Constant Propagation with Conditional Branches

Brendan Briody
Hohenhausgasse 14
78462 Konstanz
E-Mail: mail@brendan.de
Matrikel-Nr.: 01/461643
1. Introduction

This paper deals with detecting constant occurrences in programme code and propagating these down through its control flow. At first a distinction between two types of concrete occurrences has to be made: Simple - and Conditional Constants. Both of these occurrences are at first handled in separate algorithms named after their types and with Simple Constant another algorithm (Sparse Simple Constant) using a different representation called Static Single Assignment (SSA) is introduced.

All of the three algorithms use the same mathematical technique by applying lattices. Together with lattices the Static Single Assignment representation are the focal point of this paper. The algorithms are represented in order of power by the amount of constant occurrence types detected and by simplicity.

2. Why is Constant Propagation useful?

Detecting constant occurrences at compile time can make run time evaluations obsolete. This can save a great amount of runtime overhead producing smaller programmes and gaining a considerable speedup. Constant values need not be allocated like variables in memory but just inserted into expressions as constant operands. Also code, which is never reached, can be deleted by discovering conditional branches and identifying the only possible branch to be taken. A form of dead code called “unreachable code” is established.
3. Preliminaries

In order to propagate constants through a programme, a form of programme flow is necessary. For this reason the programme is broken up into different nodes and built up into a form of a control flow graph (CFG). The three types of nodes are assignment –, conditional - and a unique start node. Conditional nodes contain an expression, which is evaluated, and the control is transferred to a following node. Assignment nodes are parts of the programme where variable definitions occur in terms of other variables and constants. Not only can nodes have several outgoing edges as known in conditional nodes but also several incoming edges called joins [Fig. 1].

To describe the state of variables, lattices are assigned to each variable and can be changed throughout the programme flow.

The output of a Constant Propagation algorithm are lattice values of each variable. These lattices are associated with results and operands of expressions and every node contains latticeCells of all the variables in the programme.
The cells can hold one of 3 values distinguished by a unique level:

<table>
<thead>
<tr>
<th>Level</th>
<th>Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Highest level: top</td>
<td>variable possibly constant</td>
</tr>
<tr>
<td>Middle level</td>
<td>lattice refers to a constant i</td>
</tr>
<tr>
<td>Lowest level: bottom</td>
<td>constant value not guaranteed</td>
</tr>
</tbody>
</table>

At the end of a propagation algorithm all latticeCells contain either the middle—, or the bottom level. The algorithms take an optimistic approach assigning all variables to the top level at the start. This also depends on the source code language. In some languages undefined variables are allowed but could yield incorrect results if assigned to top level at start. In FORTRAN for example all latticeCells are set to the bottom level at the beginning.

The algorithms proceed by lowering the latticeCells at each node as information is discovered. To gain this information meet ($\prod$) rules are performed between the latticeCells of a distinct variable from one point of a node to another, or from one node to another.

![Rules for meet ($\prod$) rules](Fig. 3)

Expression evaluation rules mean that if an expression contains at least one variable with a latticeCell assigned to the bottom level, then the whole expression is of bottom level and is therefore not constant. Vice versa, all operands could be of middle level, so the expression would also evaluate to a constant.

Special evaluation rules given for Boolean expressions are:

- any OR true = true
- any AND false = false
Fixed points are places in the programme code where latticeCells of variables in question must be determined as constants (middle level) or non-constants (bottom level). An example for a fixed point is a write(()) expression shown in [Fig. 1].

To proceed through a programme flow, all algorithms also make use of a work list (WL) for propagation in which objects such as nodes are stored, evaluated, and then removed.

4. Simple Constant (SC)

The Simple Constant algorithm uses a CFG and a WL to propagate constants. Two lattice cells for each variable in each node are set: one at the entry and one at the exit with an expression containing variables or constant operands in between. Meets are performed between latticeCells at the entry and exit. A meet is then performed from the exit LatticeCell and the following entry LatticeCell of the following node [Fig. 4]. This way Lattice values are propagated down through nodes. At each node every LatticeCell of every variable in the programme is “updated” through meet operations.

The WL containing the unique start node at the beginning adds following nodes and removes preceding nodes that have been visited.

If a value of a variable changes then the following node(s) are added to the WL.

\[
\begin{align*}
\text{Assignment node} & \quad [1] \\
> & \ L(a) \\
\text{a := 8} \\
< & \ L(a)
\end{align*}
\]

\[
\begin{align*}
\text{Assignment node} & \quad [k] \\
> & \ L(a) = >\ L(a) \\
\text{b := a \times 5} \\
< & \ L(a)
\end{align*}
\]

[Fig. 4]
At a fixed point SC must decide which Lattice value a variable is to hold. In the case when a fixed point has two incoming edges [Fig. 1] the values of the variable in question coming from each edge must be compared. If values are different then a constant value cannot be guaranteed at this fixed point, no matter what lattice value each variable has at this point.

SC therefore holds a single value and Lattice of every variable in each conditional branch of a programme but does not consider which branch is to be taken.

4. Conditional Constant (CC)

This algorithm not only detects simple constants but can also find conditional constants and ignores the edge that is never taken in the CFG. The CC algorithm postpones the evaluation of a CFG edge until it is marked as executable. At start off CC marks all edges not executable and the start node is added to the WL. When an assignment node is executed the out edge is simply marked executable and added to the WL. When CC executes a conditional node the conditional expression (CE) is evaluated. If the CE evaluates to the middle lattice value then only the one branch can be taken and is added to the WL, otherwise the CE is considered to be of bottom lattice level and all edges from the conditional node are marked executable and added to the WL. A conditional expression can also be regarded as a fixed point.

CC accomplishes a form of dead code elimination called unreachable code elimination.

Information can as well be gained in conditional branches. Consider for instance:

```plaintext
if (i=1)
    then a:= i + 4
    else a:= i – 4
```

One can easily see that if the true branch is taken and a use of i occurs, an assumption can be made that i certainly holds the value 1. By inserting an assignment node between the conditional node “if (i=1)” and the true branch with i:= 1 a “raising” of the lattice value of i back to the top level can be performed if necessary.
5. Sparse Simple Constant (SSC)

SSC finds all simple constants by making use of Static Single Assignment (SSA). This chapter is supposed to give an idea of the simplicity using SSA in comparison to the two former algorithms but only detecting simple constants. At first SSC must gain information of the use-def-edges of the programme code. This knowledge is extracted from the SSA form [Fig. 5] marked with arrows.

SSC performs several steps to propagate constants:

1. **Examine assignment expressions**
   - If exp. contains no variables the middle lattice level is assigned. If the exp. cannot be established at compile time, ex. by a read() statement then the lattice value is of the bottom level. The top level is set to all other assignment expressions.

2. **Initialise WL**
   - A WL is initialised containing references to all exps. not holding the top lattice value, in this case :[1],[2],[4],[6] of [Fig. 5].

3. **Perform meets (Π)**
   - Take SSA edge of WL and perform meets between use- and def edge.
4. **Evaluate meets**

If the meet (rules in [fig. 3]) between **use** and **def** is different from the lattice value at the use end, replace this use lattice value with the performed meet.

If the new use lattice value is *lower* than the stored reference in the WL then all following SSA edges are added to the WL.

When coming up to a **def** edge in the WL with a lattice discovered as **bottom** level, replace the corresponding use edge with the constant value, ex. (def) $a_1 := 4$ → (use) $x_1 := a_1 + 2$ would evaluate to $x_1 := 4 + 2$.

A Similar handling of SSA $\Phi$ functions is also done by replacing the operand variables with the corresponding expressions: $x_2 := a_2 + 2$ ; $x_1 := 4 + 2 \rightarrow \Phi(x_1, x_2)$ would result to $\Phi(6, a_2 + 2)$.

After an element has been evaluated it is removed from the WL. The algorithm terminates when the WL is empty.

To make the SSC algorithm clearer, a simple pseudo code is shown below:

```plaintext
Meet_val, WL_element, t=0
while WL not empty
   do {
      WL_element = WL[t] // get element from the worklist
      if (WL_element.Use = NULL) Goto :Continue // possible redundancy
      While(edges of WL_element)
          do{
              Meet_val = Meet(Lattice(WL_element.Use), Lattice(WL_element.Def))
              if (Lattice(WL_element.Use) ≠ Meet_val)
                  Lattice(WL_element.Use) := Meet_val
              if (Lattice(WL_element.Use) lower than Lattice(WL[t].Use))
                  WL_add( WL[t].Use.NextEdge)
                  Lattice(WL[t].Use) := Meet_val
          }
      if ( Lattice(WL_element.Def) = middle)
          insert constant value into WL[t].Use
      if ( Lattice(WL_element.Def) = $\Phi$ function )
          evaluate def exps. and insert into WL[t].Def
          : Continue
      Remove WL[t] from WL
      t++
   }
```
6. Conclusions

Compiler optimisation should work as a seamless process, gathering information and presenting it in a compatible form for following optimisations. The first two algorithms are meant to show the differences between constant occurrences and basically how a Worklist and lattices are used. SC and CC propagate every variable in a node with a considerable overhead. SSC gains knowledge from SSA simply in form of use-def-edges and propagates variables merging defs into uses from the WL and thus producing more information towards the SSA representation. An obvious assumption is that Constant Propagation exposes redundancies to be handled in further optimisations.