defined. It is worth saying that the pseudo-use considered by Ananian's definition is only used to guide the placement of σ-functions during SSI construction. It does not alter the live-range of a variable, as doing so would cause all variables defined locally in the procedure to interfere at the exit point, which is not, in fact, the case.

For the sake of simplicity, we assume pruned SSI form, unless stated otherwise.



(a) Non-SSI program

(b) With dominance and post-dominance properties

(c) Weak SSI after renaming

Figure 4.2: Placement of ϕ-functions and σ-functions for weak SSI

4.2 Weak and strong SSI forms

Now that we have shown the different definitions, we differentiate them in the following way:

- **strong** SSI denotes the SSI form defined by Ananian, i.e., with dominance property, a pseudo-use for each variable, and the single upward-exposed use property,

- **weak** SSI denotes the SSI form defined by Singer, i.e., with dominance property and post-dominance property.

Now we will discuss the differences and common properties of these two SSI definitions. First we will highlight the way the two forms differ, in particular with variables used around loops. Then we will prove some useful lemmas, common to both forms. We will use them as the base for the liveness-related theorems in the next section.

4.2.1 Weak and strong SSI forms are not equivalent

Figures 4.1 and 4.2 illustrate the difference between the two definitions of SSI form. Both definitions insert one ϕ-function and one σ-function for variable `y`; the difference is observed with respect to variable `x`. As we just discussed, the different key properties for inserting ϕ-functions and σ-functions are linked as follows:

1. the single reaching-definition property, with pseudo-definitions, is equivalent to the dominance property;

2. the single upward-exposed-use property, with pseudo-uses, implies the post-dominance property; this is not an equivalence as illustrated in Figures 4.1 and 4.2.

The subtle difference between the single upward-exposed use property and the post-dominance property is the following. If the post-dominance property holds, then, before renaming, every use of a variable that is upward-exposed at a definition of the variable is required to post-dominate this definition. As the post-dominance relation forms a tree, this implies that, for each definition, there is a single upward-exposed use (possibly the pseudo-use, unless pruned SSI is considered). On the contrary, if the single upward-exposed use property is satisfied, then for *any* program point p, not necessarily a definition, there is a single upward-exposed use u. This implies that p is post-dominated by u, otherwise there is a path from p to u and a path from p to the pseudo-use at the CFG exit node t, not containing u, and the single upward-exposed use property is not satisfied for p.

Thus, a procedure that has been converted to strong SSI form also satisfies the criteria for weak SSI form. Strong SSI enforces the single upward-exposed use property for all program points, pruned SSI for the entire live-range, and weak SSI only for all definitions. As our example illustrates, these conditions are not equivalent. For SSA, the situation is different as enforcing the single reaching-definition property for all uses is equivalent to enforcing it for the entire live-range. To summarize, after renaming, for both weak and strong SSI forms, the unique definition of a variable dominates all its uses and each use post-dominates its definition. In (pruned) strong SSI, an additional property is true: considering that each variable has a pseudo-use at the CFG exit node, then, for each program point of the live-range, there is a unique upward-exposed use.

4.2.2 Properties of variables in weak and strong SSI forms

Variables in SSI form satisfy certain properties, common to weak and strong SSI forms, which we will exploit in our proofs regarding the structure of live-ranges.

Let `x` be a variable in (strong or weak) SSI form and let d be its (unique) definition point. In (strong or weak) SSI form, each use of `x` post-dominates d and the post-dominance relation forms a tree. Therefore, all uses belong to a path, in the post-dominator tree, from t, the CFG exit node, to d in the post-dominator tree (note that paths in the post-dominator tree take the reverse direction from the CFG, e.g., each CFG node is reachable from t). Therefore, one of these uses post-dominates all other uses. We call this use the **last use** of `x`, and denote it by u.

Let LIVE denote the live-range of `x`, i.e., the set of program points where `x` is live. By definition, for any strict program, $p \in$ LIVE if there is a path, in the CFG, from p to some use of `x` that does not go through d. Because all uses of `x` post-dominate d, and u post-dominates all other uses of `x`, the previous path can be extended to form a path from p to u that does not contain d. In other words, under both strong and weak SSI, LIVE is the set of points p such that there is a path from p to the last use u that does not contain d.

The next three lemmas hold for any variable `x`, in either weak or strong SSI. Lemma 5 states that if `x` is live at a node p of the post-dominance tree, then it is live at all descendants of p that are not descendants of d, i.e., in the whole subtree rooted at p minus the subtree rooted at d.

Lemma 5. *If $p \in$ LIVE, p pdom q, and $\neg d$ pdom q, then $q \in$ LIVE.*

Proof. Since $\neg d$ pdom q, there exists a path, in the CFG, from q to t not containing d. Because p pdom q, this path must contain p. Thus, there must exist a sub-path from q to p that does not contain d. Since $p \in$ LIVE, there exists a path from p to u that does not contain d. By concatenating these paths, we can construct a path from q to u that does not contain d; it follows that $q \in$ LIVE. □

Lemma 6 states that if `x` is live at a node p of the post-dominance tree, then it is live at all ancestors of p that are not ancestors of u, i.e., all nodes in the path of the post-dominance tree leading to p and starting from, but excluding, the least common ancestor of p and u.

Lemma 6. *If $p \in LIVE$, q* pdom *p, and $\neg q$* pdom *u, then $q \in$* LIVE.

Proof. Since $p \in$ LIVE, there is a path, in the CFG, from p to u that does not contain d. Because $\neg q$ pdom u, there exists a path from u to t that does not contain q. Concatenating the two paths yields a path from p to t via u, such that the sub-path from u to t does not contain q, and the sub-path from p to u does not contain d. Since q pdom p, this path from p to t must contain q, thus q must be in the sub-path from p to u that does not contain d. This establishes the existence of a path from q to u that does not contain d, from which follows $q \in$ LIVE. □

Lemma 7 states that if `x` is live at a node p of the post-dominance tree, then the ancestors of d that are not ancestors of p dominate p, i.e., p is dominated by all nodes in the path of the post-dominance tree leading to d and starting from, but excluding, the least common ancestor of p and d.

Lemma 7. *If $p \in$* LIVE*, q* pdom *d, and $\neg q$* pdom *p, then q dom p.*

Proof. Consider a path, in the CFG, from r to p. As $p \in$ LIVE, d dom p, thus this path contains d. Since $\neg q$ pdom p, there exists a path from p to t without q. Since q pdom d, q must be included in the path from d to p, otherwise we get a path from d to t without q. It follows that q dom p. □

4.3 The intersection graph is an interval graph

This section proves that, for a procedure in either strong or weak SSI form, the interference graph (that is, the intersection graph of the live-ranges, see Section 3.5) is an interval graph. More precisely, we prove that the nodes of a CFG can be totally ordered so that the live-range of each SSI variable corresponds to an interval in this linearized representation. Under both strong and weak SSI form, the start point of the interval is the definition of the variable. Under strong SSI form, the end point of the interval is a use of the variable, while this is not always true in weak SSI.

Our linearization order is based on the post-dominator tree, the dominance relation, and, for weak SSI, a connected minimal loop nesting forest, as described in Section 2.2.

4.3.1 Strong SSI form

The total order of blocks we provide in Section 4.3.2 for weak SSI form is of course also suitable for strong SSI form. Nevertheless, as the situation is simpler for strong SSI form, we prefer to define in this section a suitable order of blocks that is more intuitive and easier to build.

First, we compute a **dominance-based order**, i.e., a partial order of blocks that respects the dominance relation. In such an order, if a node u strictly dominates a node v, then u appears before v. Such an order can be computed by a topological traversal of the dominator tree. It can also be obtained by a preorder or a reverse postorder of any **depth-first search (DFS)** of the CFG.

Second, a preorder DFS traversal of the post-dominator tree is performed with one additional constraint on the order in which the child nodes are visited. Let u and v be two children of a given node in the post-dominator tree, then the following must hold:

1. **Constraint on dominance:** if u dom v, then the preorder DFS traversal of the post-dominator tree should visit the subtree rooted at v before it visits the subtree rooted at u.

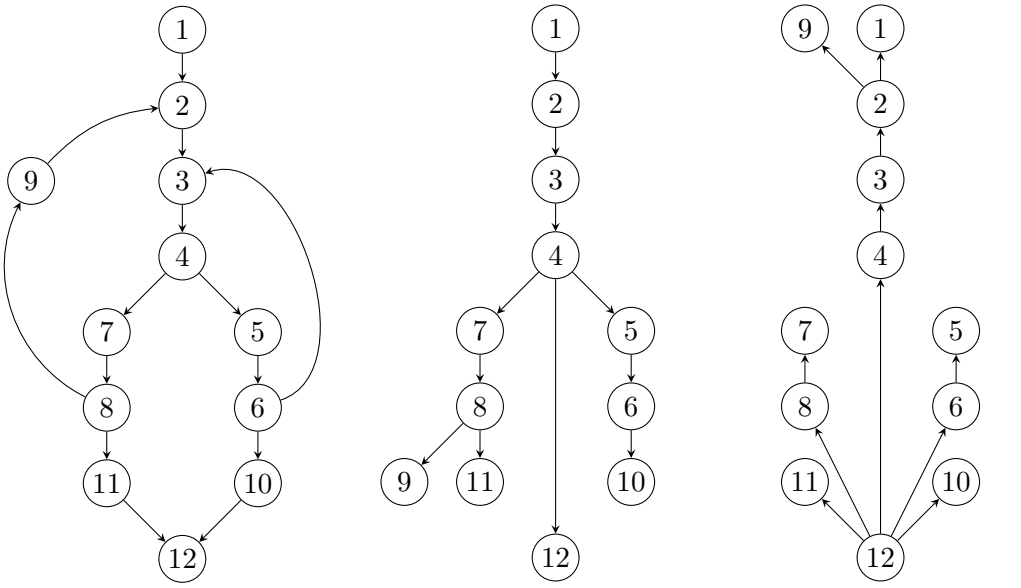
For that, it suffices to visit the children in the reverse order of the dominance-based order computed in the preceding step. The resulting linearization of the CFG nodes is called a **reverse strong interval order**. It starts at the CFG exit node and ends at the CFG entry node. The third and final step is to reverse the reverse strong interval order, yielding a **strong interval order**.

Note that an order of nodes (i.e., of basic blocks) extends directly to an order of program points by traversing all program points within a block from entry to exit.

Consider the CFG of Figure 4.3a, whose corresponding dominator and post-dominator trees are shown in Figures 4.3b and 4.3c respectively. To find a dominance-based order, we can compute a DFS of the CFG considering the nodes in preorder:

$$[1, 2, 3, 4, 7, 8, 9, 11, 12, 5, 6, 10]$$

Next, we perform a preorder DFS traversal of the post-dominator tree, shown in Figure 4.3c. The children are visited with a priority given by the reverse dominance-based order, yielding a reverse strong interval ordering. For example, nodes 8 and 11 are both children of node 12 in the post-dominator



(a) Control-flow graph (b) Dominator tree (c) Post-dominator tree

Figure 4.3: Example of control-flow graph with its associated dominator and post-dominator trees.

tree. Since node 8 dominates node 11, as shown in Figure 4.3b, we must visit node 11 before node 8 during the preorder traversal of the post-dominator tree. Similarly, we must also visit node 8 before node 4. One possible reverse strong interval order that results from this traversal is the following:

$$[12, 10, 6, 5, 11, 8, 7, 4, 3, 2, 9, 1]$$

Reversing this order yields the following strong interval order:

$$[1, 9, 2, 3, 4, 7, 8, 11, 5, 6, 10, 12]$$

Now, we move on to the proof itself. We prove that, if blocks (and thus program points) are (totally) ordered according to a strong interval order, then the live-range of each variable under strong SSI form is an interval whose start point is the variable definition and whose end point is its **last use** (as defined in Section 4.2.2).

First, we prove that, for any variable, any strong interval order visits the node (and thus the program point) that contains its definition before all other nodes where it is live. This result holds for both strong and weak SSI form and will be used in the proofs for both of these representations. In other words, the definition point will be the start point of the interval.

Theorem 7. *Given a reverse strong interval order of the nodes in a CFG in either strong or weak SSI form, the program point that contains the definition of a given variable is always visited after all other program points where it is live.*

Proof. We use the same notations as in Section 4.2.2: `x` denotes the variable of interest, d its definition point, u its last use, and LIVE its live-range. Let $p \in$ LIVE.

We first exclude the trivial case where d is post-dominated by p in which case, in a DFS of the post-dominator tree, d is always considered after p, in particular in a reverse strong interval order. Now suppose that d is not post-dominated by p. We prove that p is not post-dominated by d either. $p \in$ LIVE implies the existence of a d-free path from p to u. As u post-dominates d, there is a d-free path from u to t. These two paths can be concatenated to provide a d-free path from p to t, which proves that p is not post-dominated by d.

Since d and p are not comparable for the post-dominance, we can define $q = \text{LCA}(d, p)$, $q \neq d$, $q \neq p$, to be the least common ancestor of d and p

in the post-dominance tree. Let p' and d' be the corresponding children of q. More formally, $\text{ipdom}(p') = \text{ipdom}(d') = q$, d' pdom d, and p' pdom p. Now, $\neg p'$ pdom d, by construction of p', and u pdom d, by definition of SSI. It follows that $\neg p'$ pdom u. Since p' pdom p, by construction of p', and $p \in$ LIVE, Lemma 6 ensures that $p' \in$ LIVE. Finally, because d' pdom d, by construction of d', and $\neg d'$ pdom p', by construction of d' and p', Lemma 7 proves that d' dom p'. It follows that d' and its descendant d is visited after p' and its descendant p in a reverse strong interval order. This is due to the criterion for choosing children during the preorder DFS of the post-dominator tree: the reverse of the dominance order. □

The next theorem characterizes the live-range of a variable in strong SSI as the descendants, in the post-dominator tree, of its last use, minus the descendants of its definition.

Theorem 8. *In strong SSI form, the live-range of a variable is the set of program points post-dominated by its last use but not post-dominated by its definition.*

Proof. Again, let `x` denote the variable of interest, d its definition point, u its last use, and LIVE its live-range. Recall that d dom u and u pdom d.

First, we show that u pdom p for every point $p \in$ LIVE. Assume to the contrary that there exists a path from p to t that does not go through u. Since $p \in$ LIVE, there is also a path from p to u. The existence of these two different paths contradicts the single upward-exposed-use property of strong SSI form, for at least one point, for example the point where they split. Therefore, the complete live-range is included in the set of descendants of u in the post-dominator tree. Conversely, Lemma 5 shows that any descendant of u that is not a descendant of d belongs to LIVE. Finally, if p is a descendant of d, all paths in the CFG from p to u contain d, thus p cannot belong to LIVE. □

This yields our main result, in the case of strong SSI.

Theorem 9. *In a strong interval order, the live-range of a variable in strong SSI form corresponds to an interval. Moreover, this interval starts at the definition point of the variable and ends at its last use.*

Proof. This follows from Theorems 7 and 8. In a reverse strong interval order, which is a preorder DFS of the post-dominator tree, the last use of

a variable is encountered before all the points it post-dominates. Then, the subtrees rooted at its children are completely traversed, one after the other, the last one being the subtree that contains the definition of the variable, as Theorem 7 shows. Furthermore, according to Theorem 8, all points traversed this way before reaching the variable definition correspond exactly to the live-range of the variable. □

Corollary 1. *For a procedure in strong SSI form, the interference graph, i.e., the intersection graph of the live-ranges, is an interval graph.*

Proof. This follows immediately from Theorem 9. □

4.3.2 Weak SSI form

For weak SSI form, the general scheme of block linearization requires both a post-dominator tree and a connected minimal loop nesting forest. The construction of the order is similar to the strong interval order: first build an auxiliary order, then do a preorder DFS traversal of the post-dominator tree using the previous order, and finally reverse it. The difference is that, during the preorder DFS traversal of the post-dominator tree in the second step, one additional constraint is imposed on the order in which children of a given node are processed. Let u, v, and w be children of a given node in the post-dominator tree. The two constraints that the order must now satisfy are as follows:

1. **Constraint on dominance:** If u dom v, then the preorder DFS traversal of the post-dominator tree should visit the subtree rooted at v before it visits the subtree rooted at u.

2. **Constraint on the loop nesting forest:** if u and v belong to the same loop, and w does not, then the subtree rooted at w should not be visited between the subtrees rooted at u and v.

A total order resulting from such a traversal is called a **reverse weak interval order** and its reversal is called a **weak interval order**. Next, we prove that such an order satisfying both constraints exists.

Lemma 1 shows that any topological order of $\mathcal{F}_{\mathcal{L}}(G)$ respects the dominance if $\mathcal{L}$ is a connected minimal loop forest. In other words, when defining a reverse weak interval order, the two constraints used to decide which subtree to traverse next are not contradictory. If the second constraint (the

constraint on the loop nesting forest) is satisfied, the first one (the constraint on dominance) will be automatically satisfied. In particular, we call the order given by [27, Theorem 4] a **loop-dominance-based order**.

Consider, the example of Figure 4.4. As explained in Section 2.2, the CFG depicted in Figure 4.4a has two nested loops: $L_1 = \{2, 3, 4, 5, 6, 7, 8, 9\}$ and $L_2 = \{3, 4, 5, 6\}$. Due to the edges between the different strongly connected components encountered during the construction of the loop forest and the nesting of loops, this CFG has only two possible loop-dominance-based orders:

$$[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12] \qquad \text{(or 10 and 11 inverted)}$$

Now, we perform a preorder DFS traversal of the post-dominator tree (shown in Figure 4.3c) where the children of each node are visited according to the reverse loop-dominance-based order; in particular, node 8 must be visited before node 6. This provides us with the following reverse weak interval order:

$$[12, 11, 10, 8, 7, 6, 5, 4, 3, 2, 9, 1]$$

Reversing this order yields the following weak interval order:

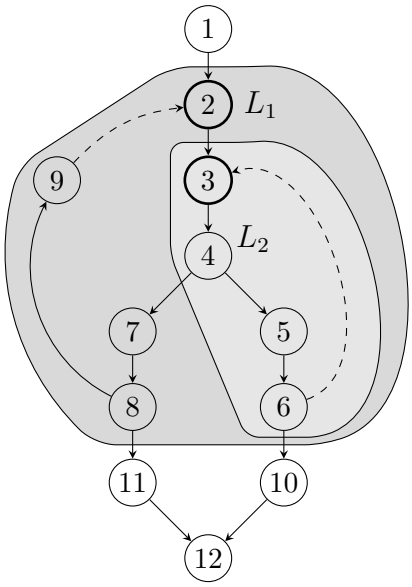
$$[1, 9, 2, 3, 4, 5, 6, 7, 8, 10, 11, 12]$$

for which, as we will show, the live-range of any variable in weak SSI is an interval.

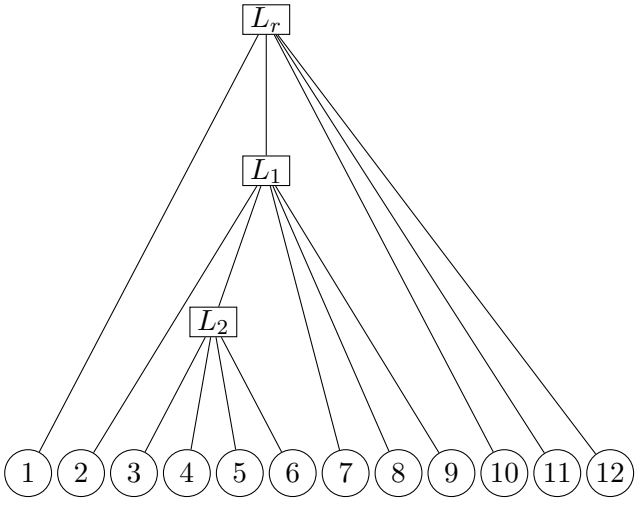
Theorem 10. *In a weak interval order, the live-range of a variable in weak SSI form corresponds to an interval.*

Proof. Again, let `x` denote the variable of interest, d its definition point, u its last use, and LIVE its live-range. Let $\leq$ denote a weak interval order. By Theorem 7, we already know that $d \leq p$ for any program point $p \in$ LIVE. To establish that the live-range corresponds to an interval, it remains to prove that $q \in$ LIVE and $d < p < q$ imply $p \in$ LIVE, for any two points p and q.

The order $\leq$ respects the post-dominance, as it is based on a preorder DFS traversal of the post-dominator tree; thus $\neg d$ pdom p and $\neg p$ pdom q. If q pdom d, then q pdom p due to the preorder DFS traversal of the post-dominator tree, and $p \in$ LIVE by Lemma 5. Similarly, u pdom p implies $p \in$ LIVE by Lemma 5 (recall that u pdom d, $u \in$ LIVE, and $\neg d$ pdom p). Therefore, we can assume that $\neg u$ pdom p and $\neg q$ pdom d.



(a) CFG with loops (shaded regions), loop-edges (dotted lines), and loop headers (bold nodes)



(b) Loop forest with an additional root node

Figure 4.4: Example of control-flow graph with an associated minimal loop forest (tree).

Recall that LCA(u, v) denotes the least-common ancestor of u and v in the post-dominator tree. Since $p < q$, the subtree rooted at q in the post-dominator tree is visited before the subtree rooted at p, from which it follows that LCA(d, q) pdom LCA(d, p). Let $z =$ LCA(d, p). Let us define d', p', and q' such that d' pdom d, p' pdom p, q' pdom q, and ipdom$(d') =$ ipdom$(p') =$ ipdom$(q') = z$. Since $\neg u$ pdom p, it follows that $\neg u$ pdom z. Since z spdom d, by definition, and u pdom d, by the definition of weak SSI form, it follows that u must be on a path in the post-dominator tree from z to d, and therefore d' pdom u. Figure 4.5 illustrates the post-dominance relationship between the various nodes (or program points) that we just introduced. In this figure, paths in the post-dominator tree are traversed from left-to-right when computing a weak interval order.

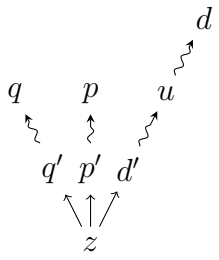


Figure 4.5: Post-dominance relation between the program points used in the proof

Since q' and u are in separate branches of the post-dominator tree, it follows that $\neg q'$ pdom u. Since $q \in$ LIVE, Lemma 6 shows that $q' \in$ LIVE. Then, using Lemma 7, we deduce that d' dom q' and u dom q'. As the dominance relation forms a tree, either u dom d' or d' dom u. If d' dom u, there is an elementary path from r to q' that goes through d' and then through u, before reaching q'. This establishes the existence of a path from u to q' that does not contain d'. Moreover, since $\neg d'$ pdom q', there exists a path from q' to t that does not contain d', which contradicts the fact that d' pdom u. Therefore, u dom d'.

Since $q' \in$ LIVE and $q' \neq d$, there is a path from q' to u such that d dominates each node on this path. The facts that d sdom u, u dom d', and d' dom q' imply that there exists a path from u to q' that includes d', and d dominates all points on this path. This, in turn, implies that d', q', and u all belong to a cycle containing nodes that are dominated by d. Let L denote

the loop in the loop nesting forest such that the removal of the loop-edges of L breaks this circuit. At least one node on this circuit must be a loop-header for L. Therefore, $d \notin L$, since d dominates each node on the circuit, and a loop-header cannot be dominated by any other node within the loop when it is a loop-entry node.

Now, recall that d', p', and q' are all children of z in the post-dominator tree, and that the preorder DFS traversal of the dominator tree, which constructs the reverse weak interval ordering, processes the subtrees rooted at the children of z in reverse loop-dominance-based order. Specifically, p' must occur between q' and d' in the reverse loop-dominance-based order. The fact that the reverse loop-dominance-based order respects the loop nesting property implies that $p' \in L$. Therefore, there exists a path from p' to u in L, thus it does not contain d, which proves that $p' \in$ LIVE. Finally, Lemma 5 shows that $p \in$ LIVE too. □

Theorem 10, therefore, proves that the live-range of each variable under weak SSI corresponds to an interval in a total order of CFG nodes. Moreover, this order depends only on the structure of the CFG and not on the relative positions of definitions and uses of variables within the basic blocks of the CFG. This order can thus be pre-computed to help with liveness analysis, as described in the following section, and it remains unchanged even if various transformations move or delete instructions, as long as the CFG structure is not modified. Note however that some forms of code motion may require the insertion of additional ϕ-functions and σ-functions in order to preserve weak SSI form.

Corollary 2. *For a procedure in weak SSI form, the interference graph, i.e., the intersection graph of the live-ranges, is an interval graph.*

Proof. This follows immediately from Theorem 10. □

The primary difference between strong and weak SSI is the "end" of the interval corresponding to the live-range of each variable. Under strong SSI, the end of the interval is the last use of the variable and it post-dominates all points of the live-range. Under weak SSI, the end of the interval may not be a use and it does not necessarily post-dominate all points of the live-range. Figure 4.3 illustrates these points: if a variable is defined in node 1 and used only in node 3, weak SSI does not require the insertion of any ϕ-functions or σ-functions: the initial live-range is not split and the variable is live

everywhere in the two loops, i.e., its live-range spans all CFG nodes 1 through 9. In particular, there is no point in the live-range that post-dominates the entire live-range. The strong interval order proposed in Section 4.3.1 would be incorrect here because 11 is between 1 and 6, but the variable is not live in 11. With the weak interval order proposed in Section 4.3.2 for weak SSI, the live-range is an interval, its end point is node 8, and 8 does not post-dominate 6.

4.4 Liveness under SSI

Since we proved our results on the properties of live-range in the previous section, we can now find out what the consequences are in term of liveness for variables under SSI. The live-range of variables all have a particular shape, this property leads to a much simpler liveness analysis.

In principle, it should be possible to convert minimal or semi-pruned SSI form to pruned SSI form using dead code elimination. This ensures that all σ-functions and ϕ-functions corresponding to a variable `x` in the pre-SSI program are placed at points where `x` was originally live, and liveness analysis is not required to ensure this property. We may still require liveness analysis in SSI form, either to perform register allocation or to eliminate as many copies as possible during SSI elimination. One possibility is to compute liveness analysis prior to building SSI form, and then updating the results to account for the fact that the conversion to SSI partitions each pre-SSI variable into a set of smaller variables; the other possibility is to perform liveness analysis once the procedure has been converted to SSI form.

This section points out a consequence of Theorem 9: under strong SSI form, iterative data-flow analysis is not required. Instead, it suffices to traverse the nodes of the CFG in reverse strong interval order and, within each node, to traverse the instructions in reverse order. During this traversal, a variable is live exactly from the first time one of its uses is encountered (which is guaranteed to be the last use) until its definition is processed. The consequences of this approach are twofold:

1. If live sets are explicitly required, neither data-flow equations nor union of sets are needed throughout the traversal: the live-out set of a basic block is exactly the live-in set of the previously-processed block. Dataflow analysis is reduced to its simplest form: a single pass over the control-flow graph is enough.

2. Register allocators based on **linear scan** [26] approximate the lifetime of each variable as an interval, rather than an exact live-range. An SSI-based implementation of a linear scan register allocator can compute live intervals exactly without the need to compute live sets explicitly.

This fast approach to liveness analysis does *not* directly work for weak SSI form since there may be no last use that post-dominates each point where a variable is live. If the basic algorithm outlined above is applied, a point where a variable becomes live may be encountered using the reverse weak interval order before the last use is encountered. Instead, we need to identify, for each variable, the last point (in the order) of the largest loop in the loop nesting forest that contains its last use but not its definition. This will be the end of the interval, as the proof of Theorem 10 shows. Be careful, this property is not due to the fact that the order of blocks respects the nesting of the loops as this is not true in general for a weak or strong interval order. If this last point is pre-computed, the liveness analysis can be performed in a unique pass, as for strong SSI, following the reverse weak interval order. Another solution is to first compute, for each variable, an incomplete interval from its definition to its last use, then to extend it in a second pass, following the weak interval order and exploiting the structure of the loop forest.

Once liveness analysis has been performed, the interval property implies a simple $O(1)$-time query to determine whether a variable `x` is live at a given point p in a procedure. Under strong SSI form, the endpoints of the interval are the definition point d and the final use u of `x`; therefore, `x` is live at p if and only if $d \leq p \leq u$ using a strong interval order. Under weak SSI, one endpoint of the live interval is d, and liveness analysis must be run in advance to determine the other endpoint, which we denote as q. Therefore, `x` is live at a point p if and only if $d \leq p \leq q$ using a weak interval order. Similarly, computing live-in and live-out sets can be done in one pass, once these intervals are identified.

4.5 Single-entry single-exit region and SSI

We will now clarify some of the mistakes made during the creation of SSI and found in subsequent uses of the SSI form.

When Ananian first presented SSI, his motivation was the simplification of backward data-flow algorithms. Because he would split the live-ranges at the post-dominance frontier, a backward data-flow analysis, which would

only gather information form uses and definition would become sparse: the information can be attached to the variables.

But the subsequent papers, who used SSI as a basis, do not use this property. Most people mistakenly thought that SSI would split at branch points and could be used to gather precise information from conditional.

What SSI effectively does is splitting the live range of a variable so that more contextual information can be attached to the variable name. But depending of the information that is wanted, more or less splits are needed. We can differentiate between several cases:

- information propagates backward and only change where uses are placed
- information propagates forward and can change even if there is no uses of the variable, for example at branch points where a related variable is checked

In the first case, the necessary split points are at the post-dominance frontier, that is the strong SSI form. In the second case, more splits can be needed. If the information that is computed is needed not only at the use and definition points, but at every point of the live-range, then a split after every branch point where information is gained and the variable is live is needed. That is the way e-SSA, a variant of the SSA form presented by Bodik et al. [5], splits the live-ranges to help solve the array bound checking problem. Otherwise if the information is only needed at use-def points, a form with less splits can be used.

Weiss [32] gives a definition of what he considers *relevant* program points: the set of branch point relevant to the execution of the program point x is the set of branch points such that there exists a path from the branch point to the exit node, and another path to x, and both paths are disjoint except for the branch point. Furthermore, as we recalled earlier, Weiss proved that the iterated set $S^+(x, exit)$, is equal to the non iterated set $S(x, exit)$, and that this set is equal to the post-dominance frontier of x. This means that whenever two nodes share the same set of relevant branch point, the execution of one node implies the execution of the other, and any information gained from a branch point is valid for both points.

This notion of relevance and the implied splitting at the relevant points is in fact the form defined by Ananian's in his pessimistic algorithm. Similarly to the dependence flow graph by Johnson et al. [23], his algorithm would split whenever the use points and the definition point would be in different

single-entry single-exit regions. Single-entry single exit regions are related to the notion of relevant points: two nodes share the same set of relevant branch points iff they are contained in the same set of SESE regions.

4.5.1 Single-entry single-exit region

Ananian's algorithm for (strong) SSI construction [2] is based on an auxiliary data structure called the **program structure tree (PST)** [22]. The PST decomposes the control flow graph into a hierarchy of canonical **single-entry single-exit (SESE) regions**. Extending the notion of dominance and post-dominance to edges instead of just nodes, a SESE region is formed by a pair of CFG edges (e_i, e_j) if e_i dom e_j, e_j pdom e_i, and e_i and e_j are **loop-cycle equivalent**, meaning that every cycle in the CFG that contains e_i also contains e_j, and vice-versa.

A SESE region (e_i, e_j) is defined to be **canonical** if e_j dom e_k for any SESE region (e_i, e_k) and e_i pdom e_k for any SESE region (e_k, e_j). The hierarchy of canonical SESE regions is based on the following definition of containment: SESE region (e_i, e_j) **contains** a basic block b if e_i dom b and e_j pdom b.

This notion of containment does not match the intuitive definition of SESE regions, we assume that a SESE region is a connected region of the control flow graph, while Johnson et al. definition does not ensure that property. Figure 4.6 shows an example where a SESE region is not a connected subgraph.

Moreover using their definition of SESE region, the following was stated a theorem:

> If R_1 and R_2 are two canonical SESE regions of a CFG, then either R_1 and R_2 are node-disjoint, or R_1 is contained within R_2 (or vice-versa).

But as shown in our example, (A, B) and (B, C) are both canonical SESE regions. However, both contain the basic block b, as per the definition of containment provided above. Therefore, it is impossible to construct a hierarchy of SESE regions, given this definition of containment.

The theorem itself contains three parts, and parts 1 and 3 wrongly assume that "an edge cannot both dominate and post-dominate a node". In the example of Figure 4.6, the three edges A, B, and C both dominate and post-dominate the node (i.e., basic block) b. Actually, reasoning with nodes (and not edges), it is not possible to construct an order of the nodes of a CFG

that respects both the dominance and post-dominance relations, i.e., an order denoted by $\leq$ such that $u \leq v$ if u dom v and $u \leq v$ if v pdom u. Such an order cannot exist because the header node of a for-loop or a while-loop may simultaneously dominates and post-dominates every other node in the loop, as shown in Figure 4.7. Thus, it is impossible to construct an order under which the header occurs both before and after each node in the loop.

The following definition for SESE would make the theorem on program structure tree hierarchy correct:

Definition 3. *A SESE region is a connected subgraph of $G = (V, E)$, such that there exists a unique incoming edge from outside the region, and a unique outgoing edge to outside the region.*

This definition directly matches the intuitive concept of single-entry single-exit, furthermore we can show that a SESE region defined this way is also a SESE region as defined by Johnson et al.

Proof. The incoming edge obviously dominates any node from the region (and thus the outgoing edge), similarly the outgoing edge post-dominates every node from the region (and the incoming edge). Let us show that our two distinguished (incoming and outgoing) edges are cycle equivalent. Given a cycle containing the incoming edge. Since the source of the incoming edge is not in the SESE region but its target is, there exists an edge from the cycle such that its source is part of the SESE region, but its target is not. But our definition of SESE states that this edges is unique, it is the outgoing edge of the SESE. Similarly we can show that every cycle containing the outgoing edge must contain the incoming edge. □

Similarly given two edges following Johnson's definition, we can show they are the incoming and outgoing edges of a SESE region matching our definition: we only need to show that there is a subgraph of the control flow graph such that they are incoming and outgoing edges.

Given an SESE region with edges (e_1, e_2), and the set of nodes such that:

- they are dominated by e_1, and not dominated by e_2
- they are post-dominated by e_2, and not post-dominated by e_1

Let us show this set form a connected subgraph.

Proof. We note $e_1 = (x_1, x_2)$ and $e_2 = (y_1, y_2)$. Take (a, b) an edge such that b is in the set and suppose (a, b) is different from e_1.

First we prove that e_1 dominates a and e_2 does not dominate a. Since e_1 dominates b, any path from the start node to b contains e_1, furthermore, since $(a, b) \neq e_1$, the same holds from a. This means e_1 dominates a. Since e_1 does not post-dominates b, there exists a (cycle-free) path denoted p_1 from b to the end node that does not contain e_1. Since e_2 post-dominate b, p_1 contains e_2. Given any cycle-free path (denoted p_2) from start to a, it contains e_1. Suppose p_2 contains e_2 as well, then the concatenation of p_1 and p_2 contains a cycle (since e_2 is contained in both sub-paths). Since e_1 and e_2 are cycle equivalent, e_1 can be found between this two occurrences of e_2. Furthermore since, e_1 is not in p_2, it must be in p_1, after e_2. But e_1 dominates e_2, so e_2 most be preceded by e_1 in p_1, this mean we would have a cycle in p_1, a contradiction. So e_2 is not contained in p_1, and e_2 does not dominate a.

Now we prove that e_2 post-dominates a while e_1 does not.

Given any path from a to the end. Since p_2 is a path from start to a that does not contain e_2 and contains e_1, and since e_2 post-dominates e_1, e_2 must be found in the path. This means e_2 post-dominates a. Since e_1 does not post-dominate b, there is a path, from b to the end, that does not contain e_1. Since $(a, b) \neq e_1$, it means e_1 does not post-dominates a as well. This proves that a is in the set.

We thus have shown the every predecessor from a node in the set is either the x_1 (if $e_1 = (a, b)$ or another node from the set. Reciprocally we can prove that the successor is either y_2 or part of the set.

This proves the set is connected. □

4.5.2 Removal of σ-functions

Since SSI and related forms are only concerned about live-range splitting, in practice it means that σ-functions are not required, only a way to split a live-range across an edge is needed. This is something that SSA and ϕ-functions already provide: every σ-function can be replaced by a corresponding ϕ-function in each successor block.

This simplification makes the issue of removing the σ-functions void, since they are not introduced in the first place. If instead we want to keep the σ-functions, then, to go out of the SSI form, we can first do remove all σ-functions with a simple copy-propagation. Using either of the schemes,

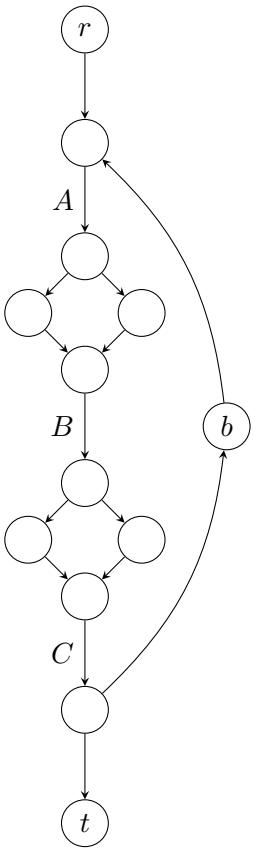


Figure 4.6: (A, B) and (B, C) are canonical SESE that have a partial overlap: both contain block b.

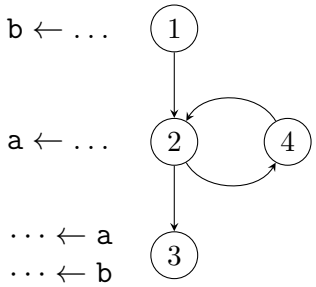


Figure 4.7: Node 2 simultaneously dominates and post-dominates node 4.

once we removed the σ-functions, going out of SSI is now the same problem as going out of SSA which we will explore in the next chapter.

Chapter 5

SSA Destruction

5.1 Motivations

A naive SSA destruction increases even more the number of new variables and the code size, because ϕ-functions are replaced by variables and copies from or to these variables. A solution, especially useful for JIT compilation, is to introduce copies on the fly, during the SSA destruction, and only when needed, as in Method III of Sreedhar et al. [31]. Also, to eliminate copies or to avoid introducing them, some interference information, and thus some liveness information, is required. Thanks to the dominance property in SSA, one can avoid building explicitly the interference graph. Also, thanks to the notion of dominance forest, Budimlić et al. [13] proposed a SSA destruction mechanism, more suitable for JIT compilation, as it reduces the number of interference tests.

In light of previous work, our primary goal was to design a new SSA destruction, suitable for JIT compilation, thus focused on **speed** and memory footprint, as previous approaches were not fully satisfactory. However, to make this possible, we had to revisit the way the SSA destruction is conceptually modeled. We then realized that our framework not only can target speed, but also addresses the problem of **correctness** and of **efficiency** (i.e., quality of the resulting code).

Several papers have already addressed the SSA destruction problem. We propose a new method because, first, we want to rely on a provably-correct method, generic, simple to implement, without special cases and patches, and in which correctness and quality of the result (i.e., optimization) are

conceptually separated. And secondly, we want to develop a technique that can be fast and not too memory-consuming, without compromising correctness and quality. Let us first go back to previous approaches.

SSA destruction was first mentioned in the seminal paper by Cytron et al. [18, Page 478]:

> "Naively, a k-input ϕ-function at entrance of a node X can be replaced by k ordinary assignments, one at the end of each control flow predecessor of X. This is always correct, but these ordinary statements sometimes perform a good deal of useless work. If the naive replacement is preceded by dead code elimination and then followed by coloring, however, the resulting code is efficient".

In other words, copies are placed in predecessor blocks to emulate the ϕ-function semantics and Chaitin-style coalescing [15] (as in register allocation) is used to remove some of them.

Although this naive translation seems, at first sight, correct, it can lead to subtle errors as pointed by Briggs et al. [9, Page 880] because of parallel copies and/or critical edges in the control flow graph. Two typical situations are identified, the "lost copy problem" and the "swap problem", some patches are proposed to handle them correctly, and a "more complicated algorithm that includes liveness analysis and a preorder walk over the dominator tree" is quickly presented for the general cases, but with neither a discussion of complexity, nor a correctness proof. Nevertheless, according to the authors, this solution "cures the problems that [they] have seen in practice" [9, Page 879].

The first solution, both simple and correct, was proposed by Sreedhar et al. [31]. In addition to the copies at the end of each control flow predecessor, the trick is to insert another copy at the entry of the basic block of the ϕ-function. This simple mechanism, detailed hereafter, is sufficient to make the translation always correct, except for special cases that we point out. Several strategies are then proposed to introduce as few copies as possible, including a special rule to eliminate more copies than with standard coalescing and so that "copies that it places cannot be eliminated by the standard interference graph based coalescing algorithm" [31, Page 196]. This last (also unproved) claim turns out to be correct, but only for the very particular way copies are inserted, i.e., always after the previously-inserted copies in the same block. Also, the way coalescing is handled is again more a patch, driven by implementation considerations, than a conceptual choice. We will come back

to this point later. Nevertheless, our technique is largely inspired by the various algorithms of Sreedhar et al.

In other words, these previous approaches face some conceptual subtleties that make them sometimes incorrect, incomplete, overly pessimistic, or too expensive. This is mostly due to the fact that a clean definition of interference for variables involved in a ϕ-function is lacking, while it is needed both for correctness (for adding necessary copies) and for optimization (for coalescing useless copies). Our first contribution, beyond algorithmic improvements, is to address this key point. Thanks to this interference definition, we develop a conceptually SSA destruction framework, in which correctness and optimization are not intermixed. The resulting implementation is much simpler, with no special cases, and we can even develop fast algorithms for each independent phase, without compromising the quality of results. The rest of this section gives an overview of our technique, before diving into more detailed explanations.

5.2 Clean approach

Consider a ϕ-function $a_0 \leftarrow \phi(a_1, \ldots, a_n)$ placed at entry of a block B_0: a_0 takes the value of a_i if the control-flow comes from the i-th predecessor block of B_0. If a_0, ..., a_n can be given the same name without changing the semantics of the program, the ϕ-function can be eliminated. When this property is true, the SSA form is said to be conventional (CSSA) [31]. This is not always the case: after copy propagation or code motion, some of the a_i may "interfere" with each other (this SSA form is sometimes called transformed SSA form, TSSA). The technique of Sreedhar et al. [31] consists in three steps:

1. translate SSA into CSSA, thanks to the introduction of copies;
2. eliminate redundant copies;
3. eliminate ϕ-functions and leave CSSA.

In their first method (Method I), the translation into CSSA is done as follows. For each ϕ-function $a_0 \leftarrow \phi(a_1, \ldots, a_n)$ at entry of block B_0:

- $n+1$ new variables a'_0, ..., a'_n are introduced;

- a copy $a'_i \leftarrow a_i$ is placed at the end of B_i, the i-th predecessor block of B_0;
- a copy $a_0 \leftarrow a'_0$ is placed at entry of the block B_0 (after ϕ-functions);
- the ϕ-function $a_0 \leftarrow \phi(a_1, \ldots, a_n)$ is replaced by $a'_0 \leftarrow \phi(a'_1, \ldots, a'_n)$.

If a block contains several ϕ-functions, then several copies needs to be introduced. They are introduced at the same place and they should be viewed as parallel copies. This is what we do here, as Leung and George in [25]. However, as far as correctness is concerned, they can be sequentialized in any order, as they concern different variables. This is what Sreedhar et al. do in all their methods.

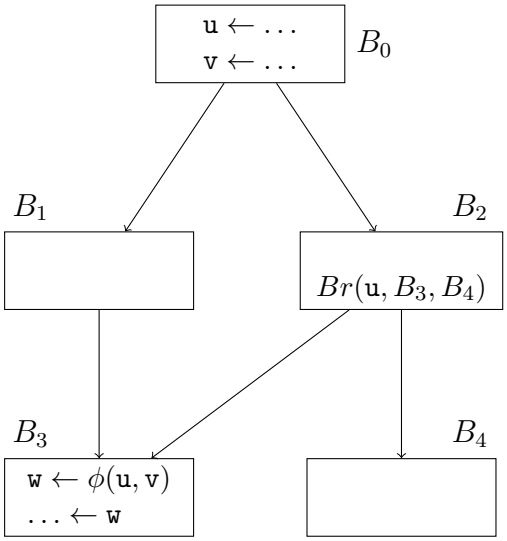
Lemma 8. *After the introduction of the new variables a'_i for all ϕ-functions and the corresponding copies, the code is in CSSA form. In other words, replacing all variables a'_i by a new unique variable for each ϕ and removing all ϕ-functions is a correct SSA destruction.*

Proof. First note that, after introduction of the copies, the code semantics is preserved. The variables a_i are copied into the variables a'_i, then fed into the new ϕ-function to create a'_0, which is finally copied into a_0. To show that the code is in CSSA, note that the variables a'_i have very short live ranges. The variables a'_i, for $i > 0$, are defined at the very end of disjoint blocks B_i, thus none is live at the definition of another: they do not interfere. The same is true for a'_0 whose live range is located at the very beginning of B_0, even if B_0 may be equal to B_i for some i. Thus, the $n + 1$ variables a'_i can share the same variable name, as they are never simultaneously live on a given execution path. □

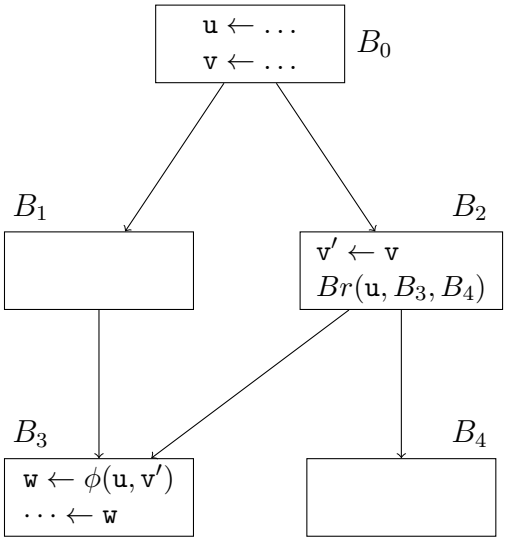
It is now clear why the proposal of Cytron et al. was wrong. Without the copy from a'_0 to a_0, the ϕ-function defines directly a_0. The live range of a_0 can be long enough to intersect with on of the a'_i, $i > 0$: it is the case if a_0 is live-out of the block B_i where a'_i is defined. Two cases are possible: either a_0 is used in a successor of $B_i \neq B_0$, in which case the edge from B_i to B_0 is *critical* (as in the "lost copy problem"), or a_0 is used in B_0 as a ϕ-function argument (as in the "swap problem"). In the latter case, if parallel copies are used, a_0 is dead before a'_i is defined but, if copies are without taking care of potential conflicts, the live range of a_0 can go beyond the definition point of a'_i and lead to incorrect code after renaming a_0 and a'_i with the same name.

As Lemma 8 explains, splitting the definition of the ϕ-function itself with one additional variable at the block entry solves the problem. In the methods of Sreedhar et al., the decision to keep the copy a'_i (in Method I) or to insert it (in Method III) depends on the intersection of the live range of a'_i with the live-out set of the block B_i. But this techniques hides a subtlety: what does "at the end of basic block" mean? Depending on the branching instruction, the copies cannot always be inserted at the very end of the block, i.e., after all variables uses and definitions. For example, for a ϕ-function after a conditional branch that uses a variable u, the copies must be inserted *before* the use of u. Since any new copy that is inserted might interfere with u, the intersection check have to be done with u as well, otherwise some incorrect code can be generated. Consider the SSA code in Figure 5.1(a), which is not under CSSA form. As u is not live-out of block B_2, the optimized algorithm (Method III) of Sreedhar et al. considers that, in order to transform the code into CSSA form, it is sufficient to insert a copy v' of v at the end of B_2. But the copy is inserted before the use of u (Figure 5.1(b)) and the code is still not CSSA since u and v' interfere. Removing the ϕ-function, i.e., giving the same name to w, u, and v' leads to the incorrect code of Figure 5.1(c). This problem, never mentioned in the literature, needs to be solved for implementors. Fortunately, it is easy to correct by considering the intersection with the set of variables live *just after the point of copy insertion*, in our case the live-out set with the addition of u, instead of simply the live-out set of the block.

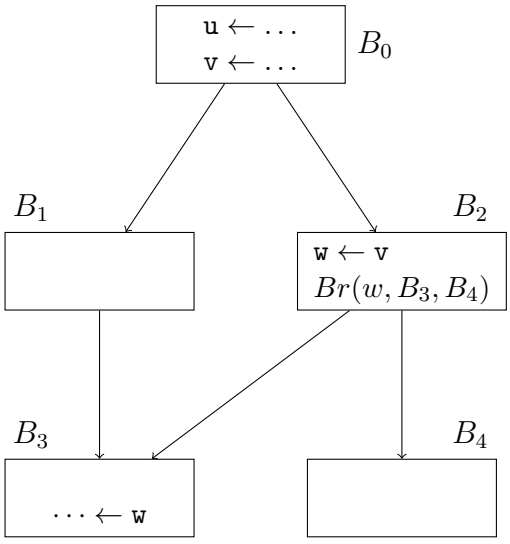
An even more subtle case can happen, when the basic block contains variables *defined after* the point of copy insertion. This configuration is possible in embedded environments, where some DSP-like branching instructions have a behavior similar to a hardware loop: in addition to the condition, a counter `u` is decremented by the instruction itself. If `u` is used in a ϕ-function in a direct successor block, no copy insertion can split its live range. In this case, it must then be given the same name as the variable defined by the ϕ-function. If both variables interfere, we cannot use the same name for both variables. In order to solve the problem, SSA could be used with more care inside the compiler: either we avoid promoting the variable to SSA, some instruction can be changed, or the control-flow edge can be split. While it might not happen in general purpose computer, this point has never been mentioned before: SSA destruction by copy insertion alone is not always possible, in presence of complex branching instructions. For example, suppose that for the code of Figure 5.2(a), the instruction selection chooses a branch with decrement (denoted Br_dec) for Block B_1 (Figure 5.2(c)). Then, the



(a) Initial SSA code



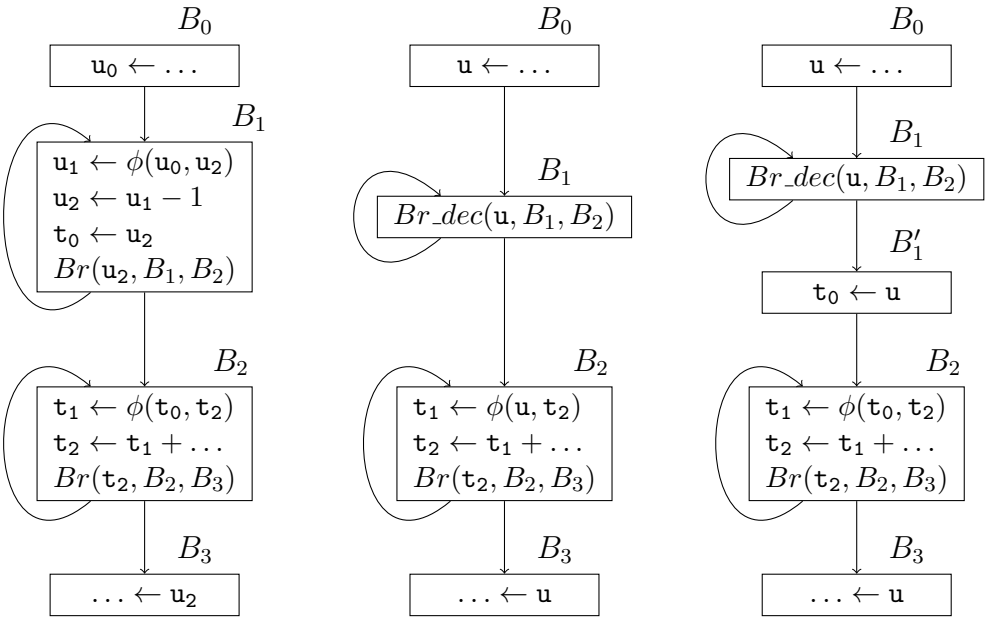
(b) Insufficient copy insertion



(c) Incorrect SSA destruction

Figure 5.1: Considering liveout sets may not be enough.

ϕ-function of Block B_2, which uses $\mathtt{u}$, cannot be removed by standard copy insertion since $\mathtt{u}$ interferes with $\mathtt{t}_1$ and its live range cannot be split. To go out of SSA, one could add $\mathtt{t}_1 = \mathtt{u} - 1$ in Block B_1 to anticipate the branch, or one could split the critical edge between B_1 and B_2 as in Figure 5.2(c). In other words, simple copy insertions as in the model of Sreedhar et al. is not enough in this particular case.



(a) Initial SSA code (b) Branch with decrement (Br_dec) (c) CSSA with an additional edge splitting

Figure 5.2: Copy insertion may not be sufficient.

These different situations illustrate again why SSA destruction must be analyzed with care, and explain the reasons why correctness should be addressed before even thinking of code optimization. Indeed, with aggressive SSA optimizations, going out of SSA form can become bug-prone and complicated.

5.2.1 Going out of CSSA: a coalescing problem

Once the copies are inserted, the code is in CSSA, as explained in Lemma 8, except for the special cases of branch with definition, described above. Then,

going out of CSSA is straightforward: all variables involved in a ϕ-function can be given the same name and the ϕ-functions can be removed. This solves the correctness aspect. To improve the code however, it is important to remove as many useless copies as possible. This can be treated as a classical coalescing problem, as in register allocation. The difference resides in the complexity: with Method I, the number of new copies and variables can be very large, which can be too costly, especially if a interference graph is used. Sreedhar et al. propose several improvements. The first proposed improvement consists of the introduction of copies only when variables interfere and the conservative update of the interference graph (Method II). A second technique consists of a more involved algorithm that also uses and updates liveness information (Method III). Finally a special "SSA based coalescing" can be used, that is useful to complement Method I and Method II but is useless after Method III. Those improved techniques all rely on the explicit representation of *congruence classes* partitioning the program variables into the sets of variables that are already coalesced together.

Sreedhar et al. use special coalescing rules depending on the method, but this approach can be simplified. Actually, once the code is in CSSA, the optimization problem is a standard aggressive coalescing problem (i.e., with no constraints on the number of target variables) and we should be able to use standard heuristics for this NP-complete problem [28, 8]. The fact that the code is in SSA does not make it simpler or special. Also, Method III turns out to give better results than Method I followed by coalescing, even though Method III was primarily designed for speed. This result can be explained with the fact that Sreedhar et al. rely on a conservative definition of interferences in order to decide if two variables can be given be same name (coalesced). As we have discussed in Section 3.5, we can easily exploit the SSA properties to identify when two variables have the same **value**: in this case, they do not interfere even if their live ranges intersect. Then, with this intrinsic definition of interferences, there is no point to compare, in terms of quality of results, a method that introduces all copies first as in Method I or on the fly as in Method III: they are equivalent. Furthermore, since our definition of interferences is more accurate, more copies can be removed.

Another weakness in the model of Sreedhar et al. is the lack of flexibility in the placement of copies, they are inserted in a sequential order at the end or entry of a block. We prefer to stick to the SSA semantics, i.e., to rely on parallel copies (all sources of every copy are read before any write occurs). We then sequentialize these copies afterward, once we know which copies

remain. The interest is two-fold. First, with sequential copies, some additional interferences between the corresponding variables appear. This gives less freedom to the coalescer, especially if we take into account additional register constraints. Second, with parallel copies, we avoid a tricky update of liveness information. Our approach allows us to handle all copies in a uniform way, this is fundamental to minimize the engineering effort.

In conclusion, with a more accurate definition of interferences, the use of parallel copies, and a generic coalescing algorithm to eliminate copies, we achieve our goal: a conceptually simple approach, provably correct, in which correctness and optimization are separated. This is of important for implementing SSA in an industrial compiler in order to avoid complex bugs. We can now give an overview of the general process before detailing each individual step.

5.2.2 Overview of the SSA destruction

To summarize, conceptually, our SSA destruction process has five successive phases:

1. Insert parallel copies for all ϕ-functions using in Method I of Sreedhar et al. and coalesce all a'_i together.
2. Build the interference graph with an accurate definition of interference, using the "SSA value" of variables..
3. Perform aggressive coalescing, possibly with renaming constraints.
4. Go out of CSSA by giving the same name to all coalesced variables.
5. Sequentialize parallel copies, possibly with one more variable and some additional copies, when swaps are needed.

Step 4 is straightforward. We already explained Step 1 in Section 5.2. Step 2 was detailed in previous chapter, in Section 3.5). Our accurate notion of interference allows us to avoid rebuilding or updating the interference graph, furthermore there is no need to rebuild or update the interference graph, no need to develop a special SSA-based coalescing algorithm as in [31], and no need to make a distinction between variables which can be coalesced with Chaitin's approach or not. The only important information is the fact that they interfere or not.

We will now describe Step 3 (Section 5.2.3), and Step 5 (Section 5.2.4).

Since correctness (Step 1) and optimization (Step 3) are independent, we can develop algorithms to make the whole process fast enough for just-in-time compilation. This will be explained in Section 5.3: fast live-range intersection test (Section 5.3.1), fast interference test and node merging (Section 5.3.2), "virtualization" of initial copy insertion (Section 5.3.3), i.e., copy insertion on the fly similar to Method III of Sreedhar et al. These techniques allow us to skip the construction of the liveness information and the interference graph, and help us lowering the memory footprint.

5.2.3 Coalescing

As previously explained, the SSA destruction is nothing but a traditional aggressive coalescing problem with no constraints on the number of colors. If all copies are initially inserted, as in Method I of Sreedhar et al., any sophisticated technique can be used. In particular, it is possible to use weights to first examine copies placed in inner loops: this approach reduces the number of static and dynamically-executed copies. Contrary to Sreedhar et al. which do not use weights, we use a classical profile information to get basic blocks frequency. However this weight may be slightly under-estimated: if a swap is needed in order to sequentialize a parallel copy, additional copy will be required (see Section 5.2.4).

We call *virtualization* the process of inserting copies on the fly, similarly to the Method III of Sreedhar et al. With this technique, the copy variables a'_i are created only when needed, but all the reasoning is done as if they were inserted. To make this possible, we need to consider ϕ-functions in a certain order, this will have an impact on the coalescing. Also, Sreedhar et al. use a deferred copy insertion mechanism that, even if not expressed in these terms, amounts to build some maximal independent set of variables (i.e., the set of variables that do not interfere), which are then coalesced. This approach gives indeed slightly better results than reasoning one copy at a time. In our virtualized version, we also process one ϕ-function at a time, and simply consider its related copies by decreasing weight.

We also address the problem of "copy sharing". Consider again the example of two successive copies $b = a$ and $c = a$. Thanks to our definition of value, the fact that b is live at the definition of c does not necessarily imply that b and c interfere. Suppose however that a interferes with b and c for some other reason. In this case, no coalescing would occur although coalescing b and

c would save one copy, by "sharing" the copy of a. Leung et al. [25] provide examples ABI constraints where this situation occurs. This sharing problem is difficult to model and optimize, in fact deciding the insertion point of the copies is even harder, but we can still provide some optimizations. We can coalesce two variables b and c if they are both copies of the same variable a and if their live ranges intersect. We restrict ourself to intersecting live-ranges because if they are disjoint, the optimization could increase the live range of the dominating variable, thus possibly creating some interference not taken into account. Section 5.2.5 mentions this important post-optimization, which is a benefit provided by our value-based interference definition.

5.2.4 Sequentialization of parallel copies

In every steps of the algorithm, we treat the copies placed by Step 1 at the end or entry of a given block as **parallel copies**. This indeed matches closely the semantics of ϕ-functions. As we explained previously, this gives us several benefits: a simpler implementation, in particular for defining and updating liveness information, a more symmetrical code, and fewer constraints for the coalescer. However, since generic parallel copies are not part of any instruction set, at the end of the process, we need to go back to standard code, i.e., write the final copies in a sequential order. In most cases, we simply need to order the copies in the right order, but sometimes an new variable, additional copies are required. We describe this sequentialization algorithm here.

Conceptually, the algorithm is very simple but deriving a fast implementation is more complicated. Consider the directed graph G whose vertices are the variables involved in the parallel copy and with an edge from a to b whenever there is a copy from a to b (we write $a \mapsto b$). This graph has the following key property: each vertex has a unique incoming edge, the copy that defines it (a parallel copy $(b \mapsto a, c \mapsto a)$ is possible but only if $V(b) = V(c)$ in which case one of the copies can be removed). Thus, G has a particular structure: each connected component is a circuit (possibly reduced to one vertex) and each vertex of the circuit can be the root of a directed tree. A graph with this property is called a windmill graph in the literature. The copies of the tree edges can be sequentialized starting from the leaves, copying a variable to its successors before overwriting it with its final value. Once these tree copies are scheduled, it remains to consider the circuit copies. If at least one vertex of the circuit was the root of a tree, it has already been copied somewhere, otherwise, we copy one of the circuit vertices into a new variable.

Then, the copies of the circuit can be sequentialized, starting with the copy into this "saved" vertex and back along the circuit edges. The last copy is done by moving the saved value in the right variable. Thus, we generate the same number of copies as expressed by the parallel copy, except possibly one additional copy for each circuit with no tree edge, i.e., no duplication of variable. For example, for the parallel copy $(a \mapsto b, b \mapsto c, c \mapsto a, c \mapsto d)$, there is one circuit (a, b, c) and an edge from c to d, so we generate the copies $d = c$, $c = a$, $a = b$, and $b = d$ (and not $b = c$).

The algorithm emulates a traversal of G (without building it), allowing to overwrite a variable as soon as it is saved in some other variable. When a variable a is copied in a variable b, the algorithm remembers b as the last location where the initial value of a is available. This information is stored into `resource`(a). The initial value that must be copied into b is stored in `pred`(b). The initialization consists in identifying the variables whose values are not needed (tree leaves), which are stored in the list `available`. The list `to_do` contains the destination of all copies to be treated. Copies are first treated by considering leaves (while loop on the list `available`). Then, the `to_do` list is considered, ignoring copies that have already been treated, possibly breaking a circuit with no duplication, thanks to an extra copy into the fresh variable n.

5.2.5 Qualitative experiments

The experiments were done on the integer subset of the SpecC2000 benchmarks compiled at aggressive optimization level. The compiler used was an Open64-based production compiler whose code generator exploits the SSA form at machine level. Renaming register constraints, such as calling conventions or dedicated registers, are also treated with preliminary copy insertions and coalesced as copies due to ϕ-functions.

First, we evaluated how the accuracy of the interference impacts the quality of coalescing. We implemented seven variants of coalescing. Below $a \mapsto b$ is a copy to be removed and X and Y are the congruence classes (see Section 5.2.1) of a and b that will be merged, in case of coalescing, into a larger class.

Intersect X and Y can be coalesced if there is no variables $x \in X$ and $y \in Y$ whose live ranges intersect.

Algorithm 13 Parallel copy sequentialization algorithm

```
1:  function SEQUENTIALIZE(parallelCopies P)
2:      available ← []
3:      to_do ← []
4:      for each (a ↦ b) ∈ P do
5:          resource(b) ← nil
6:      for each (a ↦ b) ∈ P do
7:          resource(a) ← a
8:          pred(b) ← a
9:          to_do.push(b)
10:     for each (a ↦ b) ∈ P do
11:         if resource(b) = nil then
12:             available.push(b)
13:     while to_do ≠ [] do
14:         while available ≠ [] do
15:             b ← available.pop()
16:             a ← resource(pred(b))
17:             emit_copy(a ↦ b)
18:             resource(pred(b)) ← b
19:             if pred(b) = a then
20:                 available.push(a)
21:         b ← to_do.pop()
22:         if b ≠ resource(pred(b)) then
23:             emit_copy(b ↦ n)
24:             resource(b) ← n
25:             available.push(b)
```

Sreedhar I This is the coalescing complementing Method I of Sreedhar et al. X and Y can be coalesced if there is no pair of variables $(x, y) \in (X \times Y) \setminus \{(a, b)\}$ whose live ranges intersect: (a, b) is not checked as a and b have the same value.

Chaitin X and Y can be coalesced if there is no variables $x \in X$ and $y \in Y$ such that x and y interfere following Chaitin's definition, i.e., x is live at the definition of y and this definition is not a copy $x \mapsto y$ (or the converse).

Value X and Y can be coalesced if there is no variables $x \in X$ and $y \in Y$ such that x and y interfere following our value-based interference definition, i.e., their live ranges intersect and have a different value, as explained in Section 3.5.

Sreedhar III This is the virtualization mechanism used in Method III of Sreedhar et al. Copies are inserted, considering one ϕ-function at a time, as explained in Section 5.2.3. We added the SSA-based coalescing of Method I, which is, as pointed out by Sreedhar et al., useless for ϕ-related copies, but not for copies due to register renaming constraints.

Value + IS This is Value, extended with a quick search for an independent set of variables, for each ϕ, as in Sreedhar III.

Sharing This is Value + IS, followed by our copy sharing mechanism, see Section 5.2.3. If c is live just after the copy $a \mapsto b$ and $V(c) = V(a)$, i.e., a and c have the same value, then, denoting Z the congruence class of c, 1) if $Y = Z$ and $Y \neq X$, the copy $a \mapsto b$ is redundant and can be removed; 2) if X, Y, and Z are all different, and if Y and Z can be coalesced (following the Value rule), the copy $a \mapsto b$ can be removed after coalescing Y and Z as c has already the right value.

Figure 5.3 gives, for each variant, the ratio of number of remaining static copies compared to the less accurate technique (Intersect). Comparing the cost of remaining "dynamic" copies, computed with a static estimate of the basic block frequencies, gives similar results. The first four variants show what is gained when using a more and more accurate definition of interferences (from Intersect to Value).

It is interesting to note, again, that Sreedhar I is quite inefficient as, for example, it cannot coalesce two congruence classes X and Y if $X \times Y$ contains

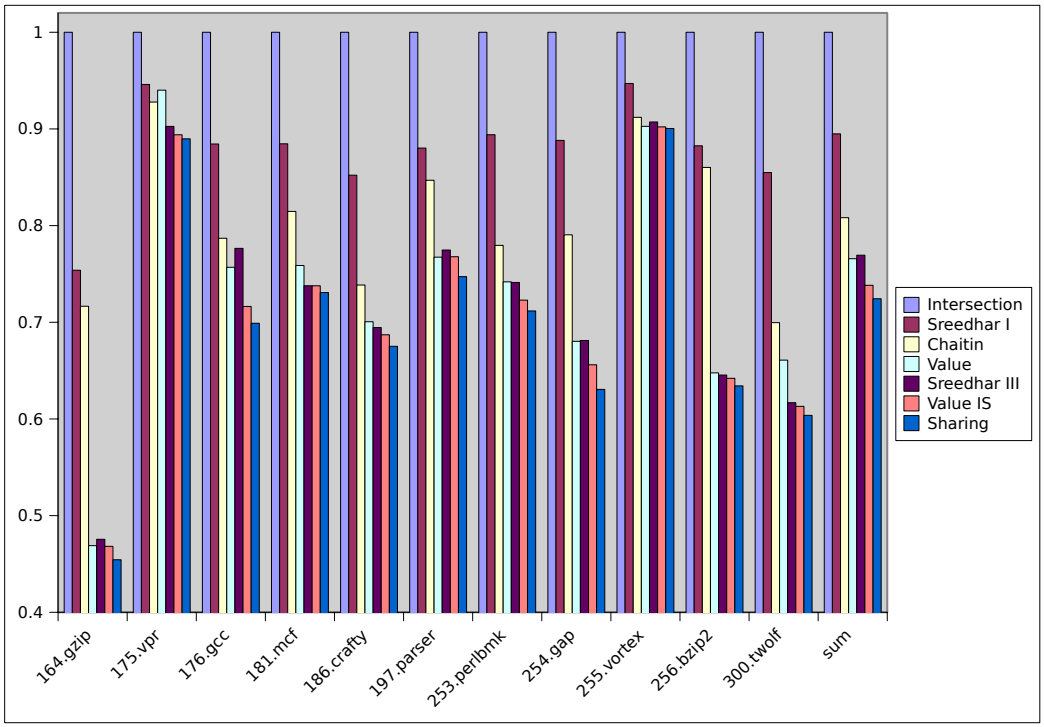


Figure 5.3: Impact of interference accuracy and coalescing strategies on remaining number of moves: number of remaining static copies compared to Intersect.

two pairs of intersecting copy-related variables. Introducing variables on the fly as in Method III avoids this problem as only copies that cannot be removed by the SSA-based coalescing are introduced (but it is not tuned in [31] to optimize weighted moves). Also, the independent set search integrated in this method improves the results compared to Value, which is the basic version with our value-based interference. If this independent set trick is also added to Value, our technique outperforms Sreedhar III (version Value + IS). The last variant, Sharing, shows that we can go even further with our additional sharing mechanism.

These experiments confirm that the decomposition of the problem into the insertion of parallel copies followed by coalescing with an accurate identification of values is sufficient to obtain the best quality so far. In addition, it is also a clean and flexible solution because, with our intrinsic value-based interference definition, the fact that two variables can be coalesced does not

depend on the way we introduce copies, either all before coalescing as in Method I or on the fly as in Method III. With less programming effort, we can do slightly better than Sreedhar III in terms of quality of results. More importantly, since our framework separates the correctness and the quality of results from how coalescing is implemented, we can now concentrate on speed and memory optimizations. These two points are addressed in the next section.

5.3 Towards a more efficient algorithm

Implementing the technique of Section 5.2.2 may be too costly. First, it inserts many instructions later considering them as useless, furthermore the insertion of the copy instructions is expensive as it needs to change the intermediate representation of the code. This process introduces many new variables as well, and the size of the variable universe has an impact on the liveness analysis and the interference graph, slowing down our algorithm. Also, if a general coalescing algorithm is used, a graph representation with adjacency lists (in addition to the bit matrix) and a working graph to explicitly merge nodes when coalescing variables, would be required. Additionally these constructions, updates, manipulations are not only time-consuming but also memory-consuming. We can speed-up the whole process by:

- avoiding the use of a working graph and even of an interference graph, while still relying classical liveness information;
- replacing classical liveness computation by fast liveness checking;
- emulating ("virtualizing") the introduction of all initial copies, as in Method III of Sreedhar et al.

If an interference graph is available, it is not clear that using an additional working graph is more expensive, but, in the context of aggressive coalescing, both Chaitin [14] and Sreedhar et al. [31] preferred not to use one. To get rid of it, Sreedhar et al. manipulate congruence classes: sets of variables that are already coalesced together. Then, two variables can be coalesced if their corresponding congruence classes do not contain any two interfering variables, one in each congruence class. This quadratic number of variable-to-variable interference tests might be expensive. In Section 5.3.2, we address

this problem and propose a linear algorithm for interference detection between two congruence classes.

When an interference graph is not available, it is not possible to directly gather the list of variables that interfere with a given variable. Instead queries are typically restricted to interference *checking*, i.e., existence of an interference. The classical approach consists in computing a full interference relation, but stored only as a bit matrix. This needs a costly *pre-computation* phase: we need to explore the whole program and use live sets to build interferences. After this pre-computation, queries are done in constant time. A second approach performs value tests, dominance checks, and liveness checks, without relying on any pre-computation of live sets. Queries are slightly more time-consuming, but they avoid the use and storage of any interference graph. Section 5.3.1 explains the use of such methods.

Finally, Section 5.3.3 explains how we adapt the virtualization mechanism used in Method III of Sreedhar et al., which inserts copies only when needed, to avoid the introduction of many new variables and of useless copies that will be removed.

5.3.1 Live-range intersection tests

Remember Section 3.5. We need two tests: a test of live range intersection and a test of equality of values. In the next section, we explain how we can check, with this notion of interference, whether two congruence classes (sets of variables already coalesced) **interfere**. Before, we need an algorithm to decide if two live ranges **intersect**. Several methods can be used for testing live-range intersection, either a classical approach building live-in and live-out sets, or a more refined approach using SSA properties, which we have described in chapter 3.

In the next section, we do not care about the algorithm used in practice, they are used as a black box for developing a linear algorithm that checks interference between congruence classes.

5.3.2 Linear interference test between two congruence classes, with extension to equalities

We now develop one of our main contribution: how to efficiently perform an interference test between two sets of already-coalesced variables (congruence

classes in Sreedhar et al. terminology). Suppose that the two tests needed to decide the interference of two SSA variables – the live-range intersection test (Section 5.3.1) and the "has-the-same-value" test (Section 3.5) – are available as black boxes. We can avoid having a quadratic number of tests. For that, we simplify and generalize the dominance-forest technique proposed by Budimlić et al. to check linearly if a set of live ranges is intersection-free or not [13]. Our contributions are:

- we avoid constructing explicitly the dominance forest;
- we are also able to check for interference between *two* sets;
- we extend it to support the notion of equality of value.

Given a set of variables, Budimlić et al. define its *dominance forest* as a graph forest where ancestors of a variable are exactly the variables that dominate it (the dominance between variables corresponds to the dominance of their corresponding definition points). The key idea of their algorithm is that the set contains two intersecting variables if and only if it contains a variable that intersects with its parent in the dominance forest. Then we simply traverse the dominance forest and check every edge for live-range intersection.

Instead of constructing explicitly the dominance forest, we represent each set of merged variables as a list ordered according to a pre-DFS order $\prec$ of the dominance tree (i.e., a depth-first search where each node is ordered before its successors). Then, because querying if a variable is an ancestor of another one can be achieved in constant time (simple dominance test, we only need to check if one is an ancestor of the other in the dominance tree), simulating the stack of a recursive traversal of the dominance forest is straightforward. Thus, as in [13], we can derive a linear-time intersection test for a set of variables, see Algorithm 14.

Suppose now that we have two intersection-free sets (two congruence classes of non-intersecting variables) `blue` and `red`. To coalesce them, we need to check if there is an intersection between both. We can do as if the two sets were merged and then apply the previous technique. However, we can save some intersection tests when we compare two variables originating from the same set: in Line 6 of the previous algorithm, the intersect test should first check if parent and current belong to a different list before running an expensive intersection test. Also, since both sets are represented as an ordered

Algorithm 14 Check intersection in a set of variables

```
1: function SET_INTERSECTION(variables sorted in preorder of the dominance tree)
2:     stack ← []
3:     for each var in variables do
4:         while stack is not empty and not dominate(stack.last, var) do
5:             stack.pop()
6:         if stack is not empty and intersect(var, stack.last) then
7:             return true
8:         stack.push(var)
9:     return false
```

list, traversing the lists in order is straightforward. We only need to use two indices and advance in the right list, according to the pre-DFS order $\prec$ of the dominance tree. Our improved algorithm uses two indices *idx_red* and *idx_blue* and the main loop of the algorithm skips variables from the same set.

Finally, we extend this intersection technique to an interference test that takes equalities into account. Suppose that b is the parent of a in the dominance forest. In the previous algorithm, the induction hypothesis states that the subset of already-visited variables does not contain any intersection. Then, if c has been already visited, the fact that b and a do not intersect guarantees that c and a do not intersect, otherwise the intersection of b and c would have been already noticed. However, for interferences with equalities, this no longer holds. The variable c may intersect b but if they have the same value, they do not interfere. The consequence is that, now, the fact that a and b do not intersect does not guarantee that a and c do not intersect. However, if a does not intersect b and any of the variables that intersect b, we can be sure that a does not intersect any of the already-visited variables. To speed up such a test and to avoid checking intersection between variables in the same set, we keep track of one additional information: for each variable a, we store the nearest ancestor of a that has the same value and that intersects it. We call it the "equal intersecting ancestor" of a. We assume that the equal intersecting ancestor is pre-computed within each set, denoted by `equal_anc_in`(a), and we will compute the equal intersecting ancestor in the opposite set, denoted by `equal_anc_out`(a). We can now give the algorithm for interference test with equality. The skeleton is the same as Algorithm 14, with the patch to progress

Algorithm 15 Check intersection between two sets of intersection-free variables

```
1: function SET_INTERSECTION(list_red, list_blue)
2:     stack ← []
3:     idx_red ← 0
4:     idx_blue ← 0
5:     while idx_red < length(list_red) or idx_blue < length(list_blue) do
6:         advance_blue ← (idx_red = length(list_red)
7:                             or (idx_red < length(list_red)
8:                                 and idx_blue < length(list_blue)
9:                                 and list_blue[idx_blue] ≺ list_red[idx_red]))
10:        if advance_blue then
11:            var ← list_blue[idx_blue]
12:            idx_blue ← idx_blue + 1
13:        else
14:            var ← list_red[idx_red]
15:            idx_red ← idx_red + 1
16:        while stack is not empty and not dominate(stack.last, var) do
17:            stack.pop()
18:        if stack is not empty and intersect(var, stack.last) then
19:            return true
20:        stack.push(var)
21:    return false
```

along the two lists `Red` and `Blue`, and where the call Line 14 is now a call to the **interference** test of Function `interference`. The principles of the algorithm are given in the codes themselves. The use of two equal intersecting ancestors, `in` and `out`, is to make sure that the Function `intersect`(a, b), which runs a possibly expensive intersection test, is performed only if a and b do not belong to the same set.

```
1: function UPDATE_EQUAL_ANC_OUT(variable a, variable b)
2:     tmp ← b
3:     while (tmp ≠ NULL) and (intersect(a, tmp) = false do
4:         tmp ← equal_anc_in(tmp)
5:     equal_anc_out[a] ← tmp
6: function CHAIN_INTERSECT(variable a, variable b)
7:     tmp ← b
8:     while (tmp ≠ NULL) and (intersect(a, tmp) = false do
9:         tmp ← equal_anc_in(tmp)
10:    if tmp = NULL then
11:        return false
12:    else
13:        return true
14: function INTERFERENCE(variable a, variable b)
15:    equal_anc_out(a) = NULL
16:    if b ≠ NULL and (a and b are in the same set) then
17:        b ← equal_anc_out(b)
18:    if b ← NULL then
19:        return false
20:    if value(a) ≠ value(b) then
21:        return chain_intersect(a, b)
22:    else
23:        update_equal_anc_out(a, b)
24:    return false
```

We point out that once a list is empty and the stack does not contain any element of this list, there is no more intersection or updates to make. Thus, the algorithm can be stopped, i.e., the while loop condition in Algorithm 14 can be replaced by:

while ($i_r <$ red.size() and $n_b > 0$) or ($i_b <$ blue.size() and $n_r > 0$) or ($i_r <$

red.size() and $i_b <$ blue.size()) **do**
...

where n_r (resp. n_b) are variables that must be implemented to count the number of elements of the stack that come from the list `Red` (resp. `Blue`).

Finally, if a coalescing of the two sets occurs, it remains to store the two lists as a unique ordered list (in linear time, in a similar joint traversal) and to update the equal intersecting ancestor `equal_anc_in`(a) for the combined set as the maximum (following the pre-DFS order $\prec$) of `equal_anc_in`(a) and `equal_anc_out`(a).

5.3.3 Virtualization of ϕ-nodes

If we implement the whole procedure as described in Section 5.2.2, we start by introducing many new variables a'_i (one for each argument of a ϕ-function, plus its result) and many copies in the block where the ϕ occurs and in its predecessors. These variables can be immediately coalesced together, in what we call a ϕ-node, and stored in a congruence class. But, in the data structures used (interference graph, liveness information, variable name universe, parallel copy instructions, congruence classes), they exist as data items and consume memory and time, even if at the end, after coalescing, they may disappear.

To avoid the introduction of these initial variables and copies, the technique is to emulate the whole process, as does Method III of Sreedhar et al., which introduces necessary copies on the fly, when they appear to be needed. However, in their method, Sreedhar et al. still manipulate an interference graph (between original variables) and, at each step, update liveness information. Before any related copy is inserted, each argument of a ϕ-function is considered to be live-out of the predecessor block it comes from. This is too pessimistic. Also, it is troublesome to maintain the liveness information once copies are inserted: recall the incorrect case of a variable used on a branching operation mentioned in Section 5.2.

Instead, we prefer to use a special location in the code, identified as a "virtual" parallel copy, where the real copies, if any, will be placed. The original arguments of a ϕ-function are then assumed, initially, to have a "use" in the parallel copy but are not considered as live-out along the corresponding control flow edge. Then, the algorithm selects copies to coalesce, following some order, either a real copy or a virtual copy. If it turns out that a virtual copy $a_i \mapsto a'_i$ (resp. $a'_0 \mapsto a_0$) cannot be coalesced, the copy is materialized in

the parallel copy and a'_i (resp. a'_0) becomes explicit in its congruence class. This way, only copies that the first approach would finally leave uncoalesced are introduced. The only key point to make the emulation of copy insertion possible is that one should never have to test an interference with a variable that is not yet materialized or coalesced. For that, ϕ-nodes are treated one by one, and all virtual copies that imply a variable of the ϕ-node are considered (either coalesced or materialized) before examining any other copies. The weakness of this approach is that a global coalescing algorithm cannot be used because only a partial view of the interference graph structure is available during the algorithm. However, the algorithm can still be guided by the weight of copies, i.e., the dynamic count associated to the block where it would be placed if not coalesced. The rest is only a matter of accurate implementation, in particular, it is very important to be careful on how the live ranges of variables are updated, following the ϕ-function semantics.

5.3.4 Results in terms of speed and memory footprint

To measure the potential of our different contributions, in terms of speed-up and memory footprint reduction, we implemented a generic SSA destruction pass that enables us to evaluate different combinations. We selected the following:

`Us I` Simple coalescing without virtualization. Different techniques for checking interferences and liveness are available.

`Sreedhar III` Native implementation of Method III of Sreedhar et al. complemented by their SSA-based coalescing for non ϕ-related copies. Both use an interference graph with a bit-matrix and liveness information with ordered sets.

`Us III` Our implementation of ϕ-nodes coalescing with virtualization followed by coalescing for non ϕ-related copies. This implementation is generic enough to support various options: with parallel or sequential copies, with or without an interference graph, with or without live sets. Hence, its implementation is less tuned than `Sreedhar III`.

By default, `Us III` and `Us I` use an interference graph and classical liveness information. The options are:

`InterCheck` No interference graph: intersections are checked using a dominance test and liveness information as in [13].

`InterCheck+LiveCheck` No interference graph and no live-in/live-out sets: intersections are checked with the fast liveness checking algorithm of [6]. This algorithm, an earlier and more complex variant of the algorithm from 3 trades a faster construction time of the needed data structures for a slower query time.

`Linear+InterCheck+LiveCheck` In addition, the linear intersection check is used instead of the quadratic algorithm.

When an interference graph, liveness sets, or live-check sets are used, timings include their construction. Figure 5.4 shows the timings for those different variants versus `Sreedhar III` as a baseline. As one could expect, `InterCheck` always slows down the execution, while `LiveCheck` and `Linear` always fasten the execution with a significant ratio. A very interesting result is that the simple SSA-based coalescing algorithm without any virtualization is as fast as the complex algorithm with virtualization. Indeed, when using `Linear+InterCheck+LiveCheck`, adding first all copies and corresponding variables before coalescing them, do *not* have the negative impact measured by Sreedhar et al. any longer.

Figure 5.5 shows the memory footprint used for interference and liveness information.

Interference graph is stored using a half-size bit-matrix. `Measured` provides the measured footprint from the statistics provided by our memory allocator. In `Sreedhar III` or `Us III`, variables are added incrementally so the bit-matrix grows dynamically. This leads to a memory footprint slightly higher than for a perfect memory. The behaviour of such a perfect memory is evaluated in `Evaluated` using the formula $\lceil \frac{\#\text{variables}}{8} \rceil \times \#\text{variables}/2$.

Liveness sets are stored as ordered sets. `Measured` provides the measured footprint of the livesets, without counting those used in liveness construction. As for the interference graph, livesets are modified by `Sreedhar III` or `Us III`. But since the number of simultaneous live variables do not change, their size remain roughly the same. Because the use of ordered sets instead of bit-sets is arguable, we evaluated for

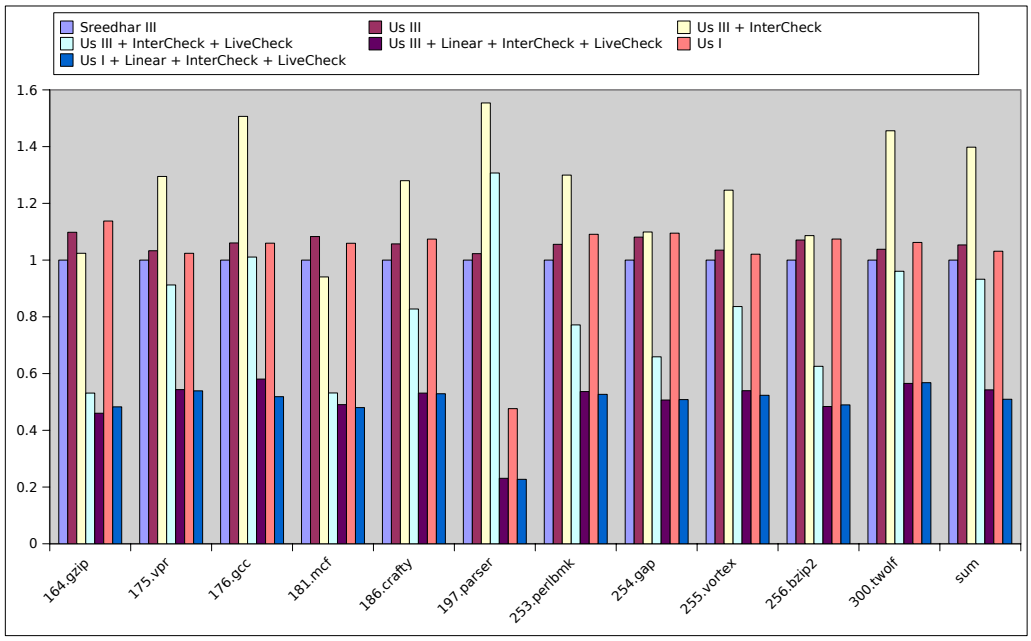


Figure 5.4: Performance results in terms of speed (time to go out of SSA).

a perfect memory the corresponding footprint of live-sets by counting the size of each livesets. For bit-sets, we evaluated using the formula $\lceil \frac{\#\text{variables}}{8} \rceil \times \#\text{basicblocks} \times 2$.

Live check uses 2 bit-sets per basic block. It uses also a few other sets during construction. Those sets are measured in the memory footprint. A perfect memory is evaluated using the formula $\lceil \frac{\#\text{basicblock}}{8} \rceil \times \#\text{basicblock} \times 2$

The results show that the main gain comes from the removal of the interference graph. We should notice that our liveness construction designed for speed but not for memory consumed a huge amount of memory. That is the reason why we removed it from the statistics. We would like to point out that in practice the memory usage for liveness construction is difficult to optimize and might lead to an important part of the memory footprint in practice.

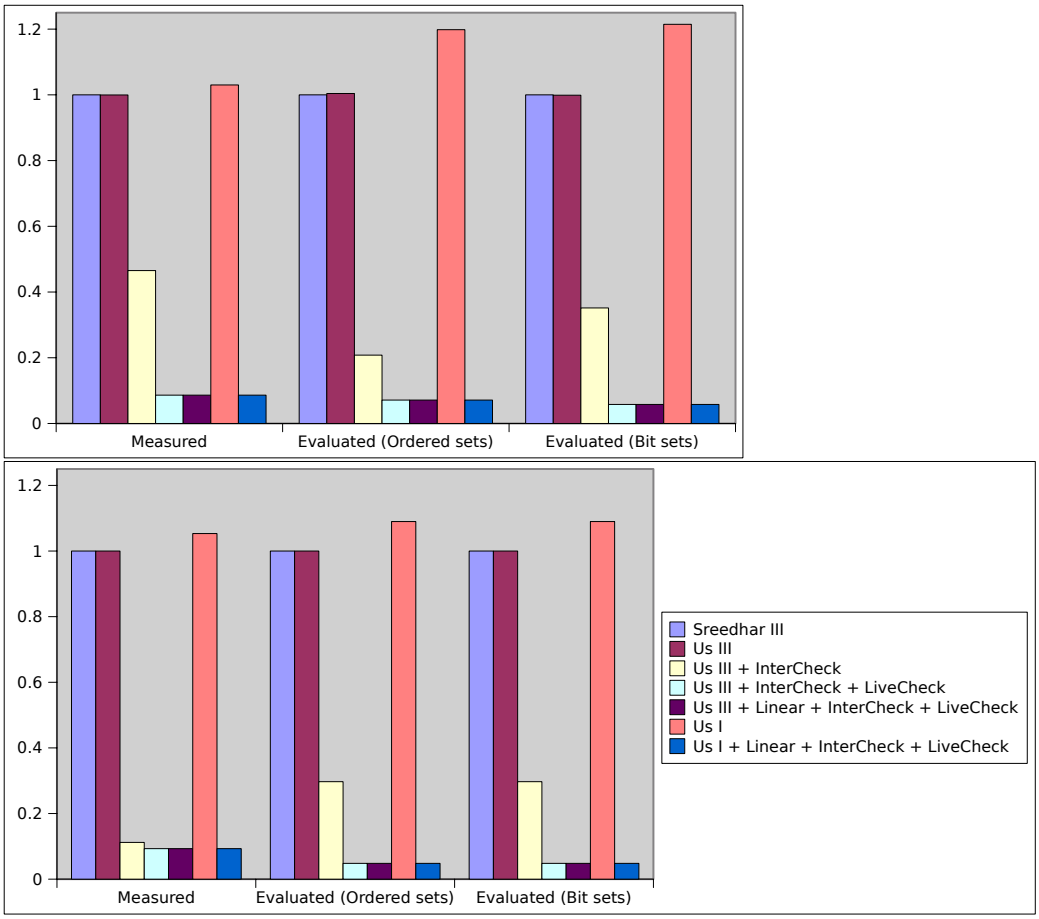


Figure 5.5: Performance results in terms of memory footprint (maximum and total).

5.4 Conclusion

In this chapter we have revisited the SSA destruction process in order to ensure its correctness, the quality of the generated code (i.e. minimize the number of new copies), and the speed of the algorithms. While previous work from Sreedhar et al. had already fixed the correctness issues, its most advanced method (Method III) is generally seen as hard to implement. The technique proposed by Budimlić et al. is geared towards speed, with the help of fast interference of SSA live-ranges, but it is difficult to implement correctly as well.

We reformulated the SSA destruction problem as an aggressive coalescing problem on CSSA, the transformed SSA resulting from Method 1 of Shreedar et al. Our experiments show that the refining a live-range intersection test with SSA value gives results as good as the more complex approach of Shreedar et al.

Then, we generalized the idea of dominance forests of Budimlić et al., first to enable interference checking between two congruence classes, then to integrate the check for equal values. In addition, we proposed a much simpler implementation which does not explicitly create the dominance forest. The reduced number of SSA variable intersection tests that results from this technique allows use to use a live-check approach, and avoid construction the live-sets.

Chapter 6

Conclusion

In this thesis we presented three main contributions around liveness, static single information, and SSA destruction.

6.1 Liveness

First, in chapter 3, we presented a novel way to build the live-sets associated with a basic block for variables under SSA form. Our technique based on the loop property bounds the number of time each basic block is visited, there is only two passes compared to the more classic data-flow algorithm which usually requires as many iteration as the depth of the loop forest of the control flow graph.

More interestingly we presented a different way to look at the liveness problem, instead of asking the question "what are the live variables as the start of the basic block?" (live-set) we instead ask "is the variable live at this program point?" (live-check). As we have shown, there exists an algorithm to answer this question which requires a much lighter pre-computed data structures than the live-set problem. But while the pre-computation is much faster and requires less memory, the drawback is that each query is more complex than a simple lookup in a set, the loop nesting forest needs to be inspected and reachability is tested for each use. This means that the runtime gain will depend on the number of queries. In practice, this threshold is usually favorable, for example the SSA destruction (see chapter 5), can reduce its runtime cost by using the liveness-query approach.

From an engineering point of view, the algorithm is interesting, as the

pre-computed data structures only depend on the shape of the control flow graph. This means that even if the code is modified, as long as it does not change the shape of the CFG, no pre-computation is needed. This is an advantage compared to the classical live-set approach, where a change of the code can trigger some tedious update of the live-sets.

Our final contribution for this chapter is a simple and precise way to look at interference. As shown by Chaitin et al., the problem of interference is not only a problem of intersection of live-ranges. The value of the variables themselves matters as well. Fortunately, the use of the SSA form, with its unique definition for every variable vastly simplifies the analysis of equalities of variables. This approach is also used in our SSA destruction algorithm from chapter 5.

6.2 Static Single Information form

Then in chapter 4, we explored an SSA variant, the static single information form. As the various definitions were inconsistent, we clarified them and explained their differences. We show that the original SSI form from Ananian (strong SSI) differ from the definition given by Singer (weak SSI form), as they do not split the live-range in the same way when the live-range of a variable at a loop exit point.

Even if the two form differ, we show that for both of them, the intersection graph of the live-ranges of the variables under SSI form is an interval graph. But our result is in fact even stronger, as our proof is constructive, we give an order of the control flow graph, depending only on the CFG, not on the variables themselves, such that every live-range form a consecutive interval.

Finally we explored the implication for liveness analysis of the result. For example, this means that the liveness can be stored in a very sparse way: only the start and end of the interval are required.

6.3 SSA destruction

Finally in chapter 5, we revisited the SSA destruction. Our approach simplifies the previous techniques, we first transform the code into CSSA form, adding copies as needed. In a second time, a classical aggressive coalescing is used to remove the useless copies. The simplicity of the approach allows us to easily

prove the correctness of the algorithm, which was not the case of the previous work in the area.

We then extend the work from chapter 3, using the notion of interference with values to reach remove as many useless copies as possible during the coalescing phase. This leads us to a generated code quality as good as previous more complex approaches.

Still using the work from chapter 3, in particular the liveness checking approach, we design fast algorithms for the whole approach. In particular, we present an fast algorithm to check interference between sets of variables, using the dominance property from SSA form to avoid useless tests, and using the single definition property to check for equality of values (refined interference test). This approach avoids building the liveness sets, which makes our algorithm faster.

6.4 Perspectives

We already have some work in progress, the first is related to SSI form. The motivation behind the SSI form was a simplification of some optimizations and analyses: the live-range splitting tries to ensure that some class of properties is valid all along every different live-range. This means that the information can be attached to the variable instead of using a more complex scheme, as would be the case if it differed in different parts of the live-range.

But the splitting done by SSI is only useful for a very small class of information, what is needed is a complete taxonomy of various analyses and optimizations. For every algorithm, we should try to determine if we can classify the information and then find out if a particular live-range splitting is suitable. We hope this work will be useful and will clarify and help people decide which variant of SSA they want to use.

Our other project is a fast register allocator based on SSA form. We can use the properties of SSA, processing the variables while following the dominance tree (in order to always encounter the definitions before every use), and coloring every variable as we encounter them. In order to make the algorithm fast and suitable for just-in-time compilers, we pick a global color for every variable, and re-color locally (inside a basic block) as needed.

Since we make a better use of the SSA properties (live-ranges are sub-tree of the dominance tree), we believe we can achieve better results than

a more classic linear scan approach, where interval have holes in them. Furthermore, our approach should prove very simple and lead to a straightforward implementation.

Bibliography

[1] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'88)*, pages 1–11. ACM, 1988.

[2] C. Scott Ananian. The Static Single Information Form. Technical Report MIT-LCS-TR-801, Laboratory for Computer Science, Massachusetts Institute of Technology, September 1999.

[3] Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, second edition, 2002.

[4] M. Bender and M. Farach-Colton. The LCA problem revisited. *LATIN 2000: Theoretical Informatics*, pages 88–94.

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

[6] Benoit Boissinot, Sebastian Hack, Daniel Grund, Benoît Dupont de Dinechin, and Fabrice Rastello. Fast liveness checking for SSA-form programs. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO'08)*, pages 35–44. ACM, 2008.

[7] Florent Bouchez, Alain Darte, Christophe Guillon, and Fabrice Rastello. Register allocation and spill complexity under SSA. Technical Report RR2005-33, LIP, ENS-Lyon, France, August 2005.

[8] Florent Bouchez, Alain Darte, and Fabrice Rastello. On the complexity of register coalescing. In *International Symposium on Code Generation*

and Optimization (CGO'07), pages 102–114. IEEE Computer Society Press, March 2007.

[9] P. Briggs, K. D. Cooper, T. J. Harvey, and L. T. Simpson. Practical improvements to the construction and destruction of static single assignment form. *Software – Practice and Experience*, 28(8):859–881, July 1998.

[10] Preston Briggs, Keith D. Cooper, and L. Taylor Simpson. Value numbering. *Software – Practice and Experience*, 27(6):701–724, 1997.

[11] Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, 16(3):428–455, 1994.

[12] B.Rosen, M.Wegman, and K.Zadeck. Global value numbers and redundant computations. In *POPL*, pages 12 – 27, 1988.

[13] Zoran Budimlić, Keith D. Cooper, Timothy J. Harvey, Ken Kennedy, Timothy S. Oberg, and Steven W. Reeves. Fast copy coalescing and live-range identification. In *ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI'02)*, pages 25–32. ACM Press, June 2002.

[14] G. J. Chaitin. Register allocation & spilling via graph coloring. In *Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pages 98–101, 1982.

[15] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6:47–57, January 1981.

[16] K. D. Cooper and L. Torczon. *Engineering a Compiler*. Morgan Kaufmann, 2004.

[17] Keith D. Cooper, Timothy J. Harvey, and Ken Kennedy. Iterative Data-Flow Analysis, Revisited. Technical Report TR04-100, Rice University, 2002.

[18] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form

and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451 – 490, 1991.

[19] Benoît Dupont de Dinechin, cois de Ferrière Fran Christophe Guillon, and Arthur Stoutchinin. Code generator optimizations for the ST120 DSP-MCU core. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES'00)*, pages 93 – 103, 2000.

[20] Lal George and Andrew W. Appel. Iterated register coalescing. *ACM Transactions on Programming Languages and Systems*, 18(3), May 1996.

[21] Paul Havlak. Nesting of Reducible and Irreducible Loops. *ACM Transactions on Programming Languages and Systems*, 19(4):557–567, 1997.

[22] Richard Johnson, David Pearson, and Keshav Pingali. The program structure tree: Computing control regions in linear time. *ACM SIGPLAN Notices*, 29(6):171–185, 1994.

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

[24] J.B. Kam and J.D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM (JACM)*, 23(1):158–171, 1976.

[25] Allen Leung and Lal George. Static single assignment form for machine code. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 204–214, 1999.

[26] Massimiliano Poletto and Vivek Sarkar. Linear scan register allocation. *ACM Transactions on Programming Languages and Systems*, 21(5):895–913, 1999.

[27] G. Ramalingam. On loops, dominators, and dominance frontiers. *ACM Transactions on Programming Languages and Systems*, 24(5):455–490, 2002.

[28] Fabrice Rastello, François de Ferrière, and Christophe Guillon. Optimizing translation out of SSA using renaming constraints. In *International Symposium on Code Generation and Optimization (CGO'04)*, pages 265–278. IEEE Computer Society Press, 2004.

[29] Jeremy Singer. Static Program Analysis Based on Virtual Register Renaming. Technical Report UCAM-CL-TR-660, University of Cambridge, Computer Laboratory, February 2006.

[30] Michael D. Smith, Norman Ramsey, and Glenn Holloway. A generalized algorithm for graph-coloring register allocation. In *ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI'04)*, pages 277–288. ACM, 2004.

[31] Vugranam C. Sreedhar, Roy Dz-Ching Ju, David M. Gillies, and Vatsa Santhanam. Translating out of static single assignment form. In *Static Analysis Symposium (SAS'99)*, pages 194 – 204, Italy, 1999.

[32] Michael Weiss. The transitive closure of control dependence: the iterated join. *ACM Lett. Program. Lang. Syst.*, 1(2):178–190, 1992.

[33] M. Wolfe. J+= J. *ACM Sigplan Notices*, 29(7):53, 1994.