Compiler Optimization Notes

May 23, 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:

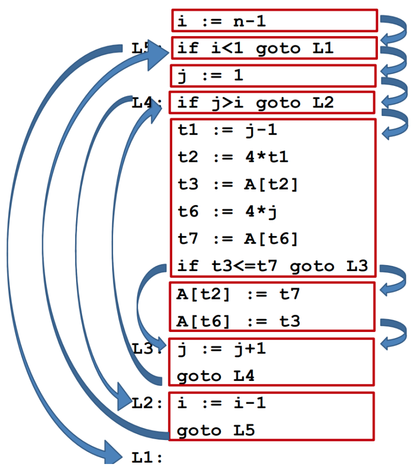

Figure 1: Example of a flow graph

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

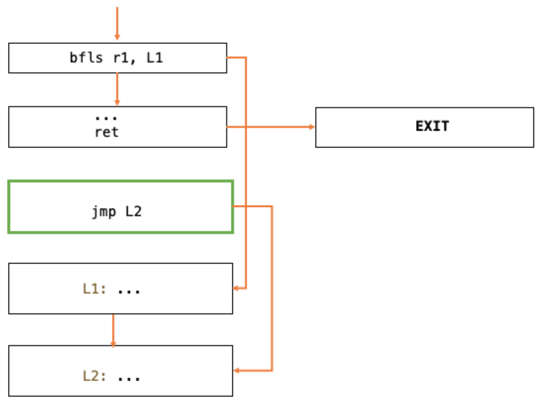

Figure 2: Example of a flow graph

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).

```
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.

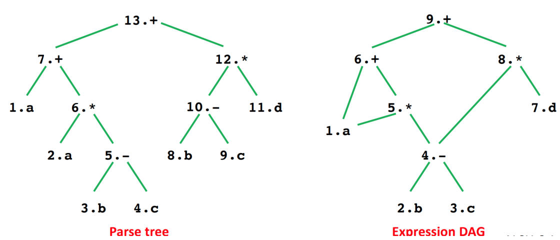

Figure 3: Example of a DAG

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

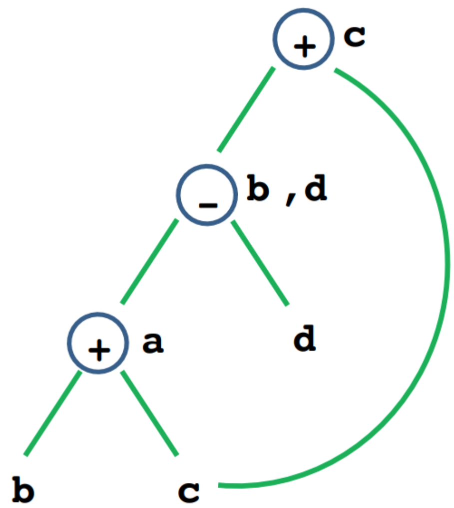

Figure 4: Example of a DAG

```
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

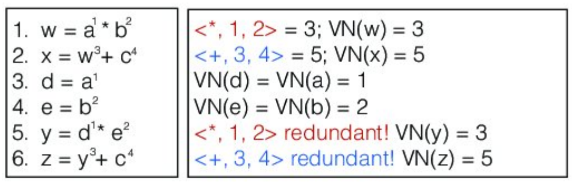

Figure 5: An example of value numbering.

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 infinite, 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 id 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.
- Backward problems (like live-variable analysis), where the information at a node n summarizes what can happen on paths from n to "exit".

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.

2.2 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.

2.2.1 Example

2.3