Compiler Optimization Notes

May 30, 2022

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:

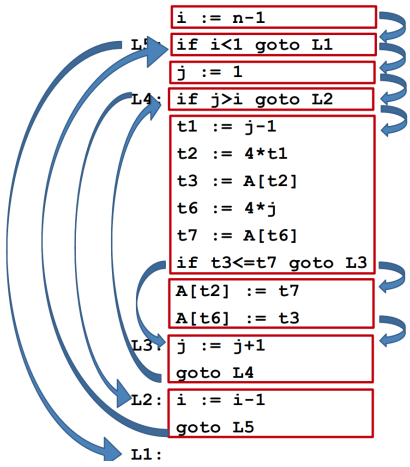


Figure 1: Example of a flow graph

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:

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

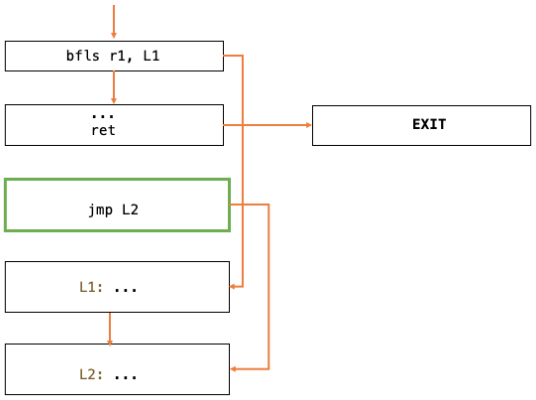


Figure 2: Example of a flow graph

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).

```
1 a = b + c;
2 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.

```
1     a = b + c;
2     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.

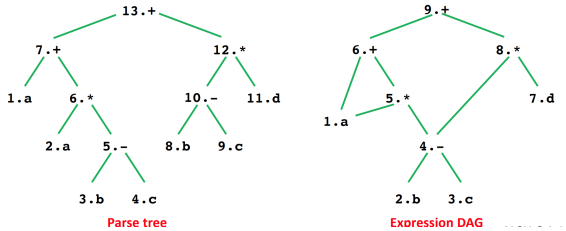


Figure 3: Example of a DAG

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

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

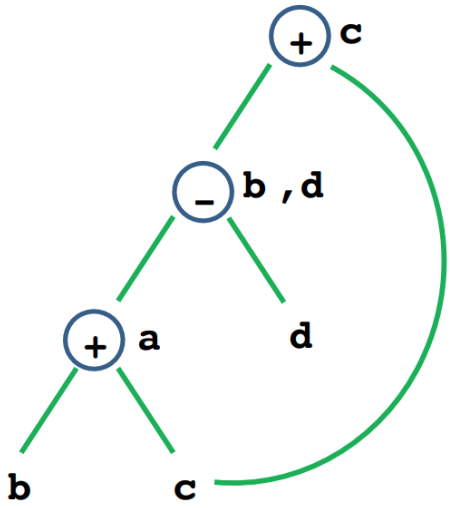


Figure 4: Example of a DAG

```
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.

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

1.4.2 Example

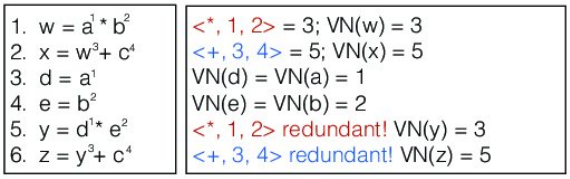


Figure 5: An example of value numbering.

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 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

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

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

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

```
5     else x = b + 4;
6     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 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 $f_n : D \to D$ (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]

[4] based on Wikipedia

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 **semantically** 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

```
1    int t = x * y;
2    if ((x+1)*(x+1) == y) {
3     t = 1;
4    }
5    if (x*x + 2*x + 1 != y) {
6     t = 2;
7    }
8    return t;
```

Listing 9: An

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.

[5] based on slides from Cambridge University

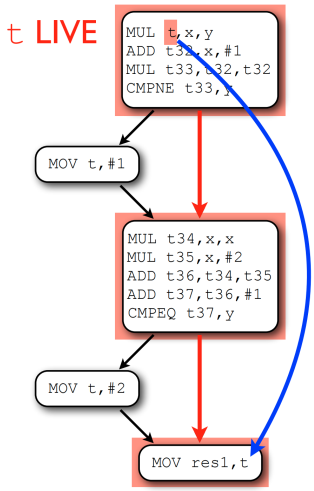


Figure 6: CFG

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

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] = ∪(out[p]), for all predecessors p of B
    oldout = out[i]
    out[i] = f_i(in[i])                                 ▷ out[i] = gen[i] ∪ (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
```

4.3 Example

Here comes an example of reaching definition.

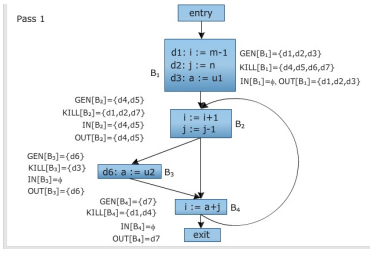


Figure 7: Pass 1

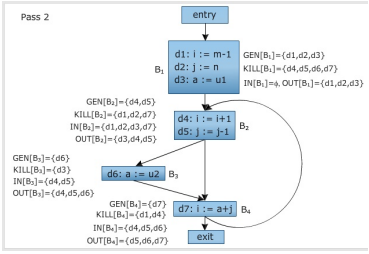


Figure 8: Pass 2

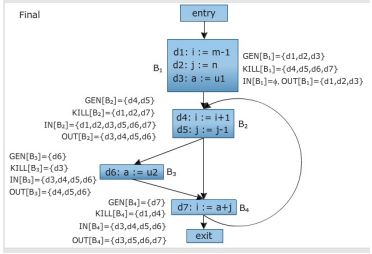


Figure 9: Pass 3

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

```
1    int z = x * y;
2    print s + t;
3    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.

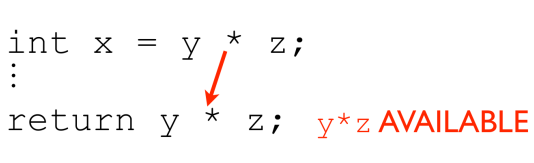


Figure 10: Available expression example

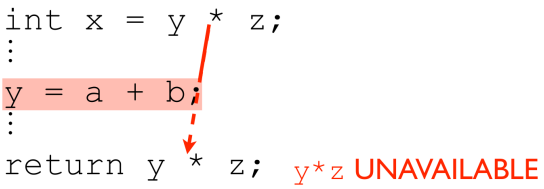


Figure 11: unavailable expression example

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.

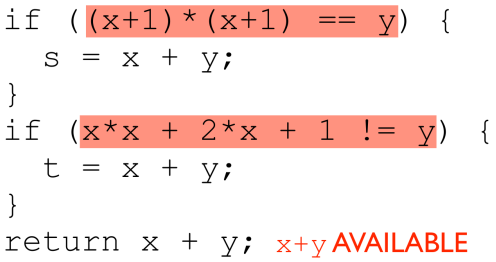


Figure 12: x+y is semantically available

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.

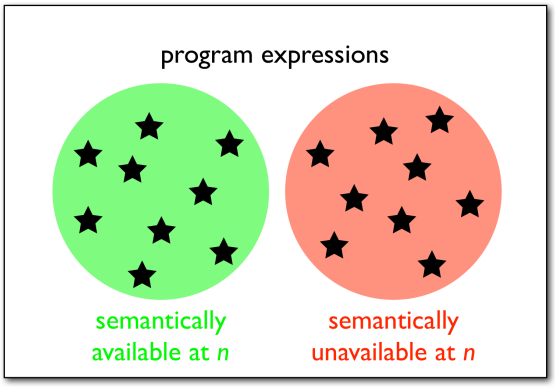


Figure 14: Semantic vs. syntactic

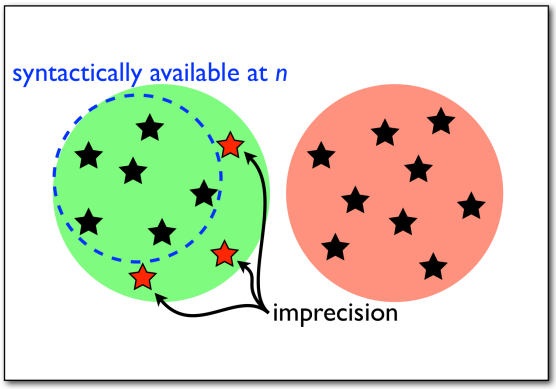


Figure 15: Semantic vs. syntactic

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?
- Spped: 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)

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

- 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 prectical 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 $x = z$
- **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).

6.4 Complete Lattice

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

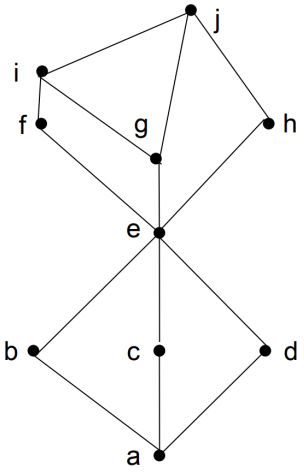
6.5 Semi-Lattice

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

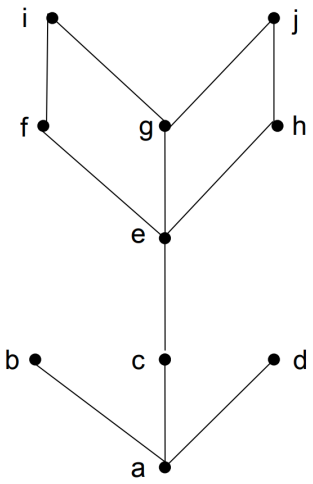
6.6 Meet Operator

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.



(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

Figure 17: Meet Operator

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 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** $x = x$

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

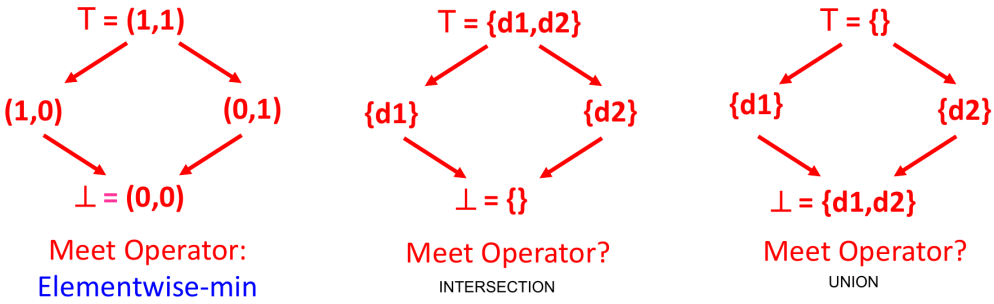


Figure 18: Different meet operator defines different lattice

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.

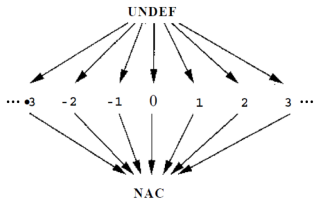


Figure 19: The lattice of constant propogation

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 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.

6.11 spped of

7 LLVM

7.0.1 PGO and LLVM

Profile Guided Optimizations.

7.1 Profile Guided Transforms

7.1.1 Spill Placement

registers

7.1.2 Code Layout

Code layout [1] is the process of ordering the blocks of the CFG. This order dictates the placement of instructions within those blocks in memory. By inserting branch instructions at the end of the basic blocks, the compiler can layout the blocks in any order. Consider the Control Flow Graph (CFG) in Figure 20. Figures 21 and 22 show two possible layouts of the CFG. In both

theese layouts, a block is followed by one of its successor blocks that is not yet laid out. Consider block A, for example. It has two successors in the CFG. Only one of the successors B or C can be placed immediately following A and is known as the **fall-through block**. In Figure 21, B is the fall-through block and in the layout in Figure 22, C is the fall-through block. The choice of which block to place as the fall-through block has performance implications. If control is often transferred to C from A often during program execution, then placing C next to A has the following advantages:

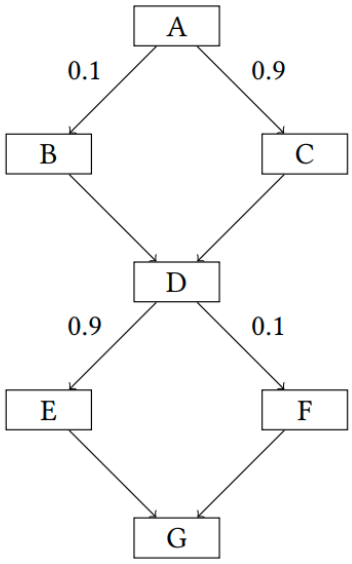


Figure 20: CFG

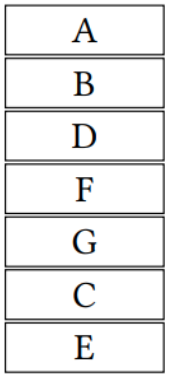


Figure 21: Layout 1

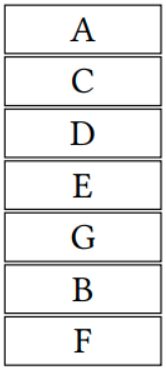


Figure 22: Layout 2

Figure 23: Code Layout

- Since the branch at the end of A is mostly not-taken, the frontend of the processor's pipeline is less likely to be stalled if it is an out-of-order superscalar processor.
- As the cacheline containing the last instruction of A also contains instructions that are more likely to execute (from block C), instruction cache utilization is likely to be better.

In the LLVM compiler, the `MachineBlockPlacement` pass performs code layout optimization. This pass relies on the branch probability analysis which provides, for each branch instruction, the probability of the branch being taken. `MachineBlockPlacement` is just one of the many transformation passes that make use of branch probability analysis. Branch probability analysis is used in another analysis called block frequency analysis that provides relative frequencies of basic blocks within a function. Block frequency analysis is used by optimizations such as inlining, spill-code placement in register allocation among others.

7.1.3 Hot/Cold Partitioning

Hot Cold splitting is an optimization to improve instruction locality. It is used to outline basic blocks which execute less frequently. The hot/cold splitting pass identifies cold basic blocks and

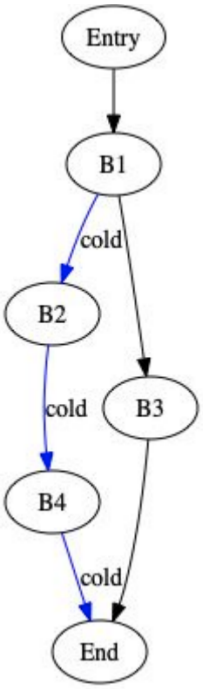


Figure 24: CFG

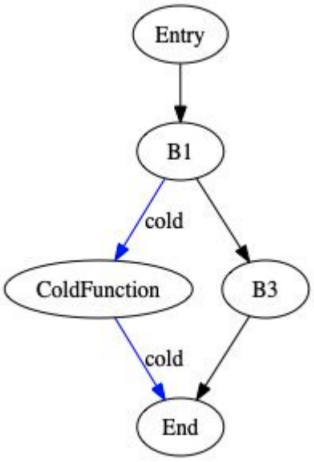


Figure 25: Layout 1

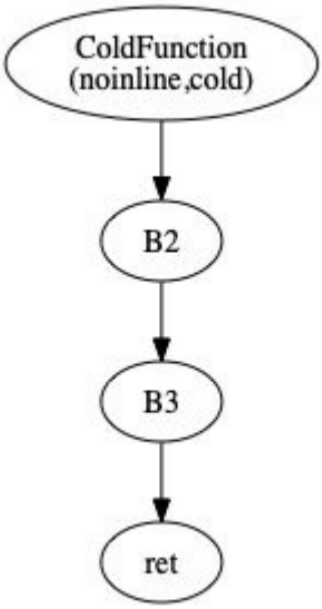


Figure 26: Layout 2

Figure 27: Code Layout

moves them into separate functions. The linker can then put newly-created cold functions away from the rest of the program . The idea here is to have these cold pages faulted in relatively infrequently, and to improve the memory locality of code outside of the cold area.

The algorithm is novel in the sense it is based on region and implemented at the IR level. Because it is implemented at the IR level, all the backend targets benefit from this implementation. Other implementations of hot-cold splitting outline each basic block separately and are implemented at the RTL level.

What applications will benefit from Hot/Cold Spliting?

- High cache misses(A giant app on a small device)
- High start-up time

7.1.4 Inliner

7.1.5 Outlining & Merging

With PGO information, we can do more aggressive outlining of cold regions in the inline candidate function. This contrasts with the scheme of keeping only the 'early return' portion of the inline candidate and outlining the rest of the function as a single function call.

Support for outlining multiple regions of each function is added, as well as some basic heuristics to determine which regions are good to outline. Outline candidates limited to regions that are single-entry & single-exit. Also we don't account for live-ranges we may be killing across the region with a function. These are enhancements we can consider in another patch.

Fallback to the regular partial inlining scheme is retained when either i) no regions are identified for outlining in the function, or ii) the outlined function could not be inlined in any of its callers.

7.1.6 Control height reduction

Control height reduction merges conditional blocks of code and reduces the number of conditional branches in the hot path based on profiles.

```
if (hot_cond1) { // Likely true.

do_stg_hot1();
}
if (hot_cond2) { // Likely true.

do_stg_hot2();
}
```

Listing 11: An

```
if (hot_cond1 && hot_cond2) { // Hot path.

do_stg_hot1();
do_stg_hot2();
} else { // Cold path.

if (hot_cond1) {
  do_stg_hot1();
}
if (hot_cond2) {
  do_stg_hot2();
}
}
```

Listing 12: An

This speeds up some internal benchmarks up to 30%.

7.2 Loop Unrolling & Loop Vectorization

```
for (int i=0; i<16; ++i)
C[i] = A[i] + B[i];
```

Listing 13: An

```
for (int i=0; i<16; i+=4) {
  C[i] = A[i] + B[i];
  C[i+1] = A[i+1] + B[i+1];
  C[i+2] = A[i+2] + B[i+2];
  C[i+3] = A[i+3] + B[i+3];
  }
```

Listing 14: An

```
  for (int i=0; i<16; i+=4)
  addFourThingsAtOnceAndStoreResult(
  &C[i], &A[i], &B[i]);
```

Listing 15: An

7.3 Partial Inline

```
  for (int i=0; i<16; i+=4)
  addFourThingsAtOnceAndStoreResult(
  &C[i], &A[i], &B[i]);
```

Listing 16: An

7.4 Partial Inline

```
  void foo() {
    bar();
    // rest of the code in foo
    }
    void bar() {
    if (X)
    return;
    // rest of code (to be outlined)
    }
```

Listing 17: An

```
  void foo() {
    if (!X)
    bar.outlined();
    // rest of the code in foo
    }
    void bar.outlined() {
    // rest of the code in bar
    }
```

Listing 18: An

References

[1] Easwaran Raman and Xinliang David Li. "Learning Branch Probabilities in Compiler from Datacenter Workloads". In: *arXiv preprint arXiv:2202.06728* (2022).