Intraprocedural Data Flow Analysis

Christian Bisig, HSR, MRU Software and Systems, E-mail: cbisig@hsr.ch
Advisers: Prof. Peter Sommerlad, Thomas Corbat

Friday 19th December, 2014

Abstract

This article analyses five practical examples with different techniques of intraprocedural data flow analysis. These static program analysing methods are part of an optimising step within compilers. The examples are based on java code snippets and show, how to find optimisation possibilities for the following shortcomings: unnecessary re-computation of an arithmetic expression, using uninitialised variables as well as redundant, dead and loop-invariant code. The intraprocedural data flow analysis methods to conquer such unwanted states are called Available Expression Analysis, Reaching Definitions Analysis, Very Busy Expression Analysis (a.k.a. Anticipatable Expressions Analysis), Live Variables Analysis and Derived Data Flow Information. To apply these methods there is either an equational or a constraint based approach. Within this article the focus lies on the equational approach which is based on two classes of data flow equations. The solutions to these equations are visualising the possible code modifications to optimise a certain part of program code. Besides the intraprocedural data flow analysis there is as well the interprocedural analysis. This article concentrates only on the intraprocedural approach.

1 Introduction

The main application of program optimisations using intraprocedural data flow analysis is inside of an optimising compiler and is called global optimisations [1] [2]. The intraprocedural methods are used for static program analysis. Normally it is not directly the code hand written by a programmer which is optimised, but an intermediate code which is generated by a pre-compiling step. The intermediate code is the representation in an language for an abstract machine (intermediate representation) i.e. it is still independent from the physical hardware on which it is intended to run. After the global optimisations the compiler translates the intermediate code into machine code. In this step different machine code optimisations can take place, which will not be further discussed in this article. Even though the intraprocedural methods are applied mostly on the intermediate code, the examples in this article are based on plain java code for the sake of a simplified illus-
The reason why intraprocedural data flow analysis and its subsequent optimisations might be necessary, are the following: During the implementation of program code, a programmer might write, move, delete or insert code parts at different points in the program. Due to this permanent editing of the code over a period of time, the chances of creating deficiencies is very high. The deficiencies created might be for example, . . .

- unnecessary re-computation of arithmetic expressions
- uninitialised variables
- code redundancy
- unused variables
- dead code

These deficiencies can result in different problems e.g. lack of performance, unnecessary allocation of working storage, unnecessary complexity of code or unnecessary code at all. But there could as well be the problem, depending on programming language and compiler, that the code is not even compilable (uninitialised variables).

If the programmer uses a plain text editor, these deficits will most likely remain undiscovered by his human eye. Except of the possible compiler errors, which must have been fixed before the program can be executed at all.

For this reason, the programmer is glad for every assistance he can get to remedy such deficiencies before he ever tries to compile his program. Intraprocedural data flow analysis contains a set of methods to find such unseen problems.

Compilers know different intraprocedural methods for code optimisations e.g. to avoid unnecessary re-computations, to remove unused (dead) code or to reposition loop invariant code. The program is then optimised for higher performance and lesser space (memory) usage, even though the source code is not optimal.

The four available data flow analysis methods can be applied with a system of equations. It is called *Equational Approach*. In this approach there are two classes of equations:

1. The first one relates output information of a label with the input information of the same label. Or vice versa (depending on the analysis method). E.g. Building the output info is done by subtracting the expressions respectively variables which are recalculated/reassigned in the observed label from the expressions/variables which are available at the input of the same label and then adding the expressions/variables which are newly generated in the same label.

   \[
   \text{out}(\ell) = (\text{in}(\ell) \setminus \text{kill}(\ell)) \cup \text{gen}(\ell)
   \]

2. The second class relates either the input to all the predecessors output (then it is called a forward analysis) or the output to all the successors input (then it is called a backward analysis). If there is more than one predecessor respectively successor, the elements are either united (\(\cup\)) or intersected (\(\cap\)), depending on the chosen analysis method. E.g.

   \[
   \text{in}(\ell) = \bigcap_{p \in \text{pred}(\ell)} \text{out}(p)
   \]
The table shows the four data flow analysis methods and how the class 2 equation is composed.

<table>
<thead>
<tr>
<th></th>
<th>union $\cup$</th>
<th>intersec. $\cap$</th>
</tr>
</thead>
<tbody>
<tr>
<td>forward</td>
<td>Reaching</td>
<td>Available</td>
</tr>
<tr>
<td></td>
<td>Definitions</td>
<td>Expressions</td>
</tr>
<tr>
<td></td>
<td>Analysis</td>
<td>Analysis</td>
</tr>
<tr>
<td>backward</td>
<td>Live</td>
<td>Very Busy</td>
</tr>
<tr>
<td></td>
<td>Variables</td>
<td>Expressions</td>
</tr>
<tr>
<td></td>
<td>Analysis</td>
<td>Analysis</td>
</tr>
</tbody>
</table>

Table 1: Class 2 Data Flow equation types

1.1 Notation

In the following two subsections some details about the notation used in the Java code listings and the data flow diagrams are explained.

1.1.1 Java program code listings

In the Java program code listings, each code line represents a label and they are referred as labels in the text. For example in the text description it will be written as "label 5 in listing 3". Which means code line 5 in the code listing which has a starting caption like "Listing 3: . . . ".

1.1.2 Data Flow Diagram

The data flow diagrams show each label as an own node. The code lines which are declaring a variable without initialisation, are not shown. The numbering of the nodes matches the code lines in the according program code listing.

2 Unnecessary re-computation

If in a program the same (arithmetical) expression appears multiple times at different points of the program, it might not be necessary to re-compute the expression every time. If the variables, used in the expression, do not change in between two appearances of the same expression, it is not necessary to waste processing time to recalculate the expression, since the result of the calculation will be the same. This section explains an algorithm to compute if an expression needs to be computed once.

2.1 Example of unnecessary re-computation

The example code in listing shows, that the expression $a + b$ appears multiple times. The question to ask is: Does the expression $a + b$ have to be re-computed every time at its appearance? The answer is no, but only if the variables ($a$ and $b$) used in the expression are not changed in between of the different appearances of the expression.

```
1 public void compute(int a, int b) {
2     int x = a + b;
3     int y = a * b;
4     while(y > a + b) {
5         a = a + 1;
6         x = a + b;
7     }
8 }
```

Listing 1: Java example, unnecessary re-computation
2.2 Analysis with Available Expressions Method

There is a forward flow analysis method which is called Available Expressions Analysis. The analysis shows for each program point, which expressions must already have been computed before. With this information it can be evaluated if an expression used at the current program point has to be recalculated or not.

In the example code (listing 1), the arithmetic expression \( a + b \) appears in multiple labels (2, 4, 6), but its variables \( a \) and \( b \) might have been changed in between their appearance (according to the program flow figure 1).

2.3 Data Flow

In the data flow diagram (figure 1) it is visible, that the label 4, which contains the while evaluation, has two entry points, the label 6 which is the last label within the while-loop and the label 3 which is right before the loop-construct. This has to be considered during the analysis, since there can be different sets of available input information, depending on the program flow at runtime. Either label 3 or 6 can be the predecessor of label 4.

Based on the data flow of the program, an analysis of the used expressions has to be done at each label. Therefore two terms for the expressions of the code example in listing 1 have to be defined.

1. An expression is called killed, at the current program point (label), if one of its variables is changed at the current label.

```plaintext
1. start
2. x = a + b
3. y = a * b
4. y > a + b
5. a = a + 1
6. x = a + b

Figure 1: procedure flow, Unnecessary recomputation for listing 1
```
2. The expressions calculated at the current program point (label) are called generated for the current label.

\[
\ell \begin{array}{c|c|c}
\text{killed} & \text{generated} \\
2 & \emptyset & \{a+b\} \\
3 & \emptyset & \{a\ast b\} \\
4 & \emptyset & \{a+b\} \\
5 & \{a+b, a \ast b, a+1\} & \emptyset \\
6 & \emptyset & \{a+b\} \\
\end{array}
\]

Table 2: Killed and generated expressions with respect to a and b.

The table shows for each label in listing which expressions are killed or generated. For label 5 it is visible, that the variable a is reassigned, which has the effect, that every expression before and after the current label which is using variable a is being killed. That means they will have to be recomputed at their next usage.

2.4 Data Flow Equations

The following two equations are now used to find which expressions are available at which point of the program. They are depending on the information gathered in the killed-generated-analysis (table 2).

\[
in(\ell) = \begin{cases} 
\emptyset & \text{for the first } \ell \\
\bigcap_{p \in \text{pred}(\ell)} \text{out}(p) & \text{otherwise}
\end{cases}
\] (2.1)

\[
\text{out}(\ell) = (\text{in}(\ell) \setminus \text{killed}(\ell)) \cup \text{gen}(\ell)
\] (2.2)

Since available expression analysis is a forward flow analysis method, the inequality for the first label is an empty set.

If the equations for each label are written down, the results are the following:

In equations:
\[
in(2) = \emptyset 
\] (2.3)
\[
in(3) = \text{out}(2) 
\] (2.4)
\[
in(4) = \text{out}(3) \cap \text{out}(6) 
\] (2.5)
\[
in(5) = \text{out}(4) 
\] (2.6)
\[
in(6) = \text{out}(5) 
\] (2.7)

Out equations:
\[
\text{out}(2) = \text{in}(2) \cup \{a + b\} 
\] (2.8)
\[
\text{out}(3) = \text{in}(3) \cup \{a \ast b\} 
\] (2.9)
\[
\text{out}(4) = \text{in}(4) \cup \{a + b\} 
\] (2.10)
\[
\text{out}(5) = \text{in}(5) \setminus \{a + b, a \ast b, a + 1\} 
\] (2.11)
\[
\text{out}(6) = \text{in}(6) \cup \{a + b\} 
\] (2.12)

2.5 Solving the problem

The equation can be solved starting from the top (equation in(2)) due to the fact, that available expressions is of the type forward flow analysis. The solution of the equations are displayed in the table.

\[
\ell \begin{array}{c|c|c}
\text{In} & \text{Out} \\
2 & \{a+b\} \\
3 & \{a+b\} \cup \{a+b, a*1\} \\
4 & \{a+b\} \cup \{a+b\} \\
5 & \{a+b\} \cup \emptyset \\
6 & \emptyset \cup \{a+b\} \\
\end{array}
\]

Table 3: Available Expressions

The table shows for each label in listing which expressions are available at the beginning (in) and at the end (out) of a label. For example when label 3 is entered,
only the expression $a + b$ has been computed before, which means it is available at that point. In the label 3 itself, the expression $a \times b$ is computed which means at its exit, there are now two expressions available $a + b, a \times b$. By analysing the label 4 with a look on table 3 it is recognisable that the expression $a + b$ is available at the entry and at the exit of the label 4 and that this expression is used in the evaluation of the while condition (see listing 1). This means, that $a + b$ is always available in label 4 and that the re-computation of it is unnecessary. Since $a + b$ is assigned to the variable $x$ both times when it is generated in label 2 and label 6, the program can be optimised as shown in listing 2.

```java
public void compute(int a, int b) {
    int x = a + b;
    int y = a * b;
    while(y > x) {
        a = a + 1;
        x = a + b;
    }
}
```

Listing 2: Java example, unnecessary re-computation optimised

The label 4 now doesn’t contain the expression $a+b$ anymore, but only its resulting value which is stored in the variable $x$.

### 3 Uninitialised variables

It is not uncommon that a programmer declares a variable at the top of his method and this variable then gets initialised depending on a condition. Which can lead to problems during compiling or during runtime, if not every variable used has been initialised properly.

#### 3.1 Example of uninitialised variables

Having a look at the code example in listing 3, there is such a conditional initialisation, but there can be a problem when the program reaches the line 8. The variable $b$ is initialised only in the else-case which is depending on the input value of the variable $a$. This means, in the case of variable $a$ being smaller or equals 0, $b$ is uninitialised at line 8 and results in a not compilable program or worse, the program will act erroneous.

```java
public void uninitialised(int a)
{
    int b;
    if(a > 0) {
        a = a + 1;
    } else {
        b = a;
    }
    a = a + b;
}
```

Listing 3: Java example, uninitialised variables

#### 3.2 Analysis with Reaching Definitions Method

The method to find possibly uninitialised variables is a forward flow analysis called Reaching Definitions. The technique analyses at each point of the program, which assignments have been made and
Figure 2: procedure flow, Uninitialised variables for listing 3

have not been changed up and until this point. With this information a variable can be defined as possibly uninitialised, if there is a path from the start of the program to the usage of the variable, along which the variable has not been assigned (initialised). Data Flow Analysis: Theory and Practice [4].

3.3 Data Flow

In the flow diagram (figure 2) it is visible, that \( b \) is initialised only in the case where the condition is not true. Which means the calculation after the if-else construct is computable only, if the if-condition was true. If the condition was false, the computation will fail.

To find out which variables are reaching which points of the program without being changed, it has to be analysed where a variables is being assigned and on which labels this has an effect (which previous or following assignments of the variable are killed) and what generation of variables are done in each label.

<table>
<thead>
<tr>
<th>( \ell )</th>
<th>killed</th>
<th>generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>4</td>
<td>{ (a,?), (a,4), (a,8) }</td>
<td>{ (a,4) }</td>
</tr>
<tr>
<td>6</td>
<td>{ (b,?), (b,6) }</td>
<td>{ (b,6) }</td>
</tr>
<tr>
<td>8</td>
<td>{ (a,?), (a,4), (a,8) }</td>
<td>{ (a,8) }</td>
</tr>
</tbody>
</table>

Table 4: Killed and generated variables

For example the label \( \ell(4) \) in table 4: the variable \( a \) is reassigned in label 4 and therefore a possible previous assignment \( (a,?) \), the current \( (a,4) \) and the following \( (a,8) \) are killed. But the variable \( a \) is as well newly generated in this label \( \ell(4) \).

3.4 Data Flow Equations

The following two equations are now used to find which variables are reaching which labels.

\[
in(\ell) = \begin{cases} 
\{ (x,?) | x \in \text{var}(\ell) \} & \text{for the first } \ell \\
\bigcup_{p \in \text{pred}(\ell)} \text{out}(p) & \text{otherwise} 
\end{cases} 
\] (3.1)

\[
\text{out}(\ell) = (\text{in}(\ell) \setminus \text{kill}(\ell)) \cup \text{gen}(\ell) 
\] (3.2)

The Reaching Definition Analysis is of the type forward flow analysis. Which means a path is followed from the top of the program to the point in question.

The first equation (3.1) adds up all defined variables from the predecessor label \( p \). It shows which variables are available at the entry of a label. The second equation (3.2)
adds the generated variables in the current
label $\ell$ to the variables available at the entry
point of $\ell$ without the variables killed in the
current label $\ell$, which shows the available
variables at the exit point of label $\ell$.

If the equations for each label are written
down, the results are the following:

In equations:

\begin{align*}
in(3) &= \{(a,?), (b,?)\} \\
in(4) &= out(3) \\
in(6) &= out(3) \\
in(8) &= out(4) \cup out(6)
\end{align*}

Out equations:

\begin{align*}
out(3) &= (in(3) \setminus \emptyset) \cup \emptyset \\
out(4) &= (in(4) \setminus \{(a,?), (a,4), (a,8)\}) \\
&\quad \cup \{(a,4)\} \\
out(6) &= (in(6) \setminus \{(b,?), (b,6)\}) \\
&\quad \cup \{(b,6)\} \\
out(8) &= (in(8) \setminus \{(a,?), (a,4), (a,8)\}) \\
&\quad \cup \{(a,8)\}
\end{align*}

3.5 Solving the problem

The solution of the resolved equations is dis-
played in table 5.

<table>
<thead>
<tr>
<th>$\ell$</th>
<th>in</th>
<th>out</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>${(a,?), (b,?)}$</td>
<td>${(a,?), (b,?)}$</td>
</tr>
<tr>
<td>4</td>
<td>${(a,?), (b,?)}$</td>
<td>${(a,4), (b,?)}$</td>
</tr>
<tr>
<td>6</td>
<td>${(a,?), (b,?)}$</td>
<td>${(a,?), (b,6)}$</td>
</tr>
<tr>
<td>8</td>
<td>${(a,?), (b,?), (a,4), (b,6)}$</td>
<td>${(a,8), (b,?), (b,6)}$</td>
</tr>
</tbody>
</table>

Table 5: Reaching Definitions

In table 5 it is visible, that the initialisa-
tion of the variable $a$ and $b$ at entry of the
label 8 ($\ell(8)$), might still be uncertain and
can result in an error when calculating the
expression $a + b$. But since $a$ is a method
interface variable only the variable $b$ might
be uninitialised.

```java
1 public void uninitialised(int a) {
2     b = 0;
3     if(a > 0) {
4         a = a + 1;
5     } else {
6         b = a;
7     }
8     a = a + b;
9 }
```

Listing 4: Java example, uninitialised
variables optimised

The listing 4 presents a corrected version
of the erroneous program shown in listing 3.
Since $b$ was not initialised at the beginning
of the method, it is now initialised with
0. A programmer has to be careful, since
some compilers have default initialisations
for primitive data types with the value of
zero. This is a change of the logic in the
program which might not be the wished
value and can lead to wrong calculations.
A uninitialised variable should rather lead
to a compiler error or a runtime error, so
that a programmer can recognise the prob-
lem and correct it.

4 Code redundancy

During developing a program it might hap-
pen, that the same arithmetic expressions
are used multiple times at different points
in the code. This might not be necessarily
a problem and in some case even wished or unpreventable. But it is much more convenient to have the expression once only in the code. This gives two advantages. First, if there is a change to the expression, it has to be change only at one point. And secondly the expression has to be computed only once during runtime.

4.1 Example of code redundancy

In the code example (listing 5), the arithmetic expression $b - a$ occurs twice, in the if-case $\ell(5)$ and the else-case $\ell(8)$ of the method, even though its variables are invariant. Obviously this example is not very critical since the expression is computed only once anyway. Due to the fact, that there is no loop and only one of the cases is being processed. But removing redundant code results in saved space of the generated program code and less memory usage. There can be other cases where a lot of performance can be gained with an optimisation of redundant or in-variant code (see section 5 Loop-invariant code).

```java
1 public void redundant(int a) {
2   int b = 3;
3   if(a > b) {
4     x = b - a;
5     y = a - b;
6   } else {
7     y = b + a;
8     x = a - b;
9   }
10 }
```

Listing 5: Java example, code redundancy

4.2 Analysis with Very Busy Expressions Method

The data flow analysis method used to find code redundancies is called Busy Expressions Analysis. This data flow analysis is of the type backward flow analysis. An expression is called very busy at a point of the program if it is guaranteed that the expression will be computed at some time in the future of the program. Thus starting at the point in question, the expression must be reached before its value changes. Which means, an expression keeps being busy as long as none of the appearing variables in the expression change.

4.3 Data Flow

The data flow diagram (figure 3) shows that the expression $a - b$ is calculated independent from the if-condition, since it appears in the if- (label 5) and the else-case (label 8).

First it is determined where the arithmetic expressions are generated respectively killed. An expression is killed if one of its variables is changed before the expression is generated (computed). This means, that at the start of a program it is assumed, that all expressions are very busy. But in table 6 it is visible, that all arithmetic expressions are killed in the label 2 of the program, since the variable $b$ is changed respectively initialised and $b$ is used in every expression of the program. After that, the expressions keep being busy until they are used (computed).
4.4 Data Flow Equations

With the following equations the very-busy-paths can be made visible with the information of the killed and generated expressions (table 6).

\[ \text{in}(\ell) = (\text{out}(\ell) \setminus \text{kill}(\ell)) \cup \text{gen}(\ell) \]  
(4.1)

\[ \text{out}(\ell) = \begin{cases} \emptyset & \text{for the last } \ell \\ \bigcap_{s \in \text{succ}(\ell)} \text{in}(s) & \text{otherwise} \end{cases} \]  
(4.2)

In the first equation (4.1) the expressions killed in the current label are subtracted from the expressions available at the exit point of the current label and then united with the generated expressions of the same label.

The second equation (equation 4.2) than unites the expressions of all successor labels which means those are the busy expressions at the exit point of the current label. If the equation for each label is written down, they look like the following:

<table>
<thead>
<tr>
<th>\ell</th>
<th>killed</th>
<th>generated</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>{b-a,a-b,b+a}</td>
<td>\emptyset</td>
</tr>
<tr>
<td>3</td>
<td>\emptyset</td>
<td>\emptyset</td>
</tr>
<tr>
<td>4</td>
<td>\emptyset</td>
<td>{a-b}</td>
</tr>
<tr>
<td>5</td>
<td>\emptyset</td>
<td>{a-b}</td>
</tr>
<tr>
<td>7</td>
<td>\emptyset</td>
<td>{b+a}</td>
</tr>
<tr>
<td>8</td>
<td>\emptyset</td>
<td>{a-b}</td>
</tr>
</tbody>
</table>

Table 6: Killed and generated expressions with respect to a and b

Figure 3: procedure flow, Code Redundancy for listing 5
In equations:

\begin{align*}
\text{in}(2) &= \{ \text{out}(2) \setminus \{ b - a, a - b, b + a \}\} \cup \emptyset \\
\text{in}(3) &= \{ \text{out}(3) \setminus \emptyset\} \cup \emptyset \\
\text{in}(4) &= \{ \text{out}(4) \setminus \emptyset\} \cup b - a \\
\text{in}(5) &= \{ \text{out}(5) \setminus \emptyset\} \cup a - b \\
\text{in}(7) &= \{ \text{out}(7) \setminus \emptyset\} \cup b + a \\
\text{in}(8) &= \{ \text{out}(8) \setminus \emptyset\} \cup a - b \\
\end{align*}

Out equations:

\begin{align*}
\text{out}(2) &= \text{in}(3) \\
\text{out}(3) &= \text{in}(4) \cap \text{in}(7) \\
\text{out}(4) &= \text{in}(5) \\
\text{out}(5) &= \emptyset \\
\text{out}(7) &= \text{in}(8) \\
\text{out}(8) &= \emptyset \\
\end{align*}

4.5 Solving the problem

As a second step the very-busy-path is visualised, i.e. the expression for each label at the entry and the exit points are listed. The very-busy-path is built up backwardly. Which means that the out equation of the label(s) closest to the exit point of the program, have an empty set. E.g., the path for the expression \( a - b \) at \( \ell(8) \) is followed backwards up until \( \ell(2) \).

Due the backwards analysis in table it is evident, that the arithmetic expression \( a - b \) is very busy up and until label \( \ell(5) \) in the if-case and label \( \ell(8) \) in the else-case. So there is an obvious optimisation, to the method as shown in listing. The expression \( a - b \) has been hoisted before the if-evaluation. This type of code editing is called Code Hoisting. A synonym for the verb hoist is e.g. lift or pull up. In the case of code hosting, this means that code parts (expressions) are pulled up to a more optimal position in the program.

\begin{verbatim}
public void redundant(int a) {
    int b = 3;
    int c = a - b;
    if(a > b) {
        x = b - a;
        y = c;
    } else {
        y = b + a;
        x = c;
    }
}
\end{verbatim}

Listing 6: Java example, code redundancy optimised

5 Loop-invariant code

In section 4 Code redundancy it was shown, that the Very Busy Expression Analysis is a valuable method to detect code redundancy and to find possible code hoistings. In this chapter another method to recognise possible code hoistings is discussed.
5.1 Example of loop-invariant code

The example in listing 7 shows two calculations within a while loop. The problem with this code is, that the arithmetic expression \( b + c \), is *invariant*. Neither of the variables used in the calculation \( b + c \) is changed, but it is recomputed in every loop cycle. For larger loops (in the number of cycles), such invariant code is a huge performance loss, since the recomputing is unnecessary.

1. ```java
   public void motion(int a) {
     int b = 2;
     int c = 3;
     int d;
     while(a < 10) {
       d = b + c;
       a = a + d;
     }
   }
```  
Listing 7: Java example, loop-invariant code

5.2 Analysis with Derived Data Flow Information

To find possible code motions (hoistings), to improve the performance within a program, a method called **Derived Data Flow Information** can be used. This method creates chains along the problem which links 

- ... each use of variable with its assignments \( \rightarrow \) *Use-Definition chains* (ud-chains)
- ... each assignment of a variable with its uses \( \rightarrow \) *Definition-Use chains* (du-chains)

This technique is not an equational based approach like the other techniques described in this document (e.g. Reaching Definitions Analysis). Derived Data Flow Information consists of two functions which return the values to build up the ud- respectively the du-chain.

The two functions are:

1. \( ud(x, \ell) \) The function receives the name of a variable and the name of the label to analyse and returns the label from which the variable might have obtained its value. The result can be more than one label.

2. \( du(x, \ell) \) The function receives as well the name of a variable and the name of the label to analyse but returns all labels where the value which is assigned at the current label, might be used.

5.3 Data Flow

The data flow diagram (figure 4) shows the label 6 right after the while condition which contains the in-variant expression \( b + c \). The expression \( a + d \) is not in-variant, since the variable \( a \) changes in the same label.

The table 8 for ud-chains shows the values of the function \( ud(x, \ell) \). E.g. the variable \( b \) used in label 6 gets its value in label 2.
\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|}
\hline
ud(x,\ell) & a & b & c & d \\
\hline
 2 & \emptyset & \emptyset & \emptyset & \emptyset \\
 3 & \emptyset & \emptyset & \emptyset & \emptyset \\
 5 & \{?\} & \emptyset & \emptyset & \emptyset \\
 6 & \emptyset & \{2\} & \{3\} & \emptyset \\
 7 & \{?\} & \emptyset & \emptyset & \{6\} \\
\hline
\end{tabular}
\caption{ud-chain}
\end{table}

The table \ref{table:du-chain} for du-chains shows the values of the function du(x, \ell). E.g. the value assigned to the variable b may be used in label 6.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|}
\hline
du(x,\ell) & a & b & c & d \\
\hline
 2 & \emptyset & \{6\} & \emptyset & \emptyset \\
 3 & \emptyset & \emptyset & \{6\} & \emptyset \\
 5 & \emptyset & \emptyset & \emptyset & \emptyset \\
 6 & \emptyset & \emptyset & \emptyset & \{7\} \\
 7 & \{5\} & \emptyset & \emptyset & \emptyset \\
 ? & \{5,7\} & \emptyset & \emptyset & \emptyset \\
\hline
\end{tabular}
\caption{du-chain}
\end{table}

\subsection*{5.4 Solving the problem}

With table \ref{table:ud-chain} and table \ref{table:du-chain} it is possible to detect possible code motions. In the optimised program shown in listing \ref{listing:optimized} the loop-invariant code b + c has been moved above the loop (hoisted). This expression will now be computed only once and not for every loop cycle. This code move is possible because in the ud-chain table \ref{table:ud-chain} it is shown that the variables used in label 6 are assigned in earlier labels (2 and 3). And the du-chain table \ref{table:du-chain} shows that the variables b and c are not used before label 6.
public void motion(int a) {
    int b = 2;
    int c = 3;
    int d = b + c;
    while(a < 10) {
        a = a + d;
    }
}

Listing 8: Java example, loop-invariant code optimised

public void unused(int b) {
    int a = 7;
    int c;
    int d = 3;
    a = 5;
    if(a > b) {
        c = a;
    } else {
        c = a + b;
        b = c;
    }
}

Listing 9: Java example, unused variables

6 Unused variables

There might be the case, that a developer writes a program and initialises a variable at the top of his program with a value. But even before the variable is used, he assigns a new value to it. This is one case of an unused variable. A second case would be, if a variable is initialised but not even used in the code.

6.1 Example of unused variables

In the example program listing 9 it is recognisable, that the variable a is initialised ($\ell(2)$) and before it is ever use, it is already reassigned ($\ell(5)$). The second issue is an unnecessary variable. The variable d is defined and initialised, but it is never used in the program.

6.2 Analysis with Live Variables method

The data flow analysis method to discover possibly unused variables (dead code) is called Live Variables Analysis, which is a backward data flow analysis technique. This analysis defines for each program point which variables will be used in the future before they will be reassigned. This means, a variable stays alive as long as its value does not change.

6.3 Data Flow

The data flow diagram (figure 5) shows that the variable a is assigned twice, once in label 2 and once in label 5, but is is not used in between those two labels. Variable d is assigned in label 4 but not used in the whole program.

In this analysis a variable is killed as soon as it has been newly assigned and it is named generated, when it is used without
being changed. The killed and generated state is shown in table 10.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
\(\ell\) & killed & generated \\
\hline
2 & \{a\} & \emptyset \\
4 & \{d\} & \emptyset \\
5 & \{a\} & \emptyset \\
6 & \emptyset & \{a,b\} \\
7 & \{c\} & \{a\} \\
9 & \{c\} & \{a,b\} \\
10 & \{b\} & \{c\} \\
\hline
\end{tabular}
\caption{Killed and generated exp.}
\end{table}

6.4 Data Flow Equations

The following two equations are now used to find out which variables are live at which point of the program.

\begin{align*}
\text{out}(\ell) &= \begin{cases}
\emptyset & \text{for the last } \ell \\
\bigcup_{s \in \text{succ}(\ell)} \text{in}(s) & \text{otherwise}
\end{cases} \quad (6.1) \\
\text{in}(\ell) &= \text{out}(\ell) \setminus \text{kill}(\ell) \cup \text{gen}(\ell) \quad (6.2)
\end{align*}

The equation 6.1 shows the variables which are live at the exit of a label. It is the union of the entry variables of all succeeding labels. If the label \(\ell\) is the last one of the program there are no live variables any more which is represented by an empty set (\(\emptyset\)).

The second equation 6.2 shows the live variables at the entry point of a label. It is composed by the live variables at the exit point of \(\ell\) without the killed variables in \(\ell\) but joined with the generated variables of the same \(\ell\).

The equations written down for each la-
bel are the following:

In equations:

\[ in(2) = (out(2) \setminus \{a\}) \cup \emptyset \]  \hfill (6.3)

\[ in(4) = (out(4) \setminus \{d\}) \cup \emptyset \]  \hfill (6.4)

\[ in(5) = (out(5) \setminus \{a\}) \cup \emptyset \]  \hfill (6.5)

\[ in(6) = (out(6) \setminus \emptyset) \cup \{a, b\} \]  \hfill (6.6)

\[ in(7) = (out(7) \setminus \{c\}) \cup \{a\} \]  \hfill (6.7)

\[ in(9) = (out(9) \setminus \{c\}) \cup \{a, b\} \]  \hfill (6.8)

\[ in(10) = (out(10) \setminus \{b\}) \cup \{c\} \]  \hfill (6.9)

Out equations:

\[ out(2) = in(4) \]  \hfill (6.10)

\[ out(4) = in(5) \]  \hfill (6.11)

\[ out(5) = in(6) \]  \hfill (6.12)

\[ out(6) = in(7) \cup in(9) \]  \hfill (6.13)

\[ out(7) = in(10) \]  \hfill (6.14)

\[ out(9) = in(10) \]  \hfill (6.15)

\[ out(10) = \emptyset \]  \hfill (6.16)

#### 6.5 Solving the problem

The solutions of the equations result in the values shown in the table 11.

<table>
<thead>
<tr>
<th>( \ell )</th>
<th>\text{in}</th>
<th>\text{out}</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>{b}</td>
<td>{b}</td>
</tr>
<tr>
<td>4</td>
<td>{b}</td>
<td>{b}</td>
</tr>
<tr>
<td>5</td>
<td>{b}</td>
<td>{a,b}</td>
</tr>
<tr>
<td>6</td>
<td>{a,b}</td>
<td>{a,b}</td>
</tr>
<tr>
<td>7</td>
<td>{a}</td>
<td>{c}</td>
</tr>
<tr>
<td>9</td>
<td>{a,b}</td>
<td>{c}</td>
</tr>
<tr>
<td>10</td>
<td>{c}</td>
<td>\emptyset</td>
</tr>
</tbody>
</table>

Table 11: Live Variables Analysis

In the optimised program listing 10 the initialisation of the variable \( a \) is now done directly in label 2 with the value 5. The table 11 shows that the variable \( a \) is not live until the exit of label 5, i.e. the first assignment is useless. Also it is visible that the variable \( d \) is never live and therefore never used in the program. For this reason the definition and initialisation of the variable \( d \) has completely removed.

```java
1  public void unused(int b) {
2      int a = 5;
3      int c;
4      if(a > b) {
5          c = a;
6      } else {
7          c = a + b;
8          b = c;
9      }
10 }
```

Listing 10: Java example, unused variables optimised

#### 7 Conclusion

The inside of a compiler is very complex and often not well known by developers using high level languages. Fact is that most of today’s running software would perform a lot less if the compilers were not having an in-built optimising module which is using the intraprocedural data flow analysis methods examined in this article. Maybe a more common usage of these data flow analysis methods are built-in in most of the modern program-code editors or Integrated Development Environments (IDEs). The IDEs contain code analysers, which provide graphical hints or notifications about
optimization possibilities to the programmer. The code developer can than solve those problems before he starts compiling. Nevertheless the intraprocedural data flow analysis techniques are just a small but very important tool to optimize a program. And in days where more and more complex software has to run on the smallest devices with limited resources, the optimising part is not loosing its significance. If I would have to chose the most important analysis method of the ones described in this article, it would probably be the Very Busy Expression Analysis or the Derived Data Flow Information to find code redundancies. Especially to find loop-invariant code which can be a huge performance loss. But who wants to have uninitialised variables in his program, therefore the Reaching Definitions Analysis is also very important.

References


