Compiler Optimization Notes

November 23, 2022

Contents

1 Local Optimizations

Local Optimizations never goes away because this is always a piece of what happens even when we talk about even more sophiscated types of optimizations.

First we will talk about how to represent the code within a function or procedure, that's using something called a flow graph which is made of basic blocks. Next we will contrast two different abstractions for doing local optimizations.

1.1 Basic Blocks/Flow graphs

1.1.1 Basic Blocks

A basic block is a sequence of instructions(3-address statements). There are some requirements for basic block:

- **Only the first instruction can be reached from outside the blcok.** The reason why this property is useful is that within a basic block, we just march instruction by instruction through the block, this simplies things at least within a basic block.
- **All the statements are executed consecutively if the first one is.**
- **The basic block must be maximal.** i.e., they cannot be made larger without violating conditions.

1.1.2 Flow graphs

Flow graph is a graph representation of the procedure. In flow graph, basic blocks are the nodes, and the edge for $B_i \to B_j$ stands for a path from node B_i to node B_j. So how will $B_i \to B_j$ happen? There are two possibilities:

- Either first instruction of B_j is the target of a goto at end of B_i.
- B_j physically follows B_i which doesn't end in an unconditional goto.

1.1.3 Partitioning into Basic Blocks

- Identify the leader of each basic block
  - First instruction
  - Any target of a jump
  - Any instruction immediately following a jump
- Basic block starts at leader and ends at instruction immediately before a leader(or the last instruction).

An example of flow graph is shown below:

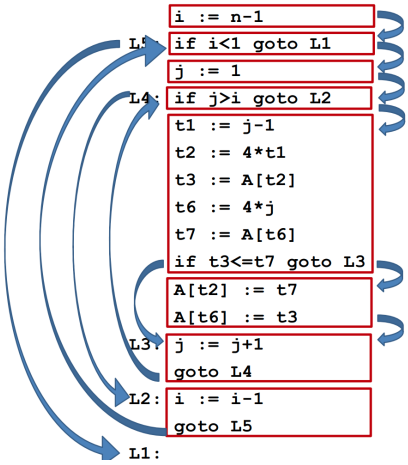


Figure 1: Example of a flow graph

1.1.4 Reachability of Basic Blocks

There is one thing interesting need to mention here. So the source code is below:

```
if x {
    ...
    return;
} else {
    ...
}
```

Listing 1: An example

The corresponding flow graph is shown in 2:

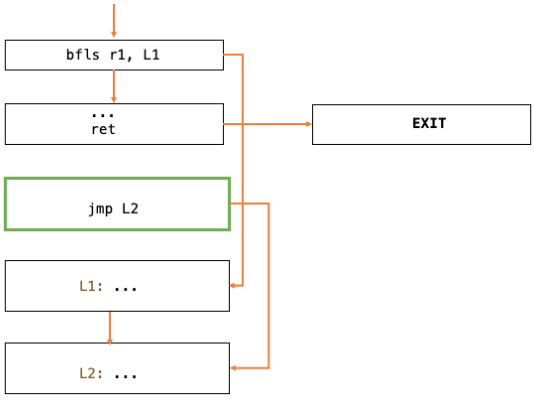


Figure 2: Example of a flow graph

We can see that the box in green is unreachable from the entry. So why is that interesting? Typically, after compiers construct the control flow graph, they will go through and remove any unreachable nodes. Just do depth first traversal of the graph from the entry node and mark all those visited nodes. So unmarked nodes will be deleted. This will help the compiler get a better optimization result.

So why do these unreachable nodes appear? The anwser is it is not the job of the front-end of the compiler to clean up the unreachable nodes.

1.2 Local optimizations

Local optimizations are those occur **within the basic blocks**.

1.2.1 common subexpression elimination

There're some types of local optimizations. One is called **common subexpression elimination**. Subexpressions are some arithmetic expressions that occur on the right hand of the instructions. The goal of this common subexpression elimination is to identify expressions that are guaranteed to produce identical values at runtime and arrange to only perform the associated computation once (when the first instance of the expression is encountered).

```
a = b + c;
d = b + c;
```

Listing 2: Subexpression example

In the example 2, `b + c` is so called coomon subexpression, we could replace the instruction containing common subexpression with an assign expression.

```
    a = b + c;
    d = a
```

Listing 3: code snippet applied common subexpression elimination to 2

You may wonder why this kind of redundancy can occure in code? Are we programmers stupid to do so? In fact, the redundancy most comes from the stage when compilers turn your source code. For example, **when you use arrays,** you need to do some arithmetic to generate the address of the array element you are accessing. So every time you referece the same array element, compiler will calculate the same address again. Similarly, if you **access offsets within fields**. Last example is **access to parameters** in the stack.

1.3 Abtraction 1:DAG

DAG is the acronym for Directed Acyclic Graph. The Directed Acyclic Graph (DAG) is used to represent the structure of basic blocks, to visualize the flow of values between basic blocks, and to provide optimization techniques in the basic block. DAG is an efficient method for identifying common sub-expressions.[1]

The parse tree and DAG of the expression $a + a * (b + c) + (b + c) * d$ is shown in 3.

In DAG, some of the computation are reused. So we can generate optimizaed code based on DAG.

[1]copied from https://wildpartyofficial.com/what-is-dag-in-compiler-construction

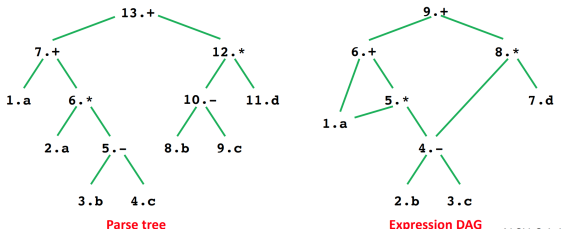


Figure 3: Example of a DAG

The optmized code for the DAG3 is:

```
t1 = b - c;
t2 = a * t1;
t3 = a + t2;
t4 = t1 * d;
t5 = t3 + t4;
```

Listing 4: code

1.3.1 How well do DAGs hold up across statements?

We have seen that DAGs can be useful in a long arithmetic expression. So how well do DAGs perform in sequence of instructions?

```
a = b + c;
b = a - d;
c = b + c;
d = a - d;
```

Listing 5: code

The corresponding DAG is shown in 4.
Based on the DAG4, one optimizaed code is 6

```
a = b+c;
d = a-d;
c = d+c;
```

Listing 6: code

6 is not correct. B need to be overwritten but not yet. So if using DAGs, you need to be very careful.

DAGs make sense if you just have one long expression, but once you have sequence of instructions overwriting variables , DAGs are less appealing because this abstraction doesn't really include the concept of time.

1.4 Abtraction 2:Value numbering

We have seen drawbacks of DAGs. One way to fix the problem is to attach variable name to latest value. Value numbering is such abstraction.

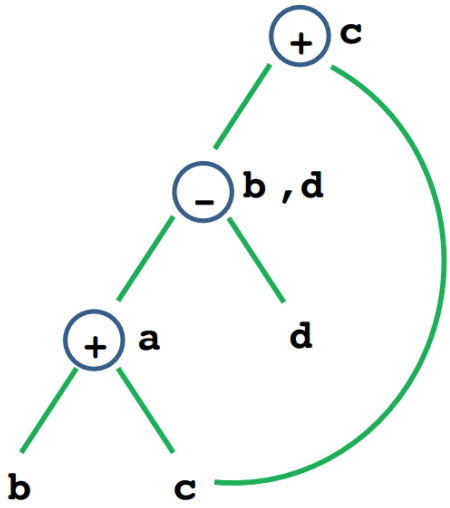


Figure 4: Example of a DAG

The idea behind value numbering is there is a mapping between variables(static) to values(dynamic). So common subexpression means same value number.

1.4.1 Algorithm

```
Data structure:
    VALUES = Table of
        expression /* [OP, valnum1, valnum2] */
        var /* name of variable currently holding expr */
For each instruction (dst = src1 OP src2) in execution order
    valnum1=var2value(src1); valnum2=var2value(src2)

    IF [OP, valnum1, valnum2] is in VALUES
        v = the index of expression
        Replace instruction with: dst = VALUES[v].var
    ELSE
        Add
            expression = [OP, valnum1, valnum2]
            var = tv
        to VALUES
        v = index of new entry; tv is new temporary for v
        Replace instruction with: tv = VALUES[valnum1].var OP VALUES[valnum2].var
                                  dst = tv
    set_var2value (dst, v)
```

Listing 7: code

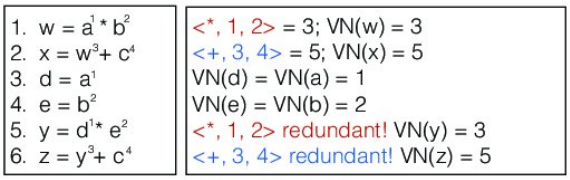


Figure 5: An example of value numbering.

1.4.2 Example

Figure 5 shows a concrete example of how VN identifies computation redundancies within a basic block. The VN processes each instruction statically. It obtains the previously computed symbolic value of each operand on the RHS, assigning a unique number on encountering a new operand. Then, it hashes the symbolic values assigned to operands together with the operator to obtain a symbolic value for the computation. If the computed symbolic value for a computation is already present in the table of previously computed values, then the current computation is redundant. In this basic block, computations on Line 5 and 6 are redundant since the computations are already computed by instruction on Line 1 and 2. [2]

[2] copied from https://www.researchgate.net/publication/283214075_Runtime_Value_Numbering_A_Profiling_Technique_to_Pinpoint_Redundant_Computations

2 Introduction to Data Flow Analysis

2.1 Motivation for Dataflow Analysis

Some optimizations[3] , however, require more "global" information. For example, consider the code 8

```
1   a = 1;
    b = 2;
3   c = 3;
    if (...) x = a + 5;
5   else x = b + 4;
    c = x + 1;
```

Listing 8: An

In this example, the initial assignment to c (at line 3) is useless, and the expression $x + 1$ can be simplified to 7, but it is less obvious how a compiler can discover these facts since they cannot be discovered by looking only at one or two consecutive statements. A more global analysis is needed so that the compiler knows at each point in the program:

- which variables are guaranteed to have constant values, and
- which variables will be used before being redefined.

To discover these kinds of properties, we use dataflow analysis.

2.1.1 What is Data Flow Analysis?

Local Optimizations only consider optimizations within a node in CFG. Data flow analysis will take edges into account, which means composing effects of basic blocks to derive information at basic block boundaries. Data-flow analysis is a technique for gathering information about the possible set of values calculated at various points in a computer program. A program's control-flow graph (CFG) is used to determine those parts of a program to which a particular value assigned to a variable might propagate. The information gathered is often used by compilers when optimizing a program.

Typically, we will do local optimization for the first step to know what happens in a basic block, step 2 is to do data flow analysis. In he third step, we will go back and revisit the individual instructions inside of the blocks.

Data flow analysis is **flow-sensitive**, which means we take into account the effect of control flow. It is also a **intraprocedural analysis** which means the analysis is within a procedure. Data-flow analysis computes its solutions over the paths in a control-flow graph. The well-known, meet-over-all-paths formulation produces safe, precise solutions for general dataflow problems. All paths-whether feasible or infeasible, heavily or rarely executed-contribute equally to a solution.

Here are some examples of intraprocedural optimizations:

- **constant propagation**. Constant propagation is a well-known global flow analysis problem. The goal of constant propagation is to discover values that are constant on all possible executions of a program and to propagate these constant values as far forward through the program

[3]based on https://pages.cs.wisc.edu/~horwitz/CS704-NOTES/2.DATAFLOW.html

as possible. Expressions whose operands are all constants can be evaluated at compile time and the results propagated further.

- **common subexpression elimination**
- **dead code elimination**. Actually, source code written by programmers doesn't contain a lot of dead code, dead code happens to occur partly because of how the front end translates code into the IR. Doing optimizations will also turn code into dead.

2.1.2 Static Program vs. Dynamic Execution

Program is statically finite, but there can be infinite many dynamic execution paths. On one hand, analysis need to be precise, so we will take into account as much dynamic execution as possible. On the other hand, analysis need to do the analysis quickly. For a compromise, the analysis result is **conservative** and what it does is for each point in the program, combines information of all the instances of the same program point.

2.1.3 Data Flow Analysis Schema

Before thinking about how to define a dataflow problem, note that there are two kinds of problems:

- Forward problems (like constant propagation) where the information at a node n summarizes what can happen on paths from "enter" to n. So if we care about what happened in the past, it's a forward problem.
- Backward problems (like live-variable analysis), where the information at a node n summarizes what can happen on paths from n to "exit". So if we care about what will happen in the future, it's a backward problem.

In what follows, we will assume that we're thinking about a forward problem unless otherwise specified.

Another way that many common dataflow problems can be categorized is as may problems or must problems. The solution to a "may" problem provides information about what may be true at each program point (e.g., for live-variables analysis, a variable is considered live after node n if its value may be used before being overwritten, while for constant propagation, the pair (x, v) holds before node n if x must have the value v at that point).

Now let's think about how to define a dataflow problem so that it's clear what the (best) solution should be. When we do dataflow analysis "by hand", we look at the CFG and think about:

- What information holds at the start of the program.
- When a node n has more than one incoming edge in the CFG, how to combine the incoming information (i.e., given the information that holds after each predecessor of n, how to combine that information to determine what holds before n).
- How the execution of each node changes the information.

This intuition leads to the following definition. An instance of a dataflow problem includes:

- a CFG,
- a domain D of "dataflow facts",
- a dataflow fact "init" (the information true at the start of the program for forward problems, or at the end of the program for backward problems),
- an operator $\wedge$ (used to combine incoming information from multiple predecessors),
- for each CFG node n, a dataflow function $f_n : D \rightarrow D$ (that defines the effect of executing n).

For constant propagation, an individual dataflow fact is a set of pairs of the form (var, val), so the domain of dataflow facts is the set of all such sets of pairs (the power set). For live-variable analysis, it is the power set of the set of variables in the program.

For both constant propagation and live-variable analysis, the "init" fact is the empty set (no variable starts with a constant value, and no variables are live at the end of the program).

For constant propagation, the combining operation $\wedge$ is set intersection. This is because if a node n has two predecessors, p1 and p2, then variable x has value v before node n iff it has value v after both p1 and p2. For live-variable analysis, $\wedge$ is set union: if a node n has two successors, s1 and s2, then the value of x after n may be used before being overwritten iff that holds either before s1 or before s2. In general, for "may" dataflow problems, $\wedge$ will be some union-like operator, while it will be an intersection-like operator for "must" problems.

For constant propagation, the dataflow function associated with a CFG node that does not assign to any variable (e.g., a predicate) is the identity function. For a node n that assigns to a variable x, there are two possibilities:

- 1. The right-hand side has a variable that is not constant. In this case, the function result is the same as its input except that if variable x was constant the before n, it is not constant after n.
- 2. All right-hand-side variables have constant values. In this case, the right-hand side of the assignment is evaluated producing consant-value c, and the dataflow-function result is the same as its input except that it includes the pair (x, c) for variable x (and excludes the pair for x, if any, that was in the input).

For live-variable analysis, the dataflow function for each node n has the form: $f_n(S) = Gen_n \cup (S - KILL_n)$, where $KILL_n$ is the set of variables defined at node n, and GEN_n is the set of variables used at node n. In other words, for a node that does not assign to any variable, the variables that are live before n are those that are live after n plus those that are used at n; for a node that assigns to variable x, the variables that are live before n are those that are live after n except x, plus those that are used at n (including x if it is used at n as well as being defined there).

An equivalent way of formulating the dataflow functions for live-variable analysis is: $f_n(S) = (S \cap NOT - KILL_n) \cup GEN_n$, where $NOT - KILL_n$ is the set of variables not defined at node n. The advantage of this formulation is that it permits the dataflow facts to be represented using bit vectors, and the dataflow functions to be implemented using simple bit-vector operations (and or).

It turns out that a number of interesting dataflow problems have dataflow functions of this same form, where GEN_n and $KILL_n$ are sets whose definition depends only on n, and the combining operator $\wedge$ is either union or intersection. These problems are called GEN/KILL problems, or bit-vector problems.

3 Live Variabl Analysis

In compilers, live variable analysis (or simply liveness analysis) is a classic data-flow analysis to calculate the variables that are live at each point in the program. A variable is live at some point if it holds a value that may be needed in the future, or equivalently if its value may be read before the next time the variable is written to. [4]

3.1 Motivation

Programs may contain

- code which gets executed but which has no useful effect on the program's overall result;
- occurrences of variables being used before they are defined;
- many variables which need to be allocated registers and/or memory locations for compilation.

The concept of variable liveness is useful in dealing with all three of these situations.

3.2 Problem formulation

Liveness is a data-flow property of variables: "Is the value of this variable needed?" We therefore usually consider liveness from an instruction's perspective: each instruction (or node of the flowgraph) has an associated set of live variables.

3.3 Semantic vs. syntactic

[5]

There are two kinds of variable liveness : Semantic liveness and Syntactic liveness.

A variable x is **semantically** live at a node n if there is some execution sequence starting at n whose (externally observable) behaviour can be affected by changing the value of x. Semantic liveness is concerned with the execution behaviour of the program.

A variable is **syntactically** live at a node if there is a path to the exit of the flow graph along which its value may be used before it is redefined. Syntactic liveness is concerned with properties of the syntactic structure of the program.

So what is the difference between Semantic liveness and Syntactic liveness? syntactic liveness is a computable approximation of semantic liveness.

Consider the example

```
    int t = x * y;
2   if ((x+1)*(x+1) == y) {
     t = 1;
4   }
    if (x*x + 2*x + 1 != y) {
6    t = 2;
    }
8   return t;
```

Listing 9: An

[4] based on Wikipedia

[5] based on slides from Cambridge University

In fact, t is dead in node `int t = x;` because one of the conditions will be true, so on every execution path t is redefined before it is returned. The value assigned by the first instruction is never used.

But on read path from 6 through the flowgraph, t is not redefined before it's used, so t is syntactically live at the first instruction.Note that this path never actually occurs during execution.

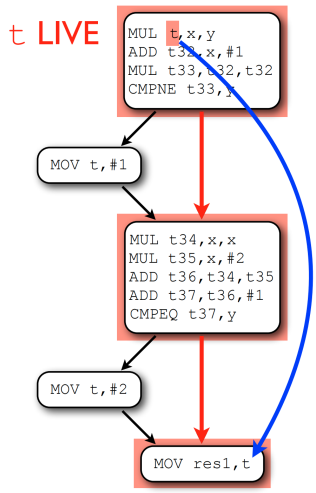


Figure 6: CFG

4 Reaching Definitions

The Reaching Definitions Problem is a data-flow problem used to answer the following questions: Which definitions of a variable X reach a given use of X in an expression? Is X used anywhere before it is defined? A definition$d$ reaches a point p if there exists path from the point immediately following d to p such that d is not killed(overwritten) along that path.

4.1 Iterative Algorithm

Here is the iterative algorithm.

Algorithm 1 Reaching Defintions:Iterative Algorithm

Input: control flow graph CFG = (N, E, Entry, Exit)

out[Entry] = $\emptyset$ ▷ Boundary condition
for `each basic block B other than Entry` **do**
  `out[B] =` $\emptyset$ ▷ Initialization for iterative algorithm
end for
while Changes to any out[] occur **do**
  for `each basic block B other than Entry` **do**
    $in[B] = \cup(out[p])$`, for all predecessors p of B`
    $out[B] = f_B(in[B])$ ▷ $out[B] = gen[B] \cup (in[B] - kill[B])$
  end for
end while

4.2 Worklist Algorithm

4.3 Example

Here comes an example of reaching definition.

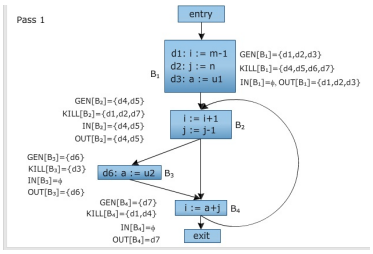


Figure 7: Pass 1

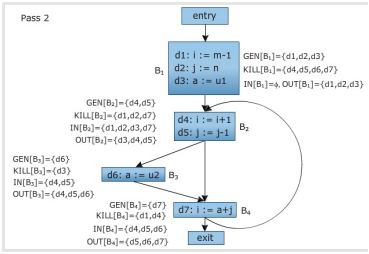


Figure 8: Pass 2

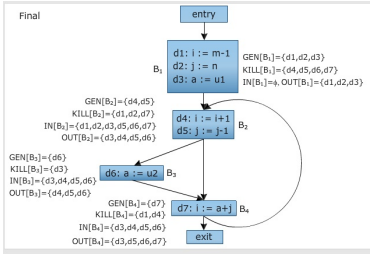


Figure 9: Pass 3

Algorithm 2 Reaching Defintions:Worklist Algorithm

Input: control flow graph CFG = (N, E, Entry, Exit)

out[Entry] = ∅ ▷ Boundary condition
ChangedNodes = N
for `each basic block B other than Entry` **do**
  `out[B] = ∅` ▷ Initialization for iterative algorithm
end for
while ChangedNodes ≠ ∅ **do**
  Remove i from ChangedNodes
  $in[B] = \cup(out[p])$, for all predecessors p of B
  $oldout = out[i]$
  $out[i] = f_i(in[i])$ ▷ $out[i] = gen[i] \cup (in[i] - kill[i])$
  if oldout ≠ $out[i]$ **then**
    for `all successors s of i` **do**
      add s to ChangedNodes
    end for
  end if
end while

5 Available Expressions Analysis

5.1 Motivation

Programs may contain code whose result is needed, but in which some computation is simply a redundant repetition of earlier computation within the same program. The concept of expression availability is useful in dealing with this situation.

5.2 Backgroud Knowledge

Any given program contains a finite number of expressions (i.e. computations which potentially produce values),so we may talk about the set of all expressions of a program. Consider the program in

```
int z = x * y;
print s + t;
int w = u / v;
```

Listing 10: An

This program contian expression `x*y,s+t,u/v`.

5.3 Problem Formulation

Availability is a data-flow property of expressions: “Has the value of this expression already been computed?” At each instruction, each expression in the programis either available or unavailable. So each instruction(or node of the flowgraph) has an associated set of available expression.

5.4 Semantic vs. Syntactic

An expression is *semantically* available at a node n if its value gets computed (and not subsequently invalidated) along every execution sequence ending at n.

```
int x = y * z;
⋮
return y * z;   y*z AVAILABLE
```

Figure 10: Available expression example

```
int x = y * z;
⋮
y = a + b;
⋮
return y * z;   y*z UNAVAILABLE
```

Figure 11: unavailable expression example

An expression is *syntactically* available at a node n if its value gets computed (and not subsequently invalidated) along every path from the entry of the flowgraph to n.

```
if ((x+1)*(x+1) == y) {
  s = x + y;
}
if (x*x + 2*x + 1 != y) {
  t = x + y;
}
return x + y;   x+y AVAILABLE
```

Figure 12: x+y is semantically available

Figure 13: x+y is syntactically unavailable

On the path in red from Figure13 through the flowgraph, $x + y$ is only computed once, so $x + y$ is syntactically unavailable at the last instruction.

Whereas with live variable analysis we found safety in assuming that more variables were live, here we find safety in assuming that fewer expressions are available. Because if an expression is deemed to be available, we may do something dangerous (e.g. remove an instruction which recomputes its value). So sometimes safe means more, but sometimes means less.

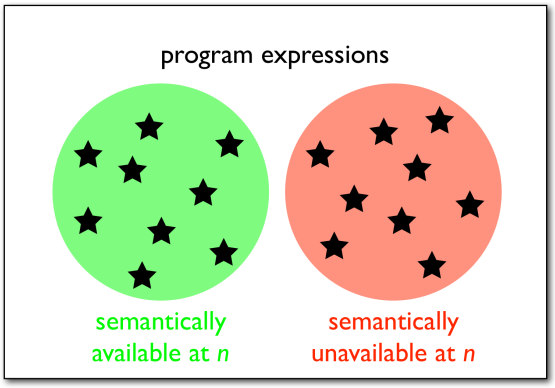


Figure 14: Semantic vs. syntactic

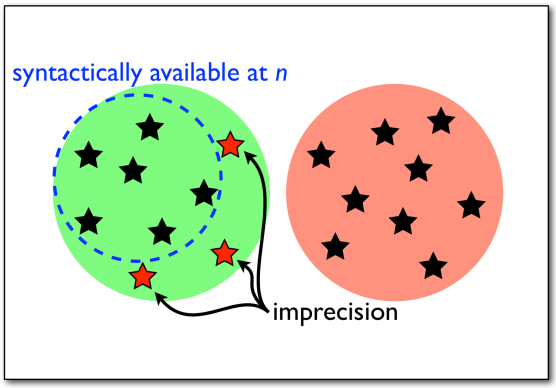


Figure 15: Semantic vs. syntactic

6 Foundations of Data Flow Analysis

We saw a lot of examples of data flow analysis, eg. reaching definitions etc. Although there were differences between differeent types of data flow analysis, they did share number of things in common. Our goal is to develop a general purpose data flow analysis framework.

There are some questions that we want to answer about a framework that performs data flow analysis.

- Correctness: Do we get a correct answer?
- Precison: How good is the answer? [6]
- Covergence: Will the analysis terminate?
- Speed: How fast is the convergence?

6.1 A Unified Framework

Data flow problems are defined by

- Domain of values V(eg, variable names for liverness, the instruction numbers for reaching definitions)
- Meet operator $V \wedge V \to V$ to deal with the join nodes.
- Initial value. Once we have defined the meet operator, it will tell us how to initialize all of the non-entry or exits nodes and the boundary conditions for entry and exit nodes.
- A set of transfer functions $V \to V$ to define how information flows across basic blocks.

Why we bother to define such a framework?

- First, if meet operator, transfer function and the domains of values are specified in proper way, we will know about correctness, precision and so on.
- From practical engineering perspective, it allows us to reuse code.

6.2 Partial Order

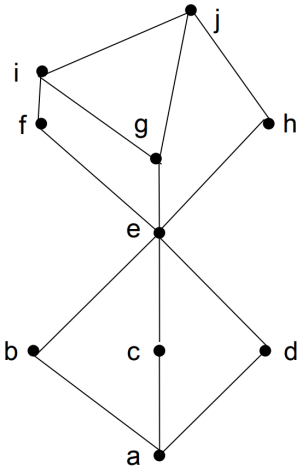
A relation R on a set S is called a **partial order** if it is

- **Transitivity** if $x \preceq y$ and $y \preceq z$ then $x \preceq z$
- **Antisymmetriy** if $x \preceq y$ and $y \preceq x$ then $x = x$
- **Reflexivity** $x = x$

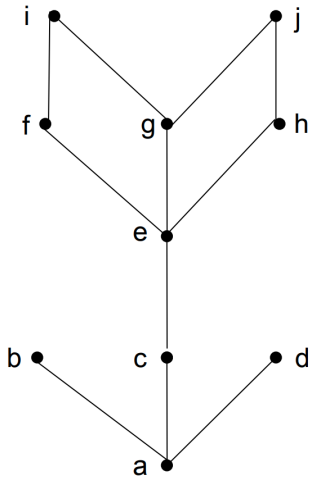
6.3 Lattice

A lattice is a partially ordered set in which every pair of elements has both a least upper bound (lub) and a greatest lower bound(glb).

[6] We want a safe solution but as precise as possible.



(a) This is a lattice example.



(b) This is not a lattice example becuasethe pair b,c deos not have a lub.

Figure 16: Two examples

6.4 Complete Lattice

A lattice A is called a complete lattice if every subset S of A admits a glb and a lub in A.

6.5 Semi-Lattice

A semilattice (or upper semilattice) is a partially ordered set that has a least upper bound for any nonempty finite subset.

6.6 Meet Operator

Meet operator must hold the following properties:

- **commutative**: $x \wedge y = y = \wedge x$. No ordering in the incoming edges.
- **idempotent**: $x \wedge x = x$
- **associative** : $x \wedge (y \wedge z) = (x \wedge y) \wedge z$
- there is a Top element T such that $x \wedge T = x$. Partly due to the way we initialize everything we need.

Meet Operator defines a partial ordering on values. This is important in ensuring the analysis converges. So what does it mean ? $x \preceq y$ if and only if $x \wedge y = x$. The $\preceq$ not means less or equal to or subset, but it really means lattice inclusion. So if $x \preceq y$, this means x is more conservative

Figure 17: Meet Operator

or constrained. In another word, x is lattice included in y. Partial ordering will also lead to some other properties

- **Transitivity** if $x \preceq y$ and $y \preceq z$ then $x \preceq z$
- **Antisymmetriy** if $x \preceq y$ and $y \preceq x$ then $x = z$
- **Reflexivity** $x = x$

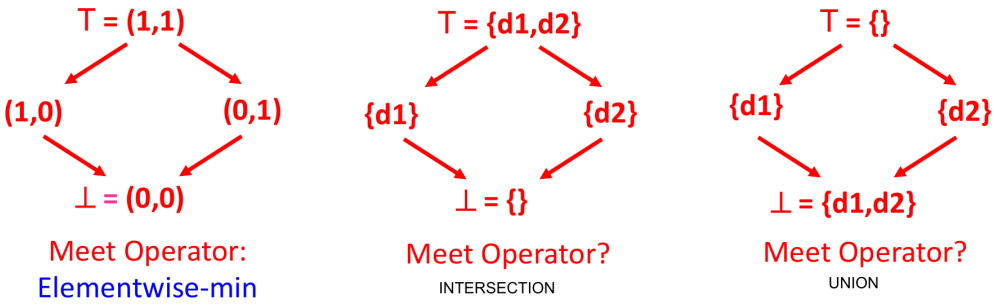


Figure 18: Different meet operator defines different lattice

For our data flow analysis, values and meet operator define a semi-lattice, which means $\top$ exits, but not necessarily $\bot$.

6.7 Descending Chain

The height of a lattice is the largest number of $\succ$ relations that will fit in a descending chain. eg. $x_0 \succ x_1 \succ x_2 \succ \ldots$

So, for reaching definitions, the height is the number of definitions.

Finite descending chain will ensure the convergence. If we don't finite descending chain, there is a possibility that the analysis will never terminate. But a infinite lattice still can have a finite descending chain. I want to note that infinite lattice doesn't always mean a non-convergence.

So consider the constant propogation, the infinite lattice has finite descending chain, so this can converge.

6.8 Transfer Functions

Tranfer function dicates how information propagates across a basic block. So what we need for our transfer function? **First**, it must have an identity function which means there exists an f such

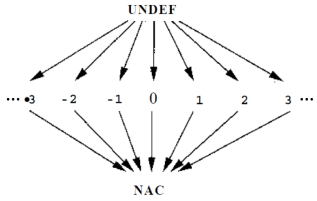


Figure 19: The lattice of constant propogation

that $f(x) = x$ for all x. For example, in Reaching Definitions and Liveness, when $Gen, KILL = \Phi$, this transfer function satisfies $f(x) = x$. **Second**, when we compose transfer functions, it must be consitent with the transfer function. So if $f_1, f_2 \in F$, the $f_1 \cdot f_2 \in F$.

For example,

$$\begin{aligned} f_1(x) &= G_1 \cup (x - K_1) \\ f_2(x) &= G_2 \cup (x - K_2) \\ f_2(f_1(x)) &= G_2 \cup [(G_1 \cup (x - K_1)) - K_2] \\ &= [G_2 \cup (G_1 - K_2)] \cup [x - (K_1 \cup K_2)] \\ G &= G_2 \cup (G_1 - K_2) \\ K &= K_1 \cup K_2 \end{aligned}$$

6.9 Monotonicity

A framework $(F, V, \wedge)$ is monotone if and only if $x \preceq y$ implies $f(x) \preceq f(y)$. This means that a "smaller(more conservative) or equal" input to the same function will always give a "smaller(more conservative) or equal" output.

Alternatively, $(F, V, \wedge)$ is monotone if and only if $f(x \wedge y) \preceq f(x) \wedge f(y)$. So merge input, then apply f is small(more conservative) or equal to apply the transfer function individually and then merge the result. Values are defined by semi-lattice, the meet operator only ever moves down the lattice from top towards the bottom. So we need to constrain the transfer function.

I will show you a unmonotone example.

Let top be 1 and bottom be 0 and the meet operator is $\cap$. $f(0) = 1, f(1) = 0$

Let's check whether reaching definitions is monotone.

Note that monotone framework does not mean $f(x) \preceq x$.

6.10 Distributivity

Reaching definitions is distributive but constant propogation is not.

7 Introduction to Static Single Assignment

Many dataflow analyses need to find the use-sites of each defined variable or the definition-sites of each variable used in an expression. The *def-use chain* is a data structure that makes this efficient: for each statement in the flow graph, the compiler can keep a list of pointers to all the use sites of variables defined there, and a list of pointers to all definition sites of the variables used there. An improvement on the idea of def-use chains is static single-assignment form, or SSA form, an intermediate representation in which each variable has only one definition in the program text. SSA is very useful for many optimizations such as Loop-Invariant Code Motion and Copy Propagation.

7.1 Definition-Use and Use-Definition Chains

Use-Definition (UD) Chains

For a given definition of a variable X, what are all of its uses?

Definition-Use (DU) Chains

For a given use of a variable X, what are all of the reaching definitions of X?

Unfortunately, it is expensive to use UD and DU chains, because if we have N defs, and M uses, the space complexity is O(NM). An example is in 20

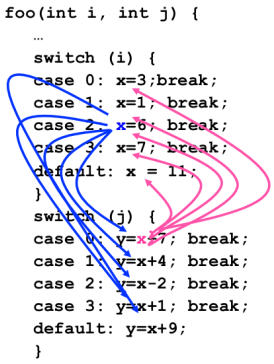


Figure 20: If a variable has N uses and M definitions (which occupy about N + M instructions in a program), it takes space (and time) proportional to N · M to represent def-use chains – a quadratic blowup.

7.2 Static Single Assignment(SSA)

Static Single Assignment

Static Single Assignment is an IR where every variable is assigned a value at most once in the program text.

the Φ function

Φ merges multiple definitions along multiple control paths into a single definition.
At a basic block with p predecessors, there are p arguments to the Φ functions.

$$x_{\text{new}} \leftarrow \Phi\left(\mathbf{x}_1, \mathbf{x}_1, \mathbf{x}_1, \ldots, \mathbf{x}_{\text{p}}\right)$$

7.2.1 Why SSA is useful?

Useful for Dataflow Analysis Dataflow analysis and optimization algorithms can be made simpler when each variable has only one definition.

Less space and time complexity If a variable has N uses and M definitions (which occupy about N + M instructions in a program), it takes space (and time) proportional to N · M to represent def-use chains – a quadratic blowup. For almost all realistic programs, the size of the SSA form is linear in the size of the original program.

Simplify some algorithms Uses and defs of variables in SSA form relate in a useful way to the dominator structure of the control-flow graph, which simplifies algorithms such as interference-graph construction.

Eliminate needless relationship Unrelated uses of the same variable in the source program become different variables in SSA form, eliminating needless relationships shown in 11.

```
1
for i <- 1 to N do A[i] <- 0
3 for i <- 1 to M do s <- s + B[i]
```

Listing 11: An example

7.3 How to represent SSA?

In straight-line code, such as within a basic block, it is easy to see that each instruction can define a fresh new variable instead of redefining an old one shown in 21

But when two control-flow paths merge together, it is not obvious how to have only one assignment for each variable. To solve this problem we introduce a notational fiction, called a Φ function. Figure 22 shows that we can combine a1 (defined in block 1) and a2 (defined in block 3) using the function $a3 \leftarrow \Phi(a1, a2)$.

unlike ordinary mathematical functions, Φ(a1, a2) yields a1 if control reaches block 4 along the edge $2 \rightarrow 4$, and yields a2 if control comes in on edge $3 \rightarrow 4$.

$$
\begin{array}{lcl}
a & \leftarrow & x+y \\
b & \leftarrow & a-1 \\
a & \leftarrow & y+b \\
b & \leftarrow & x \cdot 4 \\
a & \leftarrow & a+b
\end{array}
\qquad
\begin{array}{lcl}
a_1 & \leftarrow & x+y \\
b_1 & \leftarrow & a_1-1 \\
a_2 & \leftarrow & y+b_1 \\
b_2 & \leftarrow & x \cdot 4 \\
a_3 & \leftarrow & a_2+b_2
\end{array}
$$

(a) A straight-line program. (b) The program in single-assignment form.

Figure 21: SSA for straight-line code

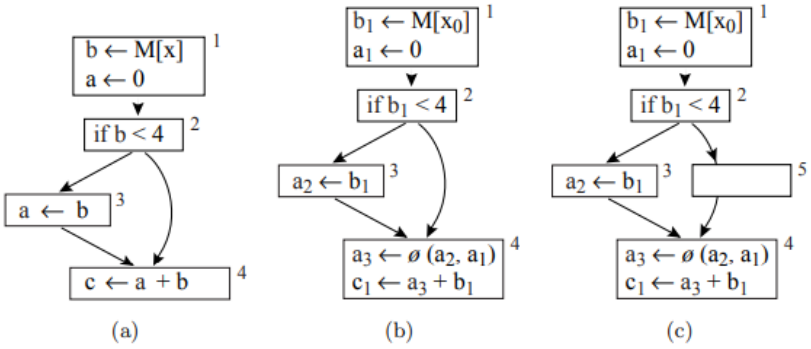


Figure 22: (a) A program with a control-flow join; (b) the program transformed to single-assignment form; (c) edge-split SSA form.

7.3.1 How does the ϕ-function know which edge was taken?

If we must execute the program, or translate it to executable form, we can "implement" the Φ-function using a move instruction on each incoming edge as shown in Figure 23. However, in many cases, we simply need the connection of uses to definitions and don't need to "execute" the Φ-functions during optimization. In these cases, we can ignore the question of which value to produce.

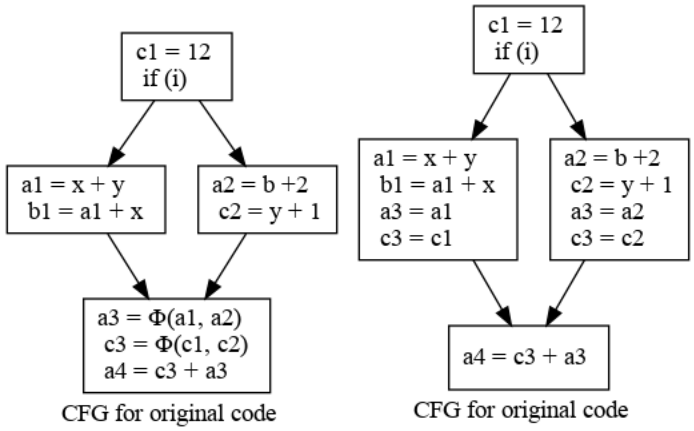
7.4 Converting to SSA form

The algorithm for converting a program to SSA form is roughly as follows:

- 1. adds Φ functions for the variables, and then
- 2. renames all the definitions and uses of variables using subscripts.

7.4.1 Trivial SSA

Trivial SSA form is based on a simple observation: Φ functions are only needed for variables that are "live" after the Φ function.



(a) Original code

(b) Code after moving instruction.

Figure 23: Implementing Φ-function

- Each assignment generates a fresh variable.
- At each join point insert Φ for all live variables.

Trivial SSA will generate some useless Φ functions. An example is shown in Figure 24 So a Φ-function is not needed for every variable at each point.

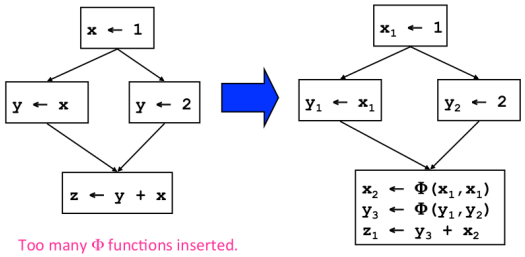


Figure 24: $x2 \leftarrow \Phi(x1, x1)$ is useless because x2 is equal to x1.

7.4.2 Minimal SSA

Minimal SSA is an updated version compared to trivial SSA.

- Each assignment generates a fresh variable.
- At each join point insert Φ for all live variables with multiple outstanding defs.

7.4.3 Path-convergence criterion

There should be a Φ-function for variable a at node z of the flow graph exactly when all of the following are true:

- 1. There is a block x containing a definition of a,
- 2. There is a block y (with $y \neq x$) containing a definition of a,
- 3. There is a nonempty path P_{xz} of edges from x to z,
- 4. There is a nonempty path P_{yz} of edges from y to z,
- 5. Paths P_{xz} and P_{yz} do not have any node in common other than z, and
- 6. The node z does not appear within both P_{xz} and P_{yz} prior to the end, though it may appear in one or the other.

We consider the start node to contain an implicit definition of every variable, either because the variable may be a formal parameter or to represent the notion of a ← uninitialized without special cases. A Φ-function itself counts as a definition of a, so the path-convergence criterion must be considered as a set of equations to be satisfied. As usual, we can solve them by iteration as shown in 3.

Algorithm 3 Iterated path-convergence criterion

while there are nodes x, y, z satisfying conditions 1–5 and
z does not contain a Φ-function for a **do**
    insert a ← Φ(a, a, . . . , a) at node Z
end while

7.4.4 Dominance property of SSA form

The iterated path-convergence algorithm for placing Φ-functions is not practical, since it would be very costly to examine every triple of nodes x, y, z, and every path leading from x and y. A much more efficient algorithm using the dominator tree of the flow graph as shown in Figure 25.

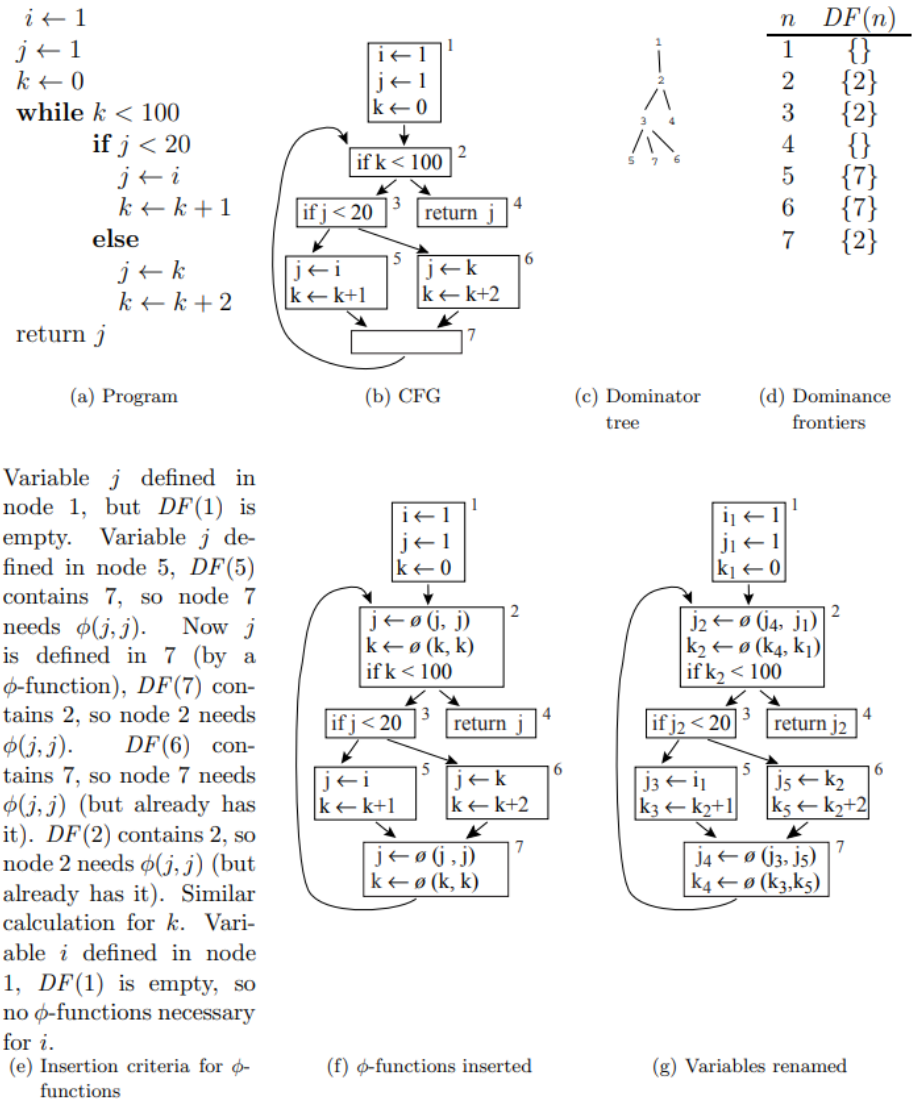


Figure 25: Conversion of a program to static single-assignment form. Node 7 is a postbody node, inserted to make sure there is only one loop edge; such nodes are not strictly necessary but are sometimes helpful.

Strictly dominance

x strictly dominates w (x sdom w) iff impossible to reach w without passing through x first.

Dominance

x dominates w (x dom w) iff x sdom w or x = w.

$$\mathrm{Dom}(n) = \begin{cases} \{n\} & \text{if } n = n_0 \\ \{n\} \cup \left(\bigcap_{p \in \mathrm{preds}(n)} \mathrm{Dom}(p) \right) & \text{if } n \neq n_0 \end{cases}$$

Dominance tree

x sdom w iff x is a proper ancestor of w.

Dominance Frontier

The dominance frontier of a node x is the set of all nodes w such that x dominates a predecessor of w, but does not strictly dominate w.

$$F(x) = \{w | \texttt{x dom pred(w) AND !(x sdom w) } \}$$

An essential property of static single assignment form is that definitions dominate uses; more specifically,

- If x is the ith argument of a Φ-function in block n, then the definition of x dominates the ith predecessor of n.
- If x is used in a non-Φ statement in block n, then the definition of x dominates n

Dominance Property of SSA

In SSA,

- If x i is used in $x \leftarrow \Phi(..., x_i, ...)$, then $BB(x_i)$ dominates ith predecessor of $BB(\Phi)$
- If x is used in $y \leftarrow ...x...$,then BB(x) dominates BB(y)

Dominance frontier criterion. Whenever node x contains a definition of some variable a, then any node z in the dominance frontier of x needs a Φ-function for a.

Iterated dominance frontier. Since a Φ-function itself is a kind of definition, we must iterate the dominance-frontier criterion until there are no nodes that need Φ-functions.

Theorem. The iterated dominance frontier criterion and the iterated path convergence criterion specify exactly the same set of nodes at which to put Φ-functions

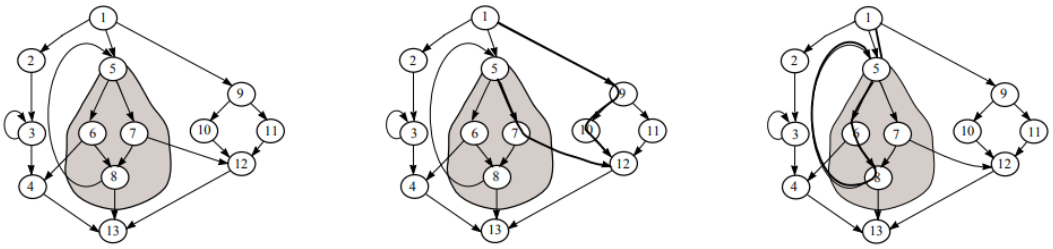


Figure 26: Node 5 dominates all the nodes in the grey area. (a) Dominance frontier of node 5 includes the nodes (4, 5, 12, 13) that are targets of edges crossing from the region dominated by 5 (grey area including node 5) to the region not strictly dominated by 5 (white area including node 5). (b) Any node in the dominance frontier of n is also a point of convergence of nonintersecting paths, one from n and one from the root node. (c) Another example of converging paths $P_{1,5}$ and $P_{5,5}$.

Proof

The sketch of a proof that shows if w is in the dominance frontier of a definition, then it must be a point of convergence.
Suppose there is a definition of variable a at some node n (such as node 5 in Figure 26b), and node w (such as node 12 in Figure 26b) is in the dominance frontier of n. The root node implicitly contains a definition of every variable, including a. There is a path P_{rw} from the root node (node 1 in Figure 26) to w that does not go through n or through any node that n dominates; and there is a path P_{nw} from n to w that goes only through dominated nodes. These paths have w as their first point of convergence.

7.5 Computing the dominance frontier

To insert all the necessary Φ-functions, for every node n in the flow graph we need DF[n], its dominance frontier. Given the dominator tree, we can efficiently compute the dominance frontiers of all the nodes of the flow graph in one pass. We define two auxiliary sets

- $DF_{local}[n]$ The successors of n that are not strictly dominated by n;
- $DF_{up}[n]$ Nodes in the dominance frontier of n that are not dominated by n's immediate dominator.

The dominance frontier of n can be computed from $DF_{local}[n]$ and $DF_{up}[n]$

$$DF[n] = DF_{\text{local}}\,[n] \quad \cup \bigcup_{c \in \text{ children } [n]} DF_{\text{up}}\,[c]$$

where children[n] are the nodes whose immediate dominator (idom) is n.

To compute $DF_{local}[n]$ 4 more easily (using immediate dominators instead of dominators), we use the following theorem: $DF_{local}[n]$ = the set of those successors of n whose immediate dominator is not n. The following computeDF function should be called on the root of the dominator tree (the start node of the flow graph). It walks the tree computing DF[n] for every node n: it computes $DF_{local}[n]$ by examining the successors of n, then combines $DF_{local}[n]$ and (for each child c) $DF_{up}[n]$.a

Algorithm 4 computeDF

```
S ← {}
for each node y in succ[n] do                    ▷ This loop computes DF_local[n]
    if idom(y) ≠ n then
        S ← S ∪ {y}
    end if
end for
for each child c of n in the dominator tree do
    computeDF[c]
    for each element w of DF[c] do               ▷ This loop computes DF_up[n]
        if n does not dominate w then
            S ← S ∪ {w}
        end if
    end for
end for
```

This algorithm is quite efficient. It does work proportional to the size (number of edges) of the original graph, plus the size of the dominance frontiers it computes. Although there are pathological graphs in which most of the nodes have very large dominance frontiers, in most cases the total size of all the DFs is approximately linear in the size of the graph, so this algorithm runs in "practically" linear time.

7.6 Inserting Φ-functions

Starting with a program not in SSA form, we need to insert just enough Φ-functions to satisfy the iterated dominance frontier criterion. To avoid re-examining nodes where no Φ-function has been inserted, we use a work-list algorithm.

Algorithm5 starts with a set V of variables, a graph G of controlflow nodes – each node is a basic block of statements – and for each node n a set A_{orig}[n] of variables defined in node n. The algorithm computes $A_\Phi[a]$, the set of nodes that must have Φ-functions for variable a. Sometimes a node may contain both an ordinary definition and a Φ-function for the same variable; for example, in Figure 26b, a $\in A_{orig}[2]$ and $2 \in A_\Phi[a]$.

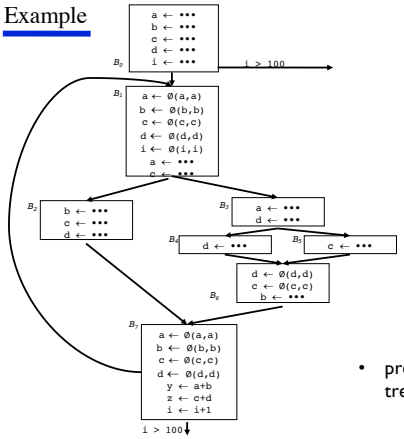
This algorithm does a constant amount of work (a) for each node and edge in the control-flow graph, (b) for each statement in the program, (c) for each element of every dominance frontier, and (d) for each inserted Φ-function. For a program of size N, the amounts a and b are proportional to N, c is usually approximately linear in N. The number of inserted Φ-functions (d) could be N^2 in the worst case, but empirical measurement has shown that it is usually proportional to N. So in practice, Algorithm 5 runs in approximately linear time.

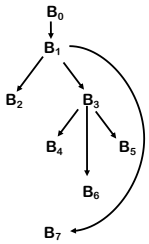
7.7 Renaming the variables

After the Φ-functions are placed, we can walk the dominator tree, renaming the different definitions (including Φ-functions) of variable a to a1, a2, a3 and so on. Rename each use of a to use the closest definition d of a that is above a in the dominator tree. Algorithm renames all uses and definitions of variables, after the Φ-functions have been inserted by Algorithm 6. In traversing the dominator tree, the algorithm "remembers" for each variable the most recently defined version of each variable, on a separate stack for each variable. Although the algorithm follows the structure of the dominator tree – not the flow graph – at each node in the tree it examines all outgoing flow edges, to see if there are any Φ-functions whose operands need to be properly numbered.

7.7.1 Example

Example

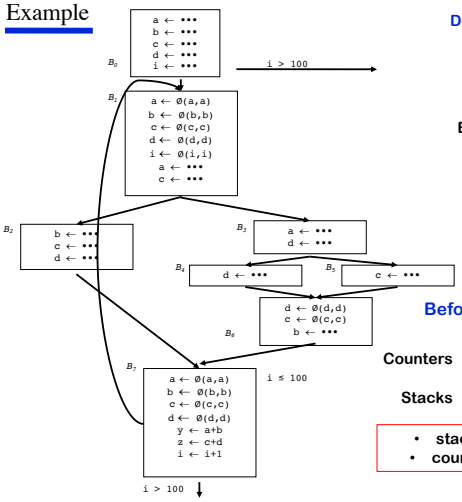




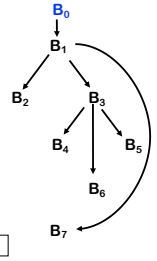
Dominance Tree

- pre-order traversal of the dominator tree: {0,1,2,3,4,5,6,7}

Example



Dominance Tree

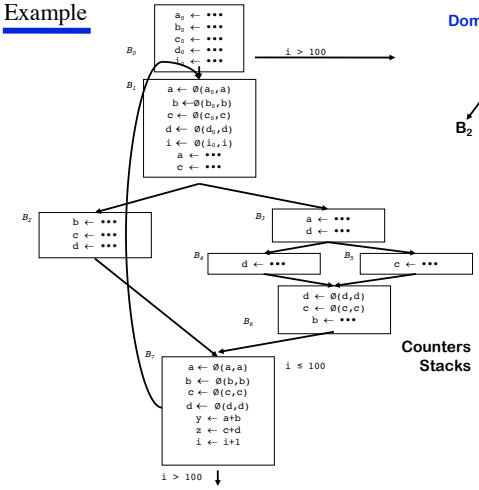


Before processing B_0

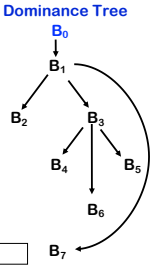
	a	b	c	d	i
Counters	0	0	0	0	0
Stacks					

- stacks for uses of variables
- counters for new definitions

Example



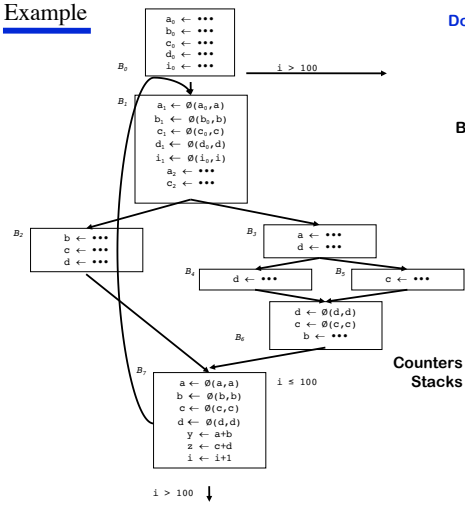
Dominance Tree



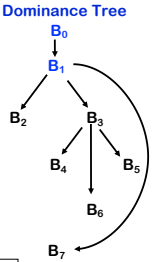
End of B_0

	a	b	c	d	i
Counters	1	1	1	1	1
Stacks	a_0	b_0	c_0	d_0	i_0

Example



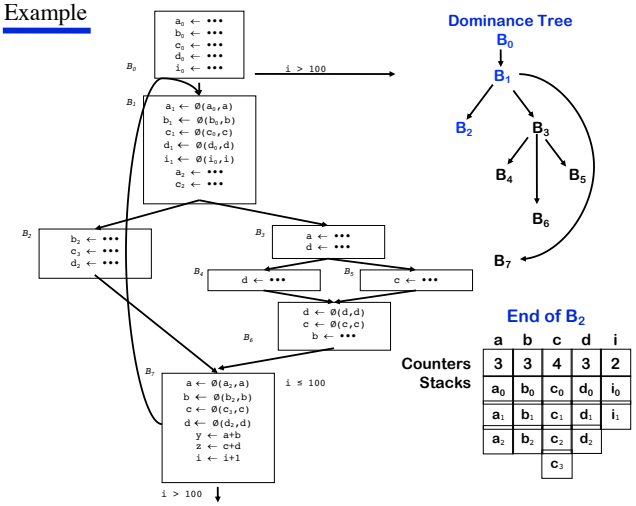
Dominance Tree



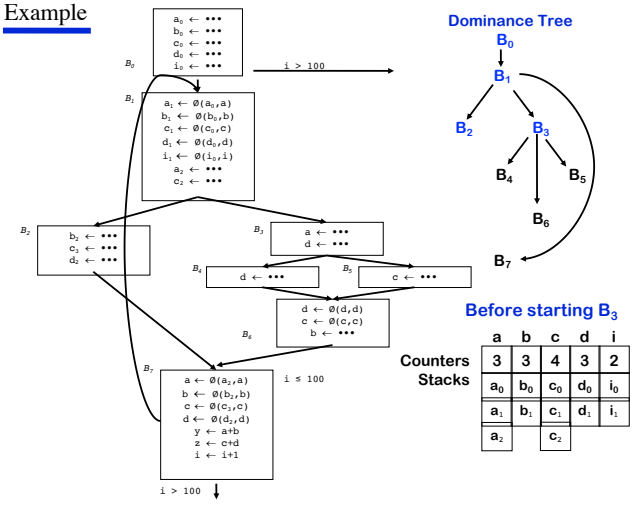
End of B_1

	a	b	c	d	i
Counters	3	2	3	2	2
Stacks	a_0	b_0	c_0	d_0	i_0
	a_1	b_1	c_1	d_1	i_1
	a_2		c_2		

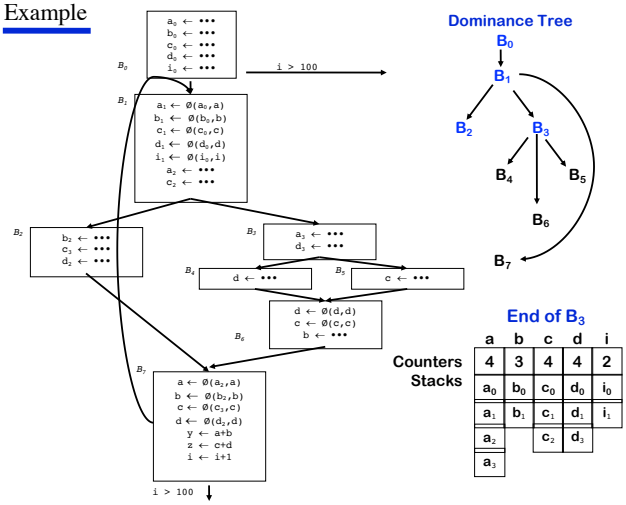
Example



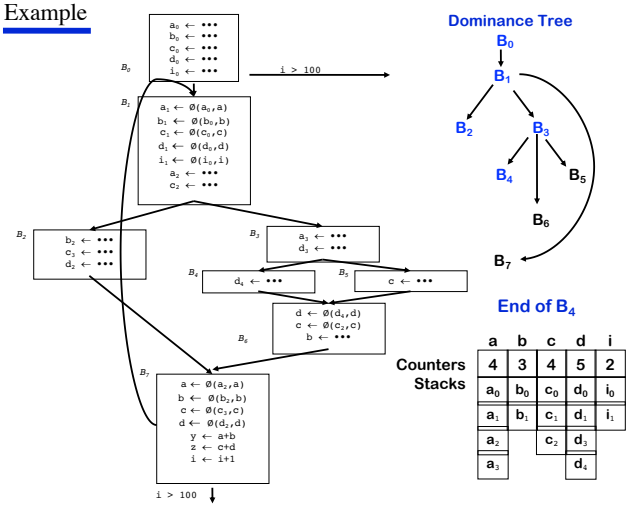
Example



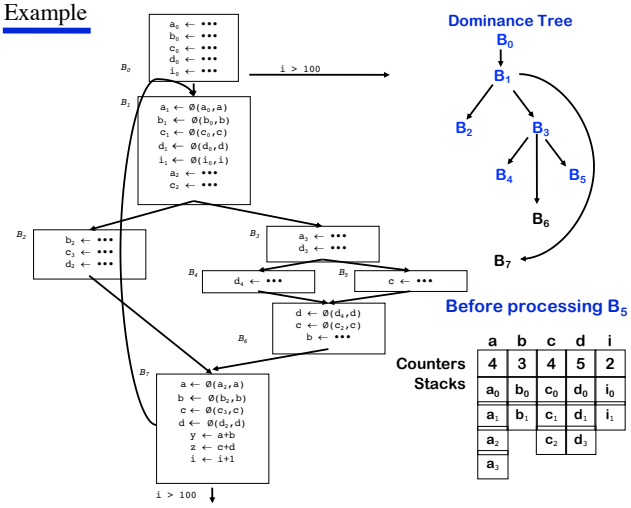
Example



Example



Example

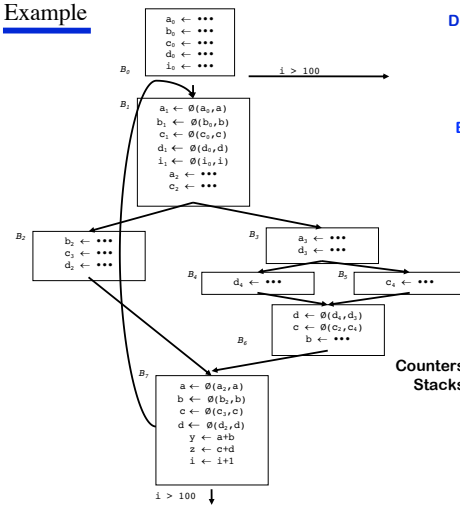


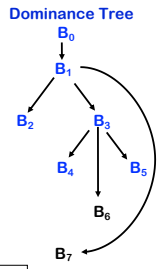
Before processing B_5

Counters
Stacks

a	b	c	d	i
4	3	4	5	2
a_0	b_0	c_0	d_0	i_0
a_1	b_1	c_1	d_1	i_1
a_2		c_2	d_3	
a_3				

Example



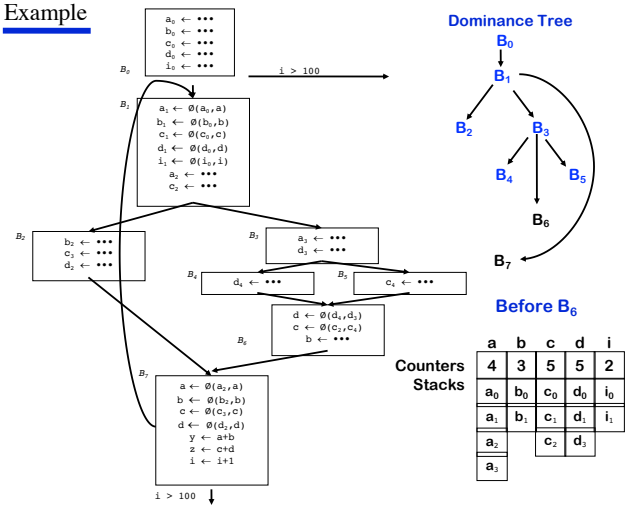


End of B_5

Counters
Stacks

a	b	c	d	i
4	3	5	5	2
a_0	b_0	c_0	d_0	i_0
a_1	b_1	c_1	d_1	i_1
a_2		c_2	d_3	
a_3		c_4		

Example

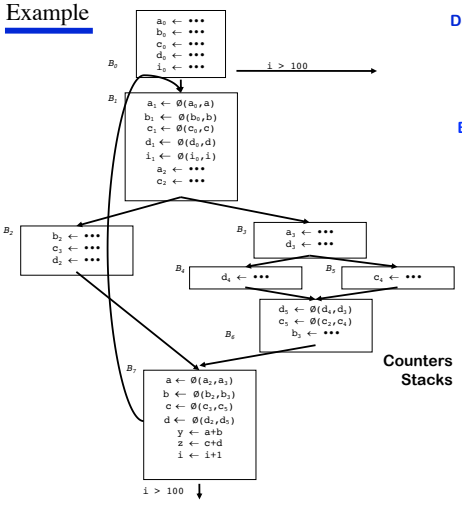


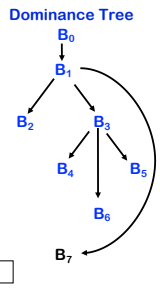
Before B_6

Counters
Stacks

a	b	c	d	i
4	3	5	5	2
a_0	b_0	c_0	d_0	i_0
a_1	b_1	c_1	d_1	i_1
a_2		c_2	d_3	
a_3				

Example



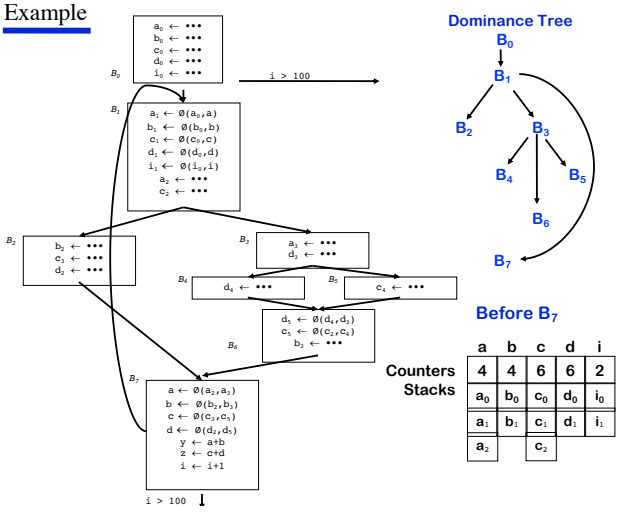


End of B_6

Counters
Stacks

a	b	c	d	i
4	4	6	6	2
a_0	b_0	c_0	d_0	i_0
a_1	b_1	c_1	d_1	i_1
a_2	b_3	c_2	d_3	
a_3		c_5	d_5	

Example

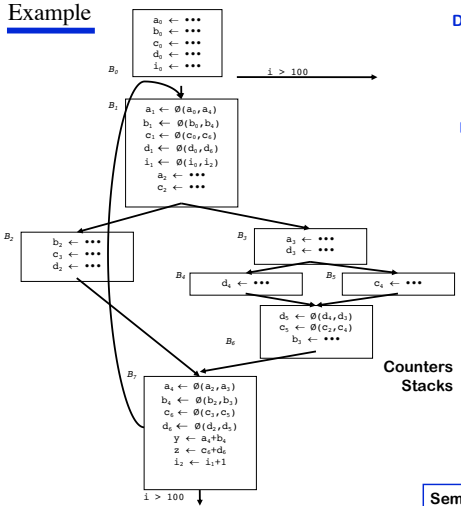


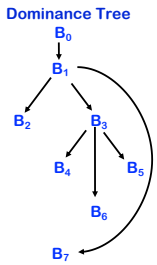
Before B_7

Counters / Stacks

a	b	c	d	i
4	4	6	6	2
a_0	b_0	c_0	d_0	i_0
a_1	b_1	c_1	d_1	i_1
a_2		c_2		

Example





End of B_7

Counters / Stacks

a	b	c	d	i
5	5	7	7	3
a_0	b_0	c_0	d_0	i_0
a_1	b_1	c_1	d_1	i_1
a_2	b_4	c_2	d_6	i_2
a_4		c_6		

Semi-pruned SSA, done!

Algorithm 5 Place-Φ-Functions

```
for each node n do
    for each variable a in A_orig[n] do
        defsites[a] ← defsites[a] ∪{n}
    end for
end for
for each variable a do
    W ← defsites[a]
    while W not empty do
        remove some node n from W
        for each y in DF[n] do
            if y ∉ A_Φ[a] then
                insert the statement a ← Φ(a, a, . . . , a) at the top of block y, where the Φ-function
has as many arguments as y has predecessors
                A_Φ[a] ← A_Φ[a] ∪ {y}
                if a ∉ A_orig[y] then
                    W ← W ∪ {y}
                end if
            end if
        end for
    end while
end for
```

7.8 Edge Splitting

Some analyses and transformations including reverse transformation from SSA back into a normal form are simpler if there is never a controlflow edge that leads from a node with multiple successors to a node with multiple predecessors. To give the graph this unique successor or predecessor property, we perform the following transformation: For each control-flow edge a ← b such that a has more than one successor and b has more than one predecessor, we create a new, empty controlflow node z, and replace the a ← b edge with an a ← z edge and a z ← b edge.

An SSA graph with this property is in edge-split SSA form. Figure 22 illustrates edge splitting. Edge splitting may be done before or after insertion of Φ-functions.

Algorithm 6 Renaming variables.

```
Initialization:
for each variable a do
    Count[a] ← 0
    Stack[a] ← empty
    push 0 onto Stack[a]
end for
Rename(n)
for each statement S in block n do
    if S is not a Φ-function then
        for each use of some variable x in S do
            i ← top(Stack[x])
            replace the use of x with x_i in S
        end for
    end if
    for each definition of some variable a in S do
        Count[a] ← Count[a]+1
        i ← Count[a]
        push i onto Stack[a]
        replace definition of a with definition of a_i in S
    end for
end for
for each successor Y of block n, do
    Suppose n is the jth predecessor of Y
    for each Φ-function in Y do
        suppose the jth operand of the Φ-function is a
        i ← top(Stack[a])
        replace the jth operand with a_i
    end for
end for
for each child X of n do
    Rename(X)
end for
for each definition of some variable a in the original S do
    pop Stack[a]
end for
```

8 SSA-Style optimizations

8.1 Constant Propagation

notes

- If v ← c , replace all uses of v with c
- If v ← Φ (c,c,c) (each input is the same constant), replace all uses of v with c

Algorithm 7 SSA-CP

```
W ← ist of all defs
while !W.isEmpty do
    Stmt S ← W.removeOne
    if (S has form v ← c) or (S has form v ← Φ(c,...,c)) then
        delete S
        for each stmt U that uses v do
            replace v with c in U
            W.add(U)
        end for
    end if
end while
```

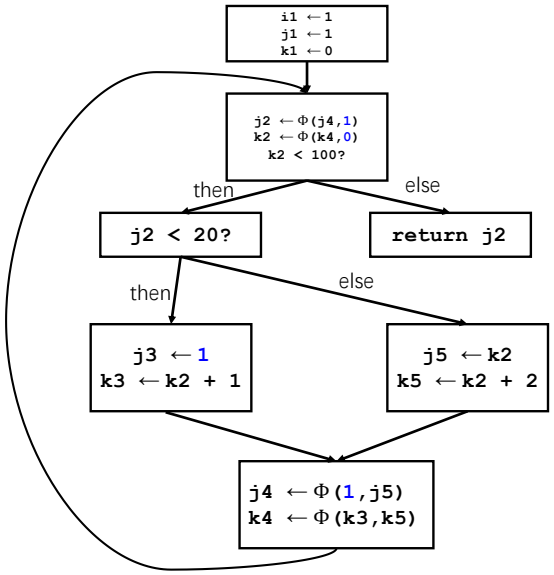
8.2 Conditional Constant Propagation

Wegman and Zadeck's Sparse Conditional Constant (SCC) algorithm was used to find constant expressions, constant conditions, and unreachable code [WZ91]. The output of the SCC algorithm is an association of variables to one of $\{\bot, c, \top\}$, where $\bot$ marks a variable that can hold different values at different times, and $\top$ means the variable is not executed. In addition, every flow-graph node (corresponding to a quadruple) is marked as executable or non-executable. We then walk the flow-graph, eliminating dead-code (quadruples marked non-executable), replacing constant variables with their values, and changing constant conditional branches to goto statements.

notes

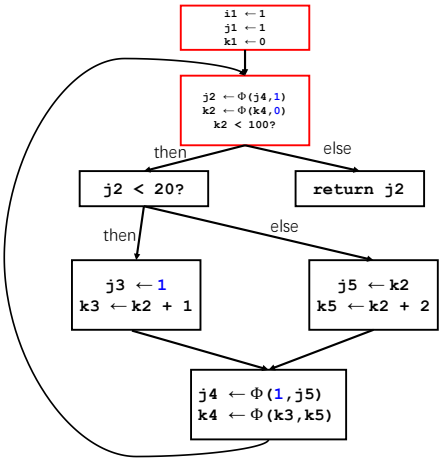
- Assume all blocks unexecuted until proven otherwise
- Assume all variables are not executed (only with proof of assignment of a non-constant value do we assume not constant)

8.2.1 Example



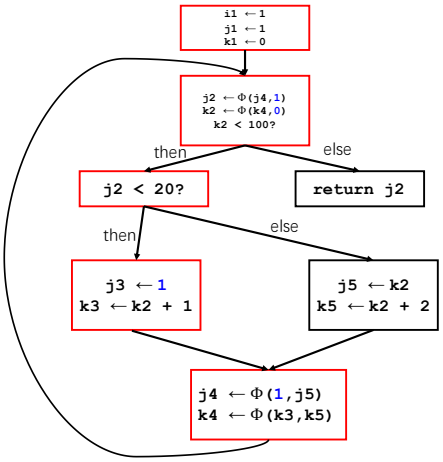
variable	value
i1	T
j1	T
j2	T
j3	T
j4	T
j5	T
k1	T
k2	T
k3	T
k4	T
k5	T

Figure 27: Original code. The black block is marked as unexecuted



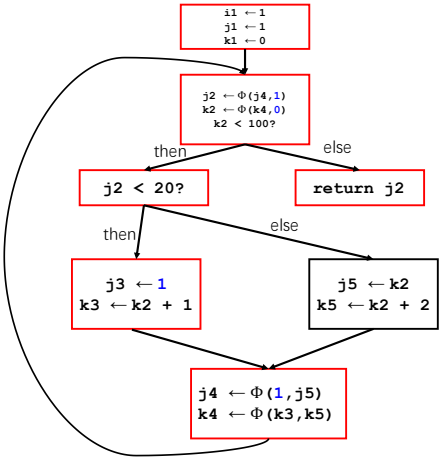
variable	value
i1	1
j1	1
j2	1
j3	T
j4	T
j5	T
k1	0
k2	0
k3	T
k4	T
k5	T

Figure 28: The read block is marked as executed. After walking the first two blocks, the value is shown above.



variable	value
i1	1
j1	1
j2	1
j3	1
j4	1
j5	T
k1	0
k2	0
k3	1
k4	1
k5	T

Figure 29: After walking 5 blocks.



variable	value
i1	1
j1	1
j2	1
j3	1
j4	1
j5	T
k1	0
k2	⊥
k3	1
k4	1
k5	T

Figure 30: Now k2 is ⊥, so the `return j2` is reachable.

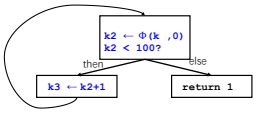


Figure 31: Code after applied SCC.

8.3 Copy Propogation

notes

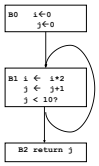
- delete x ← Φ (y,y,y) and replace all x with y
- delete x ← y and replace all x with y

8.4 Aggressive Dead Code Elimination

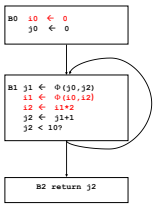
We can easily define the standard algorithm 8 below, but this algorithm may leave zombies. Look at the example in Figure 32

Algorithm 8 Dead Code Elimination

```
W ← list of all defs
while !W.isEmpty do
    Stmt S ← W.removeOne
    if |S.users|! = 0 then
        continue
    end if
    if S.hasSideEffects() then
        continue
    end if
    for def in S.operands.definers do
        def.users ← def.users - {S}
        if |def.users| == 0 then
            W ← W UNION {def}
        end if
    end for
    delete S
end while
```



(a) Original code. We can easily find that instructions relating to i are dead and can be eliminated.



(b) SSA format code. Since there is a circle use chain so we can not remove instructions relating to i because i_1 uses i_2, and i_2 uses i_1

.

Figure 32: An example to illustrate standard DCE can leave zombies.

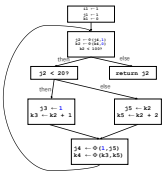
So instead of assuming everything is live until proven dead, we go another way: assuming everything is dead until proven live shown in algorithm 9.

Algorithm 9 Aggressive Dead Code Elimination

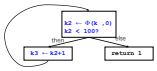
```
function INIT
    mark as live all stmts that have side-effects:
        I/O
        stores into memory
        returns
        calls a function that MIGHT have side-effects
    As we mark S live, insert S.operands.definers into W
    while |W| > 0 do
        S ← W.removeOne()
        if (S is live) then
            continue
        end if
        mark S live, insert S.operands.definers into W
    end while
end function
```

8.4.1 Problems within algorithm 9

After ADCE 9 shown in 33c, there is only one `return` statement left. However, control flow is undecidable in general, so possibly the loop in 33b will iterate indefinitely and the `return` instruction will never be executed. The problem here is we simply mark the branch statement dead.



(a) Original code in SSA format.



(b) After CCP

(c) After ADCE

Figure 33: An example to illustrate the algorithm 9 has a problem.

Also when we apply this algorithm to 32, we can find that `j2 < 10` is marked dead which is wrong. Of course, we can simply mark all branches live in the initialize stage, but this is not the ideal solution.

Now we need to carefully consider which conditional branches need to be marked live.

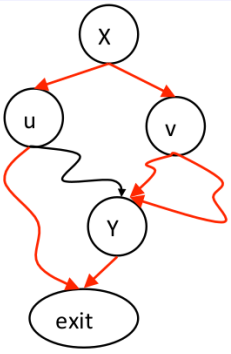
8.4.2 Control Dependence

control dependence

Y is control-dependent on X if

- X branches to u and v
- $\exists$ a path u $\rightarrow$ exit which does not go through Y
- $\forall$ paths v $\rightarrow$ exit go through Y

This means X casn determine whether or not Y is executed.



8.4.3 Aggressive Dead Code Elimination(Fixed Version)

So we make a little modification 10. When we mark S is live, we should also mark live those conditional branches upon which S is control dependent.

Algorithm 10 Aggressive Dead Code Elimination(Fixed Version)

```
function INIT
    mark as live all stmts that have side-effects:
        I/O
        stores into memory
        returns
        calls a function that MIGHT have side-effects
    As we mark S live, insert S.operands.definers into W
    S.CD^-1 into W
    while |W| > 0 do
        S ← W.removeOne()
        if (S is live) then
            continue
        end if
        mark S live, insert S.operands.definers into W
        S.CD^-1 into W
    end while
end function
```

8.4.4 Finding the Control Dependence Graph

- Construct CFG
- Add entry node and exit node
- Add (entry, exit) edge
- Create G′, the reverse CFG
- Compute D-tree in G′ (post-dominators of G)
- Compute $DF'_G(y)$ for all y ∈ G′ (post-DF of G)
- Add (x,y)∈ G to CDG if x ∈ $DF'_G(y)$

So let us calculate the control dependence for Figure 32a which is shwon in Figure 34. Since Block1 is control dependent on Block1, so the conditional branch in Block1 `j2 < 10 ?` should be marked live now.

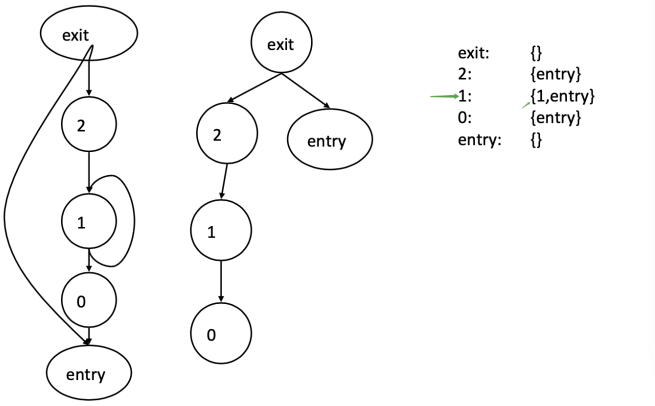


Figure 34: From left to right are G′, post-dominators of G and post-DF of G respectively.

9 The LLVM project

LLVM is an open-source framework providing a modern collection of modular and reusable compiler and toolchain technologies [20]. The project stemmed from the work of Chris Lattner, who first implemented core elements of LLVM to support the research of his master thesis in 2002 [26]. One of the key strengths of LLVM is that it faces active development from an expert community of contributors, and is widely used across the industry and academia alike [11]. The project received the ACM Software System Award in 2012 [4] as an acknowledgement of its contribution to compiler research and implementation.

LLVM officially acknowledges more than 10 main sub-projects [20], which range in diversity from a debugger [9] to a symbolic execution tool [8]. In addition to these, the official website presents a long list of miscellaneous projects that are based on components of the LLVM infrastructure [19].

All projects which are part of the LLVM ecosystem are built upon the core libraries, which are arguably the centrepiece of LLVM. They host the source- and target-independent optimizer (Section 3.3), the implementation of the LLVM intermediate representation (Section 3.2), and a suite of command-line tools useful for code manipulation (Section 3.4). The core libraries also implement various back-end passes that translate IR to machine code for different platforms (x86, PowerPC, Nvidia GPUs). For building a complete compiler for a source language, the LLVM core libraries readily provide the optimizing pass and code generation for common architectures, the frontend being the only missing component. Clang is by far the most prominent front-end implemented in LLVM [5], and it targets the family of C languages (C/C++ and Objective-C/C++). Coupled with the core libraries, Clang is a powerful compiler producing high-performance code, and positions itself as a direct competitor to both gcc and the Intel compiler.

10 Loop Invariant Computation and Code Motion

Loop-Invarian Code Motion (LICM) recognizes computations within a loop that produce the same result each time the loop is executed. These computations are called loop-invariant code and can be moved outside the loop body without changing the program semantics. The positive effects of LICM are:

- The shifted loop invariants exhibit a reduced execution frequency.
- The transformation may shorten variable live ranges leading to a decreased register pressure. This circumstance may in turn reduce the number of required spill code instructions.
- Moving code outside a loop reduces the loop's size which may be beneficial for the I-cache behavior since more loop code can reside in the cache.

Besides these positive effects on the code, LICM may also degrade performance. This is mainly due to two reasons. First, the newly created variables to store the loop-invariant results outside the loop, may increase the register pressure in the loops since their live ranges span across the entire loop nest. As a result, possibly additional spill code is generated. Second, LICM might lengthen other paths of the control flow graph. This situation can be observed if the invariants are moved from a less executed to a more frequently executed path, e.g., moving instructions above a loop's zero-trip test.

10.1 Finding natrual loops

Not every cicle is a loop in CFG. From a intuitive perspective, a loop must has a single entry and edges must from at least one circle.

Back Edge

A back edge is an arc t→h whose head h dominates its tail t

Natural Loop

The natural loop of a back edge t→h is the smallest set of nodes that includes t and h, and has no predecessors outside the set, except for the predecessors of the header h.

Reducible

A flow graph is reducible if every retreating edge in any DFST (Deep-First Spanning Tree) for that flow graph is a back edge.

Testing reducibility Take any DFST for the flow graph, remove the back edges, and check that the result is acyclic.

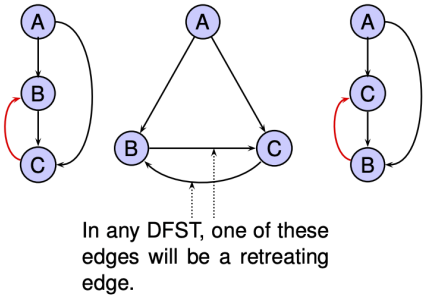


Figure 35: Example: Nonreducible Graph

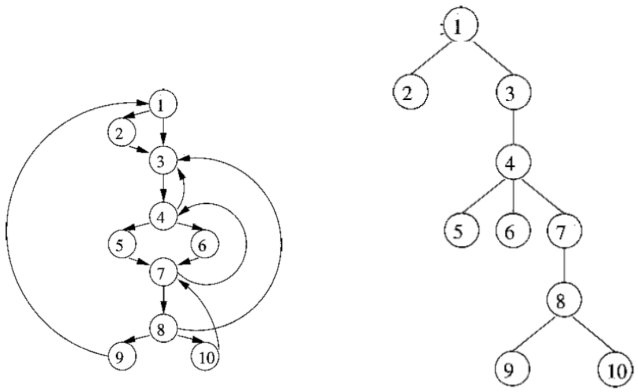
10.2 Algorithm to Find Natural Loops

10.2.1 Step 1. Finding Dominators

We can formulate this as Data Flow Analysis problem. Since Node d dominates node n in a graph (d dom n) if every path from the start node to n goes through d. So if d dom n iff iff d dom p for all pred p of n.

Direction	Forward
Values	Basic Blocks
Meet operator	$\cap$
Top(T)	Universal Set
Bottom	ϕ
Boundary condition for entry node	ϕ
Initialization for internal nodes	T
Finited escending chain?	✓
Transferfunction	OUT $[\mathbf{b}] = \{\mathbf{b}\} \cup \left(\cap_{\{p=pred(b)\}} \text{OUT } [\mathbf{p}]\right)$
Monotone&Distributive?	✓

With rPostorder, most flow graphs (reducible flow graphs) converge in 1 pass.



(a) CFG.

(b) Dominance Tree of the corresponding CFG.

Figure 36: An example of Dominance tree.

10.2.2 Step 2. Finding Back Edges

Depth-first spanning tree Edges traversed in a depth-first search of the flow graph form a depth-first spanning tree. We categorize edges in CFG as follows:

- Forward edges (node to proper descendant).
- Retreating edges (node to ancestor).
- Cross edges (between two nodes, neither of which is an ancestor of the other.)

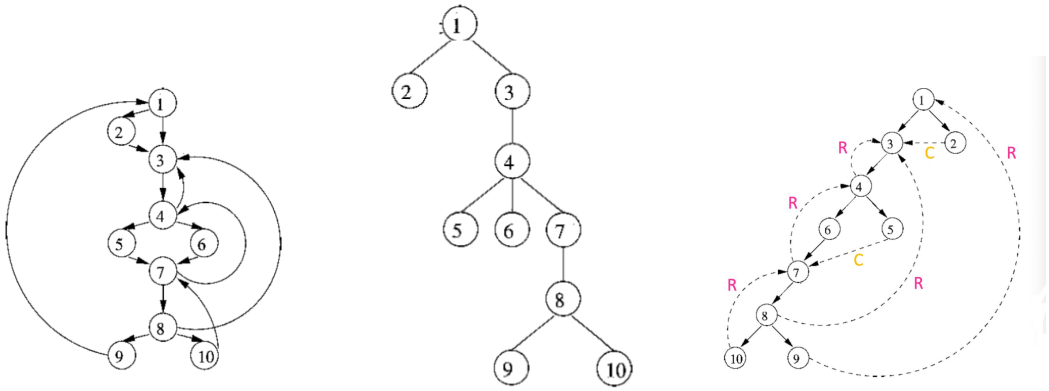
This is something difficult to understand. Let's make it simpler. We can number each node when we visit it. So each edge should be satisfied the following property:

Forward/Advancing edges $n_1 \to n_2$	$\text{num}(n_1) < \text{num}(n_2)$ and n_1 is ancestor of n_2
Cross edges $n_1 \to n_2$	$\text{num}(n_1) > \text{num}(n_2)$ and neither n_1 is ancestor of n_2 nor n_2 is ancestor of n_1
Retreating edges $n_1 \to n_2$	$\text{num}(n_1) > \text{num}(n_2)$ and n_2 is ancestor of n_1

Of these edges, only retreating edges go from high to low in DF order.

Algorithm

- Perform a depth first search
- For each retreating edge $t \to h$, check if h is in t′s dominator list



(a) CFG.

(b) Dominance Tree of the corresponding CFG.

(c) R means back edges and C means cross edge.

Figure 37: An example of Back Eges.

10.2.3 Step 3. Constructing Natural Loops

Algorithm For each back edge $t \to h$:

- delete h from the flow graph
- find those nodes that can reach t (those nodes plus h form the natural loop of $t \to h$)

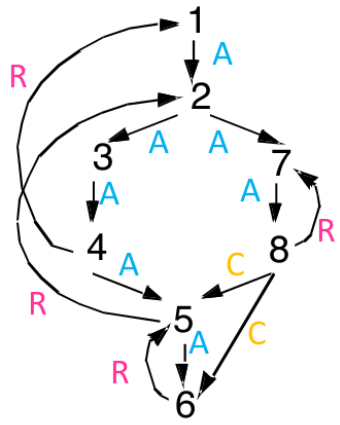


Figure 38: For this flow graph, for back edge $8 \to 7$, natrual loop is { 7,8 }, for back edge $5 \to 2$, natrual loop is { 2,3,4,5,6,7,8 },back edge $4 \to 1$, natrual loop is { 1,2,3,4,5,6,7,8}.

10.3 Inner Loops

If two different loops don't have ths same header, they are either disjoint or one is entirely contained the other (inner loop is the one that contains no other loop.). If two loops share the same header shown in reffig:p65, it is hard to tell which is the inner loop. But we can combine and treat as one loop.

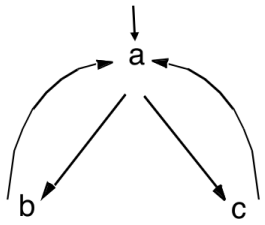


Figure 39: Two loops share the same header.

10.4 Loop-Invariant Computation and Code Motion

Loop-Invariant Computation

A loop-invariant computation is a computation whose value doesn't change as long as control stays within the loop. loop invariant whose operands are defined outside loop or invariant themselves.

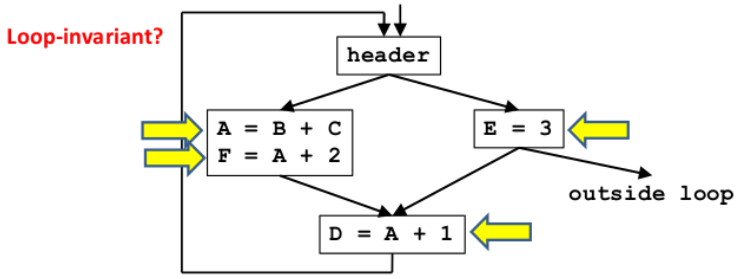


Figure 40: For this CFG, `A = B+C, F=A+2, E=3` are Loop-Invariant Computation, but `D = A + 1` is not.

Not all loop invariant instructions can be moved to preheader.

10.5 LICM Algorithm

- Find invariant expressions
- Conditions for code motion

- Code transformation

10.6 Find invariant expressions

- Compute reaching definitions
- Repeat: mark `A = B + C` as invariant if
  - All reaching definitions of B are outside the loop or there is exactly one reaching definition for B and it is from a loop-invariant statement inside the loop.
  - Check the same for C.
- Code transformation

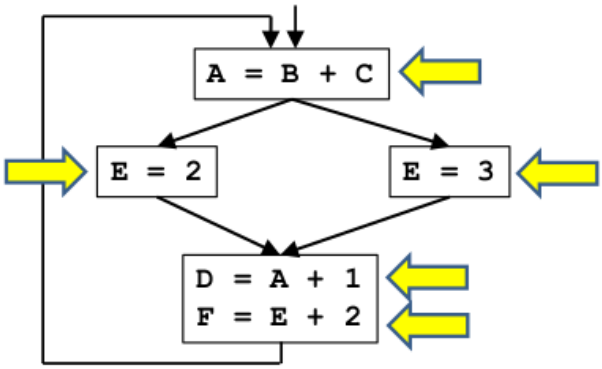


Figure 41: For this CFG, `A = B+C`, `E = 2`, `E = 3`, `D = A + 1` are Loop-Invariant Computation, but `F = E + 2` is not because two definitions of E reach `F = E + 2`.

10.7 Conditions for Code Motion

Algorithm 11 Code Motion Algorithm

Given: a set of nodes in a loop
Compute reaching definitions
Compute loop invariant computation
Compute dominators
Find the exits of the loop (i.e. nodes with successor outside loop)
Candidate statement for code motion:
    loop invariant
    in blocks that dominate all the exits of the loop shown in 42
    assign to variable not assigned to elsewhere in the loop
    in blocks that dominate all blocks in the loop that use the variable assigned shown in 43
Perform a depth-first search of the blocks
    Move candidate to preheader if all the invariant operations it depends upon have been moved

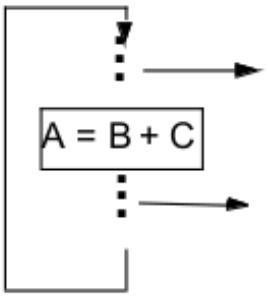


Figure 42: It is not safe to move `A = B + C` outside the loop because we can jump of the loop before executing `A = B + C`

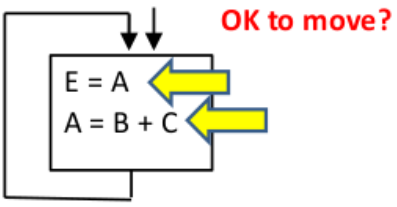


Figure 43: It is not safe to move `A = B + C` outside the loop because if so, the first time we enter the loop `E = A` will not the same as before.

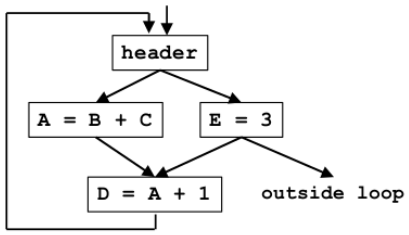


Figure 44: Only `E = 3` can be moved outside the loop.

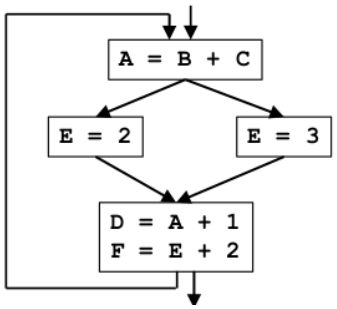


Figure 45: Only `A = B + C , D = A + 1` can be moved outside the loop.

10.8 More Aggressive Optimizations

10.8.1 Gamble on: most loops get executed

We can relax constraint of dominating all exits on some cases.

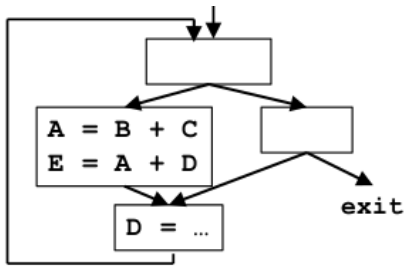


Figure 46: `A = B + C` cannot be removed outside the loop because it doesn't dominat the exit. But if A is not live after the loop, we can actually move it to the preheader. The only thing we need to consider is that this statement can cause an exception.

10.8.2 Landing pads

`while` loop is not very convenient for optimization since it checks before entering the loop. We can use landing pads to solve this.

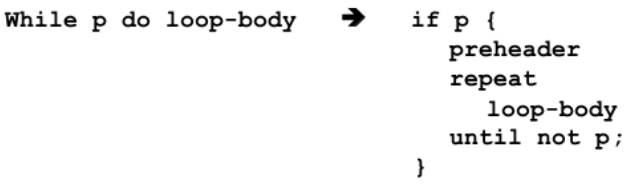


Figure 47: Transforms for while loops.

11 Induction Variables and Strength Reduction

Strength reduction is an optimization technique which substitutes expensive operations with computationally cheaper ones. For example, a very weak strength reduction algorithm can substitute the instruction `b = a * 4` with `b = a << 2`.

11.1 Motivation

Opportunities for strength reduction arise routinely from details that the compiler inserts to implement source-level abstractions. To see this, consider the simple code fragment shown in Figure 64. Figure 48a shows source code and the same loop in a low-level intermediate code. Notice the instruction sequence that begins at the label L2. The compiler inserted this code (with its multiply) as the expansion of A[i]. Figure 48b shows the code that results from applying Strength Reduction, Figure 48c is followed by dead-code elimination. The compiler created a new variable, $t2'$, to hold the value of the expression $i * 4 + A$. Its value is computed directly, by incrementing it with the constant 4, rather than recomputing it on each iteration as a function of i. Strength reduction automates this transformation.

```
for(i=0; i<100; i++)
       A[i] = 0;
```

```
    i = 0
L2: IF i>=100 GOTO L1
    t1 = 4 * i
    t2 = &A + t1
    *t2 = 0
    i = i+1
    GOTO L2
L1:
```

(a) Origianl code.

```
t1' = 0
t2' = &A
L2: IF t1'>=400 GOTO L1
    t1 = t1'
    t2 = t2'
    *t2 = 0
    t1' = t1'+4
    t2' = t2'+4
    GOTO L2

L1:
```

(b) After induction variable substitute.

```
       t2' = &A
       t3' = &A + 400
L2:    IF t2'>=t3' GOTO L1 *t2'= 0
       t2' = t2'+ 4
       GOTO L2
L1:
```


(c) Final code.

Figure 48: An example of strength reduction.

11.2 Definitions

Basic Induction Variable

A basic induction variable (e.g., i as shown in Figure 48a) is a variable X whose only definitions within the loop are assignments of the form: X = X+c or X = X-c, where c is either a constant or a loop-invariant variable.

Induction Variable

An induction variable is either a basic induction variable B, or or a variable defined once within the loop (e.g., t1,t2 as shown in Figure 48a) , whose value is a linear function of some basic induction variable at the time of the definition: $A = c_1 * B + c_2$

The FAMILY of a basic induction variable B is the set of induction variables A such that each time A is assigned in the loop, the value of A is a linear function of B. (e.g., t1, t2 is in family of i as shown in Figure 48a)

11.3 Optimizations

11.3.1 Strength Reduction

Algorithm 12 Strength Reduction Optimizations

A is an induction variable in family of basic induction variable B (i.e., $A = c_1 * B + c_2$)
Create new variable A′
Initialize in preheader A′ $= c_1 * B + c_2$
Track value of B: add after $B = B + x$: $A' = A' + x * c_1$
Replace assignment to A: replace lone $A = \ldots$ with $A = A'$

11.3.2 Optimizing non-basic induction variables

- copy propagation
- dead code elimination

11.3.3 Optimizing basic induction variables

Eliminate basic induction variables used only for calculating other induction variables and loop tests.

Algorithm 13 Optimizing basic induction variables

Select an induction variable A in the family of B, preferably with simple constants ($A = c_1 * B + c_2$).
Replace a comparison such as `if B > X goto L1` with `if (A′ >` $c_1 * X + c_2$`) goto L1` (assuming c 1 is positive)
if B is live at any exit from the loop, recompute it from A′, After the exit, $B = (A' - c2)/c1$

11.4 Further Details

```
k = 0;
for (i = 0; i < n; i++){
       k = k+3 ;
       … = m;
       if(x<y)
         k = k+4
       if(a < b)
         m = 2 * k;
       k = k - 2;
       … = m;
}
```

(a) A more complex example. k and i are both basic induction variables. m is in the family of k.

```
k = 0;
m' = 0;
for (i = 0; i < n; i++){
       k = k+3 ;
       m' = m'+6;
       … = m;
       if(x<y)
         k = k+4
         m' = m'+8;
       if(a < b)
         m = m' ;
       k = k - 2;
       m' = m'-4;
       … = m;
}
```

(b) After induction variable substitute.

Figure 49: A more complex example of strength reduction.

11.5 Finding Induction Variable Families

Let B be a basic induction variable, A is in the family of B if it satisfies one the following conditions

- **Condition C1** A has a single assignment in the loop L of the form A = B*c, c*B, B+c, etc
- **Condition C2** A is in family of B if $D = c_1 * B + c_2$ for basic induction variable B and:
  - Rule 1: A has a single assignment in the loop L of the form A = D*c, D+c, etc
  - Rule 2: No definition of D outside L reaches the assignment to A
  - Rule 3: Every path between the lone point of assignment to D in L and the assignment to A has the same sequence (possibly empty) of definitions of B

```
L2: IF i>=100 GOTO L1
    t2 = t1 + 10
    t1 = 4 * i
    t3 = t1 * 8
    i = i + 1
    goto L2
L1:
```

Figure 50: i is abasic induction variable, t1 t2 are in family ofi, but t2 is not because it violates the condition C2 rule 2.

```
L3: IF i>=100 GOTO L1
    t1 = 4 * i
    IF t1 < 50 GOTO L2
    i = i + 2
L2: t2 = t1 + 10
    i = i + 1
    goto L3
L1:
```

Figure 51: i is abasic induction variable, t1 is in the family of i. t2 is not because it violates the Condition2 rule3(some path reaches t2 includes `i = i+1` but some not.).

12 Partial Redundancy Elimination

Partial redundancy elimination (PRE) is a global optimization introduced by Morel and Renvoise[1]. It combines and extends two other techniques: common subexpression elimination and loop-invariant code motion.

An expression is partially redundant at point p if it is redundant along some, but not all, paths that reach p. PRE converts partially-redundant expressions into redundant expressions. The basic idea is simple. First, it uses data-flow analysis to discover where expressions are partially redundant. Next, it solves a data-flow problem that shows where inserting copies of a computation would convert a partial redundancy into a full redundancy. Finally, it inserts the appropriate code and deletes the redundant copy of the expression.

A key feature of PRE is that it never lengthens an execution path. To see this more clearly, consider the example shown in Figure 52. In the fragment on the left, the second computation of `x + y` is partially redundant; it is only available along one path from the if. Inserting an evaluation of `x + y` on the other path makes the computation redundant and allows it to be eliminated, as shown in the right-hand fragment. Note that the left path stays the same length while the right path has been shortened.

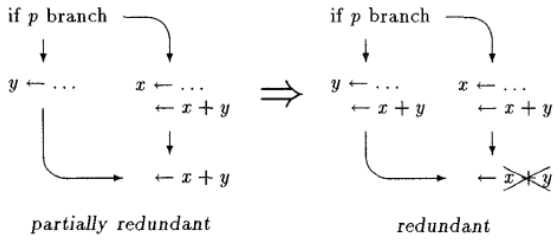


Figure 52

Loop-invariant expressions are also partially redundant, as shown in Figure 53. On the left, `x + y` is partially redundant since it is available from one predecessor (along the back edge of the loop), but not the other. Inserting an evaluation of `x + y` before the loop allows it to be eliminated from the loop body.

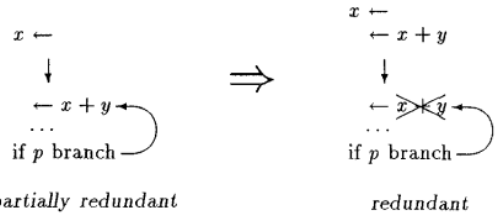


Figure 53

12.1 Finding Partially Available Expressions

For every expression, we can do a dataflow analysis.

Direction	Forward
Meet operator	$\cup$
Lattice	$\{0, 1\}$
Top(T)	0
Bottom	1
Boundary condition for entry node	0
Initialization for internal nodes	T
Finited escending chain?	✓
Transferfunction	$PAVOUT[i] = (PAVIN[i] - KILL[i]) \cup AVLOC[i]$
Monotone&Distributive?	✓
AVLOC	Expression is locally available (AVLOC) if downwards exposed.
KILL	Expression is killed (KILL) if any assignments to operands.

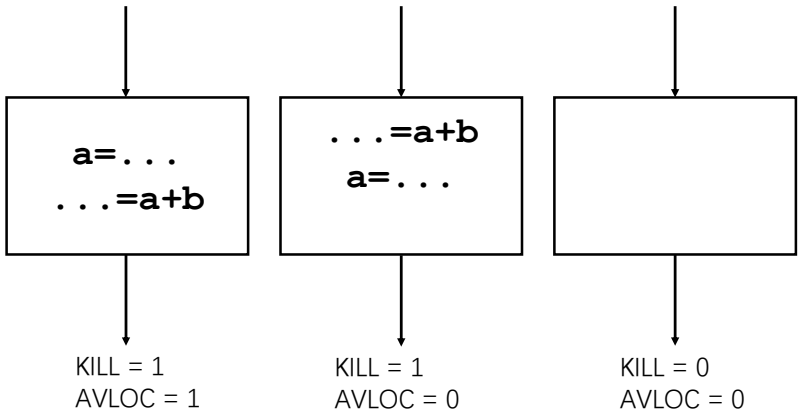


Figure 54: For `a+b`, the result of Partially Available Expressions's transfer function within a basic block.

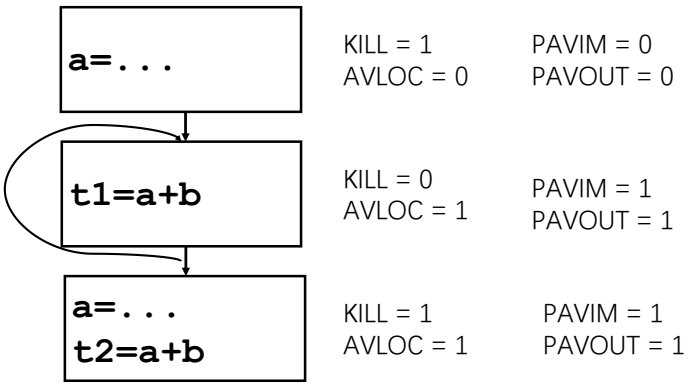


Figure 55: For `a+b`, the result Partially Available Expressions of dataflow analysis.

12.2 Finding Anticipated Expression

For PRE, care must be taken that the hoisting would do no harm: Never introduce a new expression along any path. Otherwise the hoisting will lengthen at least one trace of the program, defying optimality; even worse, if the hoisted instruction throws an exception, the program's semantics change.

Local Anticipability(ANTLOC)

An expression may be locally anticipated in a block i if there is at least one computation of the expression in the block i, and if the commands appearing in the block before the first computation of the expression do not modify its operands.

Direction	backward
Meet operator	$\cap$
Lattice	$\{0, 1\}$
Top(T)	1
Boundary condition for exit node	0
Initialization for internal nodes	T
Finited escending chain?	✓
Transferfunction	$ANTIN[i] = ANTLOC[i] \cup (ANTOUT[i] - KILL[i])$
Monotone&Distributive?	✓
ANTLOC	Expression is locally anticipated(ANTLOC)) is upward exposed.

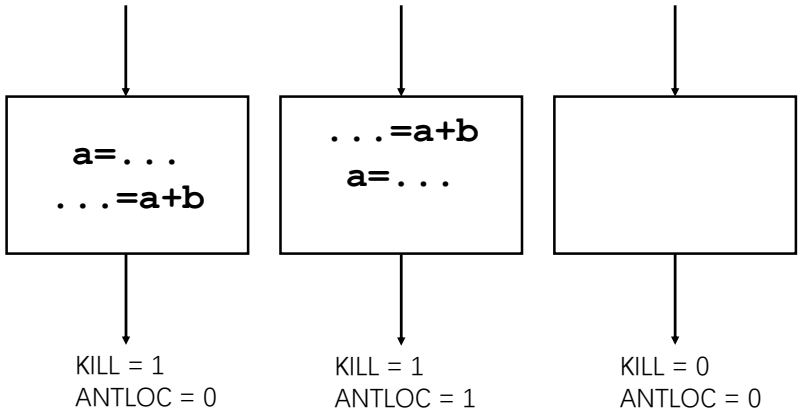


Figure 56: For `a+b`, the result of Anticipated Expression's transfer function within a basic block.

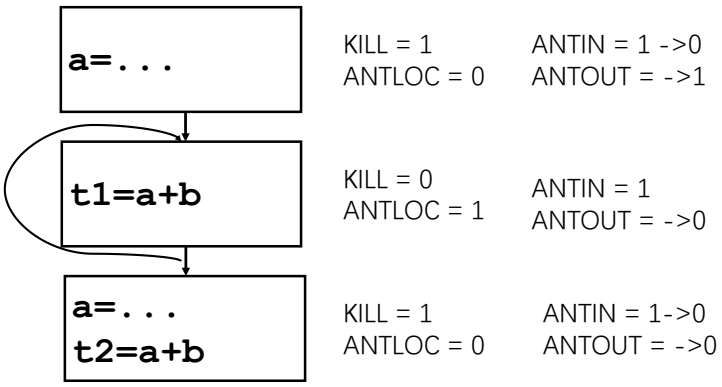


Figure 57: For `a+b`, the result Anticipated Expression of dataflow analysis.

12.3 Where Do we Want to Insert Computations?

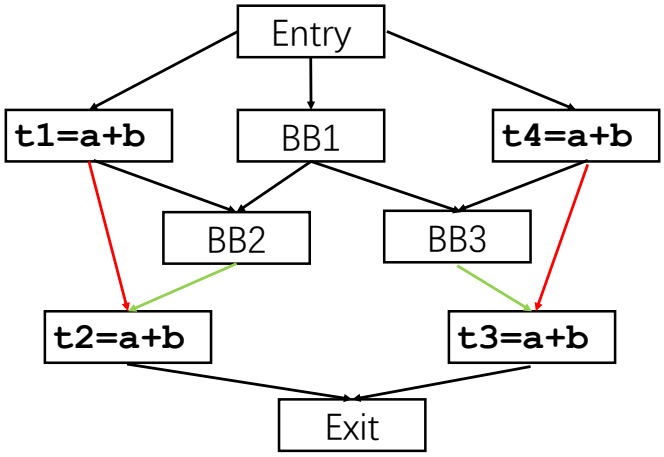


Figure 58: For `a+b`, `t2 = a + b` and `t3 = a + 4` are both partially redundant. But where can we insert the new computation `a + b` in order to optimally eliminate redundancy? The best choice is BB1 because this wil make `a + b` fully redundant. But if we insert to BB2 and BB3, there are some path that calculate `a + b` more than once (e.g. Entry→`t1= a+b` → BB2→`t2= a+b`)

We define "**Placement Possible** "(PP) dataflow analysis. First of all, we are only going to place computations at the end of the blocks. We want to insert the computation at the earliest place where our **Placement Possible** is true. If the **Placement Possible** is true output of a block(PPOUT), it means it is fine to insert at the end of this block or earlier. If it is true athe the beginning of the block, it means we could insert it at the beginning of the block or earlier. Becuase we want to insert it at the end of the block, so if **Placement Possible** is true at the beginning of the block(PPIN), you won't insert it in the block, it means you need to take care of something before. So when PPIN is true, it really means is for every predecessor block, it is either possible to keep moving it back and make it fully redundant by placing in that block or earlier or it is not necessary because it is already generatedin one of those blocks.

We insert if PPOUT is true and either PPIN is false so we cannot move it back any further or if we locally kill it then clearly we have to insert it here because trying to insert it earlier would not work. And we only want to insert it if it is not already available.

We want to delete an expression where PPIN is true (somehow we make it fully redundant) and it is anticipated locally.

For safety reasons, if we want to place at output of a block, we want to place at entry of all successors. (PPOUT)

$$\text{PPOUT[i]} = \begin{cases} 0 & i = \text{ entry} \\ \bigcap_{s \in \text{succ}(i)} \text{PPIN[s]} & \text{otherwise} \end{cases}$$

When PPIN is true, if means

- we have a local computation to place, or a placement at the end of this block which we can move up
- we want to move computation to output of all predecessors where expression is not already available (don't insert at input) (for every predecessor,)
- we gain something by moving it up (PAVIN heuristic) (not too far)

$$\text{PPIN[i]} = \begin{cases} 0 & i = \text{ exit} \\ ([\text{ ANTLOC[i]} \cup (\text{PPOUT } [i] - \text{KILL}[i])] & \\ \cap \bigcap_{p \in \text{ preds } (i)} (\text{ PPOUT } [p] \cup \text{ AVOUT } [p]) & \text{otherwise} \\ \cap \text{ PAVIN } [i]) & \end{cases}$$

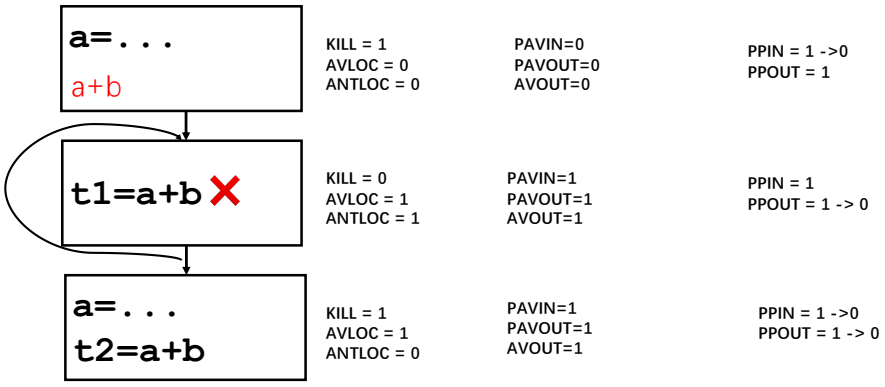


Figure 59: Example for PRE for `a+b`

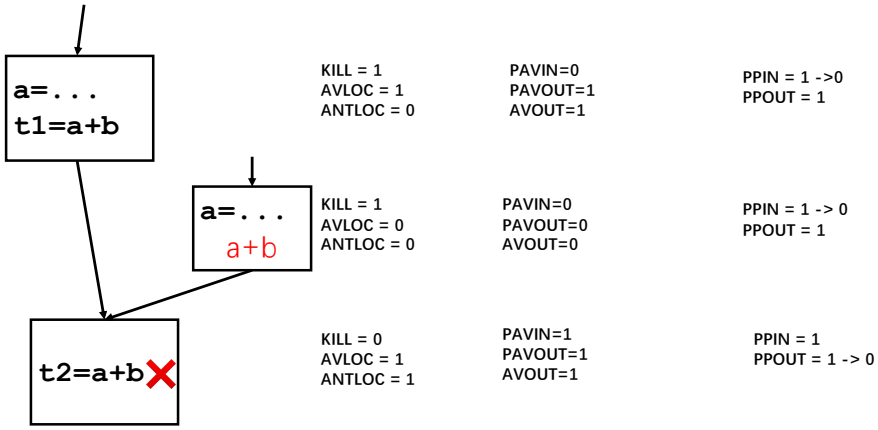


Figure 60: Example for PRE for `a+b`

12.3.1 Safety

It is safe to insert only if anticipated.

$$\text{PPIN[i]} \subseteq (\text{PPOUT[i]} - \text{KILL[i]}) \cup \text{ANTLOC[i]}$$

$$\text{PPOUT[i]} = \begin{cases} 0 & i = \text{entry} \\ \bigcap_{s \in \text{succ}(i)} \text{PPIN[s]} & \text{otherwise} \end{cases}$$

$$\text{INSERT} \subseteq \text{PPOUT} \subseteq \text{ANTOUT}$$

So it is safe.

12.4 Perform

On every path from an INSERT, there is a DELETE.

12.5 Limiations

12.6 A new way to think about partial redundancy

We want to insert the new computation where it is not partially available there.

Anticipable(Very Busy) Expression

An expression e is anticipable at a program point p if e will be computed along every path from p to p_{end}, and no variable in e is redefined until its computation. It is safe to move an expression to a basic block where that expression is anticipable. By "safe" we mean "performance safe", i.e., no extra computation will be performed. Notice that if an expression e is computed at a basic block where it is both available and anticipable, then that computation is clearly redundant.

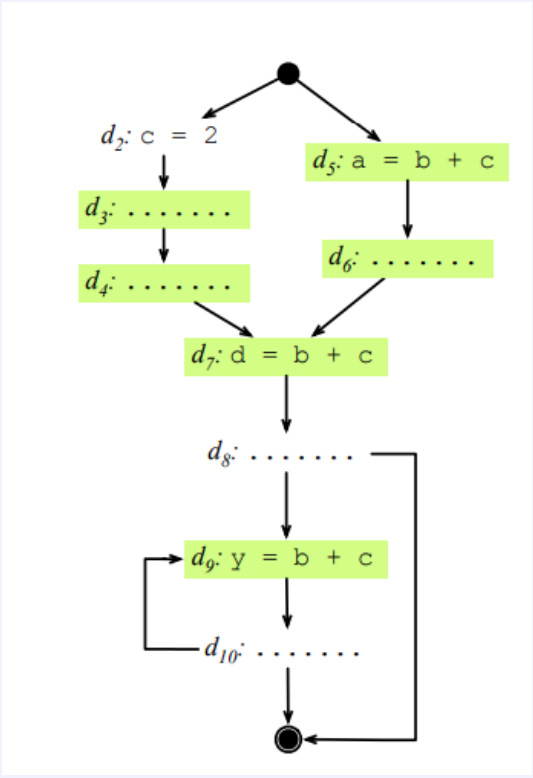


Figure 61: For `b+c`, the green blocks are anticipable points.

The key to partial redundancy elimination is deciding where to add computations of an expression to change partial redundancies into full redundancies (which may then be optimized away). There are now two steps that we must perform:

- First, we find the earliest places in which we can move the computation of an expression without adding unnecessary computations to the CFG. This step is like pushing the computation of the expressions up.
- Second, we try to move these computations down, closer to the places where they are necessary, without adding redundancies to the CFG. This phase is like pulling these computations down the CFG. So that we can, for instance, reduce register pressure.

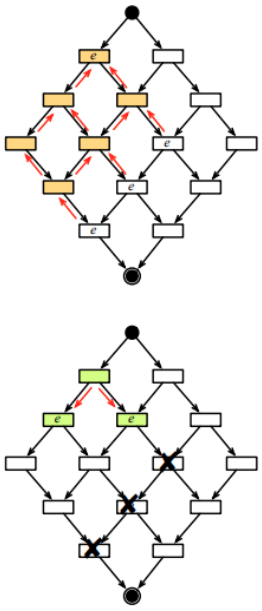


Figure 62: Pushing up, Pulling down.

12.6.1 Earliest Placemen

We must now find the earliest possible places where we can compute the target expressions. Earliest in the sense that p1 comes before p2 if p1 precedes p2 in any topological ordering of the CFG.

$$EARLIEST(i, j) = IN_{ANTICIPABLE}(j) \cap \overline{OUT_{AVAILABLE}(i)} \cap (KILL(i) \cup \overline{OUT_{ANTICIPABLE}(i)})$$

For the Fisrt part, We can move an expression e to an edge ij only if e is anticipabled at the entrance of j. If the expression is available at the beginning of the edge, then we should not move it there. But the Second part, If an expression is anticipable at i, then we should not move it to ij, because we can move it to before i. On the other hand, if i kills the expression, then it cannot be computed before i.

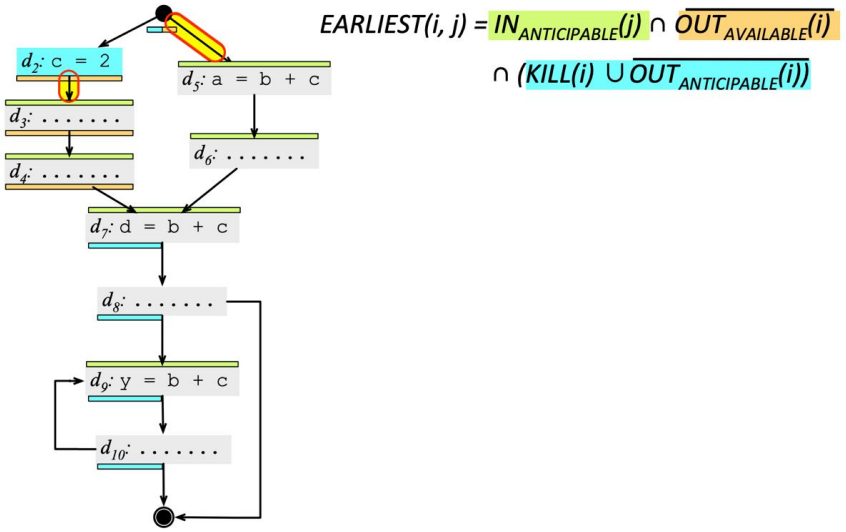


Figure 63: An example for calculating EARLIEST.

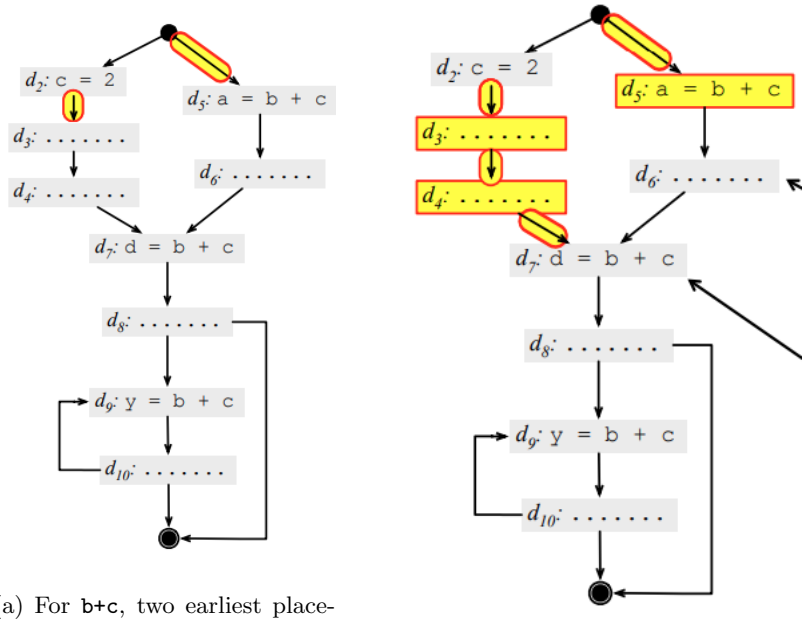
12.6.2 Latest Placement

$$\text{IN}_{\text{LATER}}(j) = \cap_{i \in \text{pred}(j)} \text{LATER}(i,j)$$
$$\text{LATER}(i,j) = \text{EARLIEST}(i,j) \cup \left(\text{IN}_{\text{LATER}}(i) \cap \overline{\text{EXPR}(i)}\right).$$

LATER(i,j) is true if we can move the computation of the expression down the edge ij. An expression e is in EXPR(i) if e is computed at i. This predicate is also computed for edges, although we have IN_{LATER} being computed for nodes.

For LATER(i,j): If EARLIEST(i, j) is true, then LATER(i,j) is also true, as we can move the computation of e to edge ij without causing redundant computations. If $IN_{\text{LATER}}(i,j)$ is true, and the expression is not used at i, then LATER(i,j) is true. If the expression is used at i, then there is no point in computing it at ij, because it will be recomputed at i anyway.

For $IN_{\text{LATER}}(i,j)$, it is a condition that we propagate down. If all the predecessors of a node j accept the expression as nonredundant, then we can compute the expression down on j.



(a) For `b+c`, two earliest placement points is colored in red.

(b) For `b+c`, Latest placement edhes and blocks.

Figure 64: A more complex example of strength reduction.

12.6.3 Where to Insert Computations?

We insert the new computations at the latest possible place.That is

$$INSERT(i, j) = LATER(i, j) \cap \overline{IN_{LATER}(j)}$$

There are different insertion points, depending on the structure of the CFG, if x ∈ INSERT(i, j):

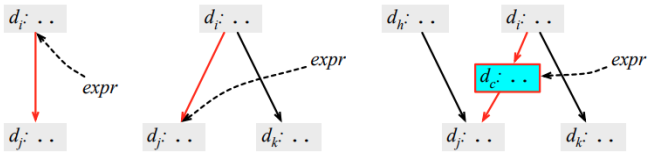


Figure 65: Different inser9on points

12.7 Modify CFG

Rename all compuation of the expression.

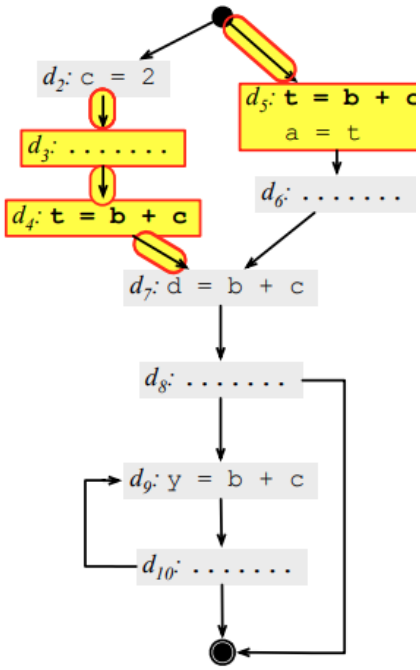


Figure 66: For `b+c`, the result of applyiny modifying CFG.

12.8 Which Computations to Remove?

We remove computations that are already covered by the latest points, and that we cannot use later on.

$$DELETE(i) = EXPR(i) \cap \overline{IN_{LATER}(i)}$$

For First part, of course, the expression must be used in the block, otherwise we would have nothing to delete. For second part, The expression may not be a computation that is necessary later on.

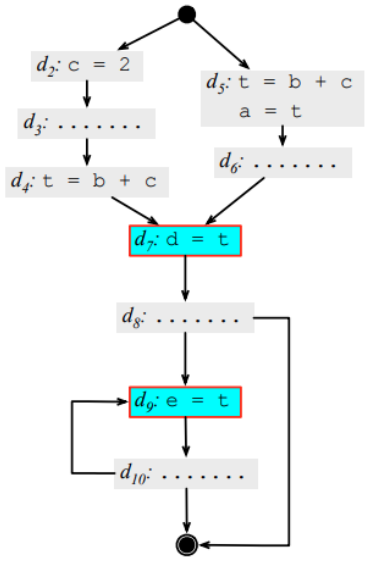
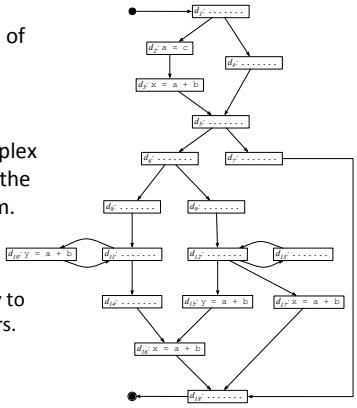


Figure 67: For `b+c`, the result of applyiny deleting redundancy `b+c`

12.9 A fully explained example

An Example to Conquer them All

- The original formulation of Lazy Code Motion was published in a paper by Knoop *et al*[◇].
- The authors used a complex example to illustrate all the phases of their algorithm.
- Many papers are build around examples.
  - That is a good strategy to convey ideas to readers.



◇: Lazy Code Motion, PLDI (1992)

Available Expressions

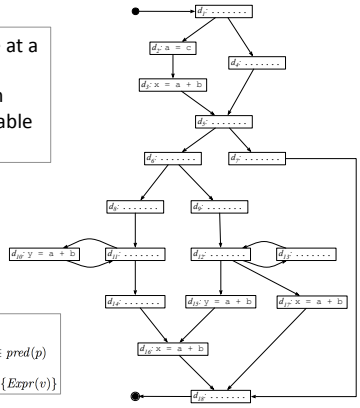
An expression *e* is available at a program point *p* if e is computed along every path from p_{start} to *p*, and no variable in *e* is redefined until *p*.

What is the OUT set of available expressions in the example?

$p : v = E$

$$IN(p) = \bigcap OUT(p_s), p_s \in pred(p)$$

$$OUT(p) = (IN(p) \cup \{E\}) \setminus \{Expr(v)\}$$



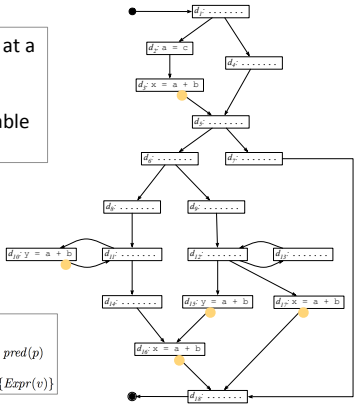
Available Expressions

An expression *e* is available at a program point *p* if e is computed along every path from p_{start} to *p*, and no variable in *e* is redefined until *p*.

$p : v = E$

$$IN(p) = \bigcap OUT(p_s), p_s \in pred(p)$$

$$OUT(p) = (IN(p) \cup \{E\}) \setminus \{Expr(v)\}$$



Anticipable Expressions

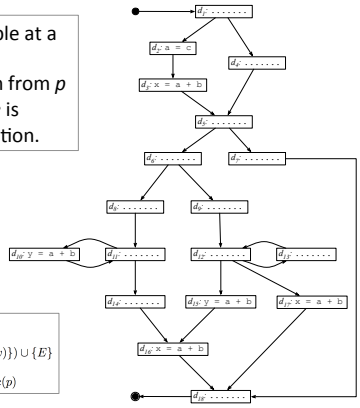
An expression *e* is anticipable at a program point *p* if *e* will be computed along every path from *p* to p_{end}, and no variable in *e* is redefined until its computation.

What is the IN set of anticipable expressions in the example?

$p : v = E$

$$IN(p) = (OUT(p) \setminus \{Expr(v)\}) \cup \{E\}$$

$$OUT(p) = \bigcap IN(p_s), p_s \in succ(p)$$



Anticipable Expressions

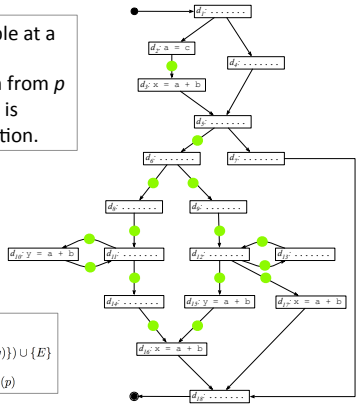
An expression *e* is anticipable at a program point *p* if *e* will be computed along every path from *p* to p_{end}, and no variable in *e* is redefined until its computation.

What is the IN set of anticipable expressions in the example?

$p : v = E$

$$IN(p) = (OUT(p) \setminus \{Expr(v)\}) \cup \{E\}$$

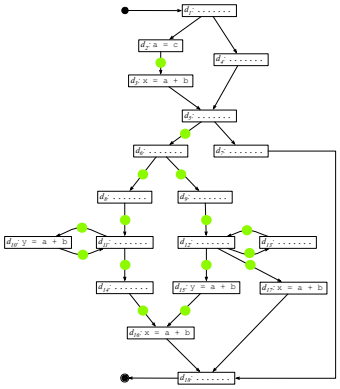
$$OUT(p) = \bigcap IN(p_s), p_s \in succ(p)$$



EARLIEST(i, j) = $IN_{ANTICIPABLE}$(j) ∩ ~~$OUT_{AVAILABLE}$(i)~~ ∩ (KILL(i) ∪ ~~$OUT_{ANTICIPABLE}$(i)~~)

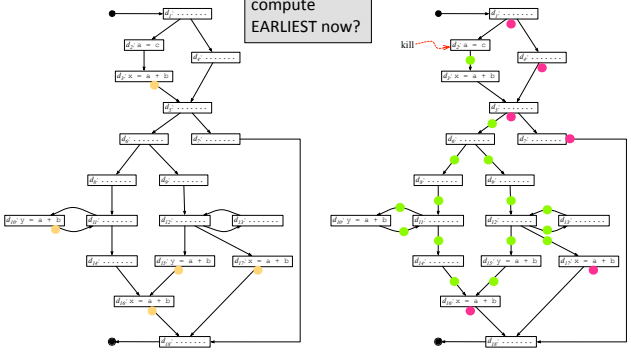
What is KILL(i) ∪ $\overline{OUT_{ANTICIPABLE}(i)}$ in our running example?

This figure shows the IN sets of anticipability analysis.



EARLIEST(i, j) = $IN_{ANTICIPABLE}$(j) ∩ ~~$OUT_{AVAILABLE}$(i)~~ ∩ (KILL(i) ∪ ~~$OUT_{ANTICIPABLE}$(i)~~)

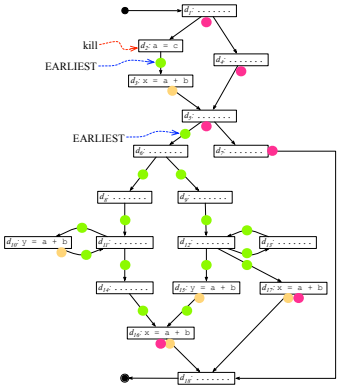
Can you compute EARLIEST now?



EARLIEST(i, j) = $IN_{ANTICIPABLE}$(j) ∩ ~~$OUT_{AVAILABLE}$(i)~~ ∩ (KILL(i) ∪ ~~$OUT_{ANTICIPABLE}$(i)~~)

We have two EARLIEST edges in this CFG.

- Anticipable at IN(j)
- Not anticipable at OUT(i)
- Available at OUT(i)

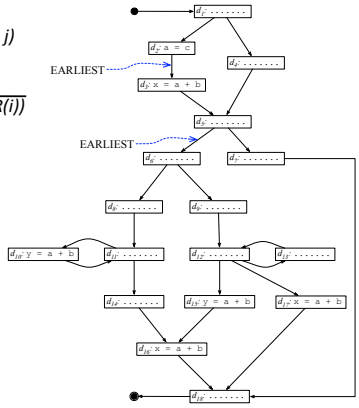


Latest

$IN_{LATER}(j) = \cap_{i \in pred(j)} LATER(i, j)$

$LATER(i, j) = EARLIEST(i, j) \cup (IN_{LATER}(i) \cap \overline{EXPR(i)})$

The goal now is to compute the latest IN sets, and the latest edges. Can you do it?



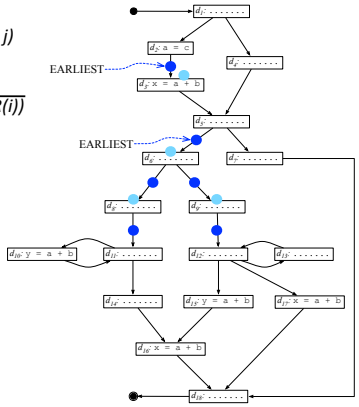
Latest

$IN_{LATER}(j) = \cap_{i \in pred(j)} LATER(i, j)$

$LATER(i, j) = EARLIEST(i, j) \cup (IN_{LATER}(i) \cap \overline{EXPR(i)})$

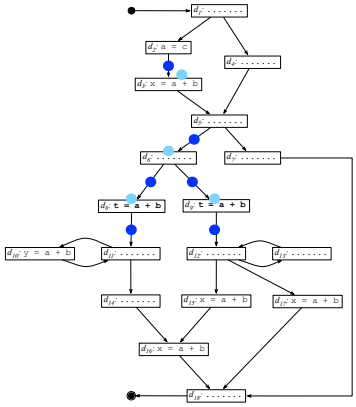
The LATEST edges are marked with the dark blue circles.

The LATEST IN sets are marked with the light blue circles.



$INSERT(i, j) = LATER(i, j) \cap \overline{IN_{LATER}(j)}$

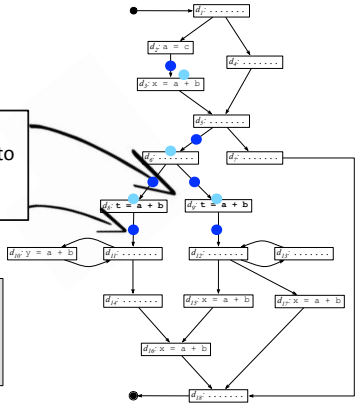
We must now find the sites where we can insert new computations of a + b. Can you compute INSERT(i, j)?



$INSERT(i, j) = LATER(i, j) \cap \overline{IN_{LATER}(j)}$

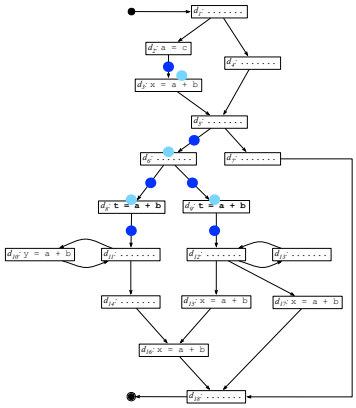
In this example, we only have two sites to insert new computations.

Do you remember why we insert the computation of a + b at d_8, instead of d_{11}?



$$DELETE(i) = EXPR(i) \cap \overline{IN_{LATER}(i)}$$

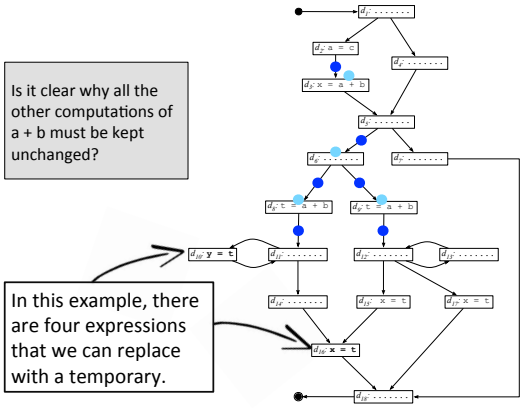
We must now delete redundant computations that exist at program points p. Can you determine DELETE(p)?



$$DELETE(i) = EXPR(i) \cap \overline{IN_{LATER}(i)}$$

Is it clear why all the other computations of a + b must be kept unchanged?

In this example, there are four expressions that we can replace with a temporary.



13 Profile Guided Optimizations

13.1 Efficient Path Profiling

13.2 Improved Basic Block Reordering

Improved Basic Block Reordering [2] is published by Andy Newell and Sergey Pupyrev from Facebook.

Given a directed control flow graph comprising of basic blocks and frequencies of jumps between the blocks, find an ordering of the blocks such that the number of fall-through jumps is maximized. This is the maximum directed TRAVELING SALESMAN PROBLEM (TSP). Solving TSP alone is not sufficient for constructing a good ordering of basic blocks. It is easy to find examples of control flow graphs with multiple different orderings that are all optimal with respect to the TSP objective. Consider for example a control flow graph in Figure 68 in which the maximum number of fall-through branches is achieved with two orderings that utilize a different number of I-cache lines in a typical execution. For these cases, an algorithm needs to take into consideration non-fall-through branches to choose the best ordering. However, maximizing the number of fall-through jumps is not always preferred from the performance point of view. Consider a control flow graph with seven basic blocks in Figure 69. It is not hard to verify that the ordering with the maximum number of fall-through branches is one containing two concatenated chains, B0→B1→B3→B4 and B5→B6→B2 (upper-right in Figure 69). Observe that for this placement, the hot part of the function occupies three 64-byte cache lines. Arguably a better ordering is the lower-right in Figure 69, which uses only two cache lines for the five hot blocks, B0, B1, B2, B3, B4, at the cost of breaking the lightly weighted branch B6→B2.

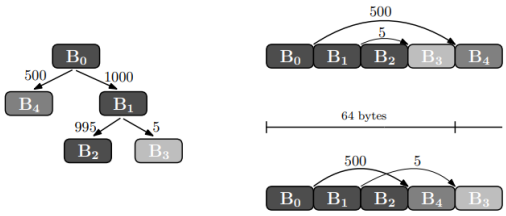


Figure 68: Two orderings of basic blocks with the same TSP score (1995) resulting in different I-cache utilization. All blocks have the same size of 16 bytes and colored according to their hotness in the profile.

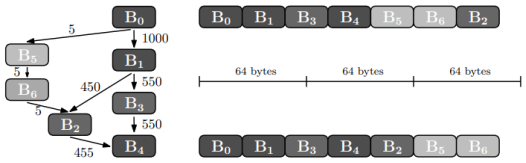


Figure 69: A control flow graph with jump frequencies (left) and two possible orderings of basic blocks (right). All blocks have the same size (in bytes) and colored according to their hotness in the profile. An optimal TSPbased layout (upper right) utilizes three cache lines for the hot code, while an arguably better layout (lower right) can be built with a new EXTTSP model.

13.2.1 Contribution

The contributions of the paper are the following.

- Identify an opportunity for improvement over the classical approach for basic block reordering, initiated by Pettis and Hansen [3]. Then they extend the model and suggest a new optimization problem with the objective closely related to the performance of a binary.
- Develop a new practical algorithm for basic block reordering. The algorithm relies on a greedy technique for solving the optimization problem.
- Propose a Mixed Integer Programming formulation for the aforementioned optimization problem, which is capable of finding optimal solutions on small functions

13.2.2 New ideas

In their study, they consider the following features.

- The length of a jump impacts the performance of instruction caches. Longer jumps are more likely to result in a cache miss than shorter ones. In particular, a jump with the length shorter than 64 bytes has a chance to remain within the same cache line.
- The direction of a branch plays a role for branch predicting. A branch s→t is called forward if s $<$ t, that is, block s precedes block t in the ordering; otherwise, the branch is called backward.
- The branches can be classified into unconditional (if the out-degree is one) and conditional (if the out-degree is two). A special kind of branches is between consecutive blocks in the ordering that are called fall-through; in this case, a jump instruction is not needed.
- They introduce a new score that estimates the quality of a basic block ordering taking into account the branch characteristics. In the most generic form, the new function, called EXTENDED TSP (EXTTSP), is expressed as follows:

$$\text{ExtTSP} = \sum_{(s,t)} w(s,t) \times K_{s,t} \times h_{s,t}(\text{len}(s,t))$$

where the sum is taken over all branches in the control flow graph. Here $w(s,t)$ is the frequency of branch s→t and $0 \leq K_{s,t} \leq 1$ is a weight coefficient modeling the relative importance of the branch for optimization. We distinguish six types of branches arising in code: conditional and unconditional versions of fall-through, forward, and backward branches. Thus, we introduce six coefficients for EXTTSP. The lengths of the jumps are accounted in the last term of the expression, which increases the importance of short jumps. A non-negative function $h_{s,t}(len(s,t))$ is defined by value of 1 for zero-length jumps, value of 0 for jumps exceeding a prescribed length, and it monotonically decreases between the two values. To be consistent with the objective of TSP, the EXTTSP score needs to be maximized for the best performance. Notice that EXTTSP is a generalization of TSP, as the latter can be modeled by setting $K_{s,t} = 1, h(len(s,t)) = 1$ for fall-through branches and $K_{s,t} = 0$ otherwise.

They use machine learning methods to find parameters for EXTTSP that have the highest correlation with the performance of a binary in the experiment.

$$\text{ExtTSP} = \sum_{(s,t)} w(s,t) \times \begin{cases} 1 & \text{if } \text{len}(s,t) = 0, \\ 0.1 \cdot \left(1 - \frac{\text{len}(s,t)}{1024}\right) & \text{if } 0 < \text{len}(s,t) \leq 1024 \\ & \text{and } s < t, \\ 0.1 \cdot \left(1 - \frac{\text{len}(s,t)}{640}\right) & \text{if } 0 < \text{len}(s,t) \leq 640 \\ & \text{and } t < s, \\ 0 & \text{otherwise.} \end{cases}$$

. Intuitively, EXTTSP resembles the traditional TSP model, as the number of fall-through branches is the dominant factor. The main difference is that EXTTSP rewards longer jumps. The impact of such jumps is significantly lower and it linearly decreases with the length of a jump. Next we summarize our high-level observations regarding the new score function.

13.2.3 Algorithm

Algorithm 14 Basic Block Reordering

Input: control flow graph $G = (V, E, w)$, the entry point $v^\star \in V$
Output: ordering of basic blocks $(v^\star = B_1, B_2, \ldots, B_{|v|})$
function ReorderBasicBlocks
  for $v \in V$ **do**
    $Chains \leftarrow Chains \cup (v)$
  end for
  while $|Chains| > 1$ **do** ▷ chain merging
    for $c_i, c_j \in Chains$ **do**
      $gain[c_i, c_j] \leftarrow$ ComputeMergeGain$(c_i, c_j\ )$
    end for
    src, $dst \leftarrow \arg\max_{i,j} \text{gain}\,[c_i, c_j]$ ▷ find best pair of chains
    $Chains \leftarrow Chains \cup Merge(src, dst) \backslash \{src, dst\}$; ▷ merge the pair and update chains
  end while
  return ordering given by the remaining chain;
end function
function ComputeMergeGain(src, dst)
  for $i = 1$ **to** blocks(src) **do** ▷ try all ways to split chain src
    $s_1 \leftarrow src[1 : i]$ ▷ break the chain at index i
    $s_2 \leftarrow src[i + 1 : blocks(src)]$
    score $_i \leftarrow \max \begin{cases} \text{ExtTSP}\,(s_1, s_2, dst) & \text{if } v^* \notin dst \\ \text{ExtTSP}\,(s_1, dst, s_2) & \text{if } v^* \notin dst \\ \text{ExtTSP}\,(s_2, s_1, dst) & \text{if } v^* \notin s_1, dst \\ \text{ExtTSP}\,(s_2, dst, s_1) & \text{if } v^* \notin s_1, dst \\ \text{ExtTSP}\,(dst, s_1, s_2) & \text{if } v^* \notin src \\ \text{ExtTSP}\,(dst, s_2, s_1) & \text{if } v^* \notin src \end{cases}$ ▷ try all valid ways to concatenate
  end for
  return $\max_i$ score $_i$ − ExtTSP(src) − ExtTSP(dst) ▷ the gain of merging chains src and dst
end function

References

[1] Etienne Morel and Claude Renvoise. "Global optimization by suppression of partial redundancies". In: *Communications of the ACM* 22.2 (1979), pp. 96–103.

[2] Andy Newell and Sergey Pupyrev. "Improved basic block reordering". In: *IEEE Transactions on Computers* 69.12 (2020), pp. 1784–1794.

[3] Karl Pettis and Robert C Hansen. "Profile guided code positioning". In: *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*. 1990, pp. 16–27.