Side-Effect Removal Transformation

Mark Harman, **Lin Hu**,
Brunel University,
Uxbridge, Middlesex,
UB8 3PH, UK.
Tel: +44 (0)1895 274 000
Fax: +44 (0)1895 251 686
Mark.Harman@brunel.ac.uk

Malcolm Munro, **Xingyuan Zhang**,
University of Durham,
South Road, Durham,
DH1 3LE, UK.
+44 (0) 191 374 2634
+44 (0) 191 374 2560
Malcolm.Munro@durham.ac.uk

Keywords: Transformation, Side effects

Abstract

A side effect is any change in program state that occurs as a by-product of the evaluation of an expression. Side effects are often thought to impede program comprehension and to lead to complex, poorly understood and occasionally undefined semantics.

Side-Effect Removal Transformation (SERT) improves comprehension by rewriting a program p which may contain side effects into a semantically equivalent program p' which is guaranteed to be side-effect free.

This paper introduces the SERT approach to the side-effect problem, briefly reporting initial experience with an implementation of SERT for C programs called Linsert[1]*.*

1 Introduction

A side effect is any state change caused by the evaluation of an expression. A side–effect free expression, when evaluated simply returns its value, causing no change in state. The Programming Language C Standard defines side effects in a more general way:

> "Accessing a volatile object, modifying an object, modifying a file, or calling a function that does any of those operations are all *side effects*, which are changes in the state of the execution environment. Evaluation of an expression may produce side effects." (Section 5.1.2.3 of [1].)

In this paper only the side effect of updating the value of a variable will be considered. The approach presented here could be extended to handle the more challenging problem of removing the general class of side-effects identified in the quotation above, but this currently remains a problem for future work.

The presence of side–effects inhibits the application of many software engineering techniques, such as symbolic execution [7], slicing [4, 34, 13], partial evaluation [2, 16] and transformation [33, 28, 30], which typically work on side effect-free systems[2].

The presence of side-effects is also widely believed to inhibit program comprehension. For instance, Kernighan and Pike [23], advise abstinence from side effects, in all but, well-understood, special cases, saying

> Don't use side effects except for a very few idiomatic constructions like
>
> ```
> a[i++] = 0;
> c = *p++;
> *s++ = *t++;
> ```

Other authors also warn against the problems that side effects can cause [32, 29].

Programming style guidelines and standards also tend to deprecate the use of side effects in all but special cases. For example, Paul Haahr [17] says

> Good rules of thumb are that non-compound statements should have one side-effect each and expressions should rarely have side-effects.

Cannon et al. [5] describe the way in which side effects work against program maintainability:

[1] The Linsert tool is available (for non-commercial use) at http://www.brunel.ac.uk/~csstmmh2/linsert.

[2] Such approaches can usually be defined to handle side effects, but this often requires effort disproportionate to the perceived gain.

> One should consider the tradeoff between increased speed and decreased maintainability that results when embedded assignments are used in artificial places. For example,
>
> ```
> a = b + c;
> d = a + r;
> ```
>
> should not be replaced by
>
> ```
> d = (a = b + c) + r;
> ```
>
> even though the latter may save one cycle. In the long run the time difference between the two will decrease as the optimizer gains maturity, while the difference in ease of maintenance will increase as the human memory of what's going on in the latter piece of code begins to fade.

The transformation-based approach advocated in the present paper would, in fact, perform the opposite transformation, replacing the side-effecting code

```
d = (a = b + c) + r;
```

with the preferred side effect free version:

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

The reader who is in any doubt as to the way in which side effects obfuscate the meaning of an expression need only consider the simple examples in Figure 1. Each is only a single short assignment statement, involving an expression with at most two side-effects. However, it is something of a challenge to state with certainty what the meaning of each assignment is.

The side-effect removal transformations described here clarifies the meaning of these assignments, producing the results presented in Figure 2. The semantics of some of the assignments (and any program which contains them) are 'undefined' according to the ISO C standard. This 'undefined semantics' will be taken into account in side-effect removal transformation.

This paper introduces a source–to–source transformation algorithm which transforms a sequence of statements with side effects into an equivalent sequence of statements which are side–effect free. There are three reasons why side effect removal is useful:

1. It makes existing tools more widely applicable;

`a = ++x + x;`
`b = (++x, ++x);`
`c = ++x + ++x;`
`d = ++x && ++x;`
`f = (x=4) + (y=5);`
`g = (x=4) + (x=5);`

Figure 1: Some Examples of Side Effects

2. It seems likely that it will tend to improve program comprehension;
3. The transformation serves to clarify the semantics of the programming language.

The rest of the paper is organised as follows.

Section 2 provides the theoretical foundations of Side Effect Removal Transformation (SERT). Section 3 describes three strategies for achieving side effect removal. SERT can lead to an increase in program size, because the side effect free program has to take account of the effect of the side effects in the original. Section 4 illustrates the way in which post processing transformation can be applied to the side-effect free program produced by Linsert. Linsert is the authors' current implementation of SERT, which uses one of the three strategies (post placement) described in Section 3. Section 5 briefly describes the relationship between the side effect *removal* approach advocated in the present paper, with previous literature on side effect *detection*.

2 Theoretical Foundations

This section presents the definitions of 'side effect', the subset of `C` (relevant to the study of side effects) accepted by the Linsert tool and formalises the concept of valid expressions (i.e. those whose semantics is 'defined' according to the C standard [1]).

Linsert implements side-effect removal for a subset of the full `C` language, called `C--`. The syntax of `C--` expressions is defined in Figure 3. The domain I, is the syntactic domain of identifiers and N is the domain of numerals for the natural numbers. The syntax of statements is defined in Figure 4. The language `C--` contains sufficient statement and expression syntax from full `C` to allow all intraprocedural side effect issues to be considered.

As with the full language `C`, an expression can be 'converted' into a statement by suffixing it with a semicolon (such statements are known as *expression–statements* [24, 1]). In practice, an expression–statement will only be useful if it performs side–effects, because its value is ignored. Also, in keeping

`a = ++x + x;`	`undefined`
`b = (++x, ++x);`	`b = x+2;  x = x+2;`
`c = ++x + ++x;`	`undefined`
`d = ++x && ++x;`	`if (x == -1) { d = 0;  x = 0; }` `else { d = x+2!=0;  x = x+2; }`
`f = (x=4) + (y=5);`	`f = 9;  x = 4;  y = 5;`
`g = (x=4) + (x=5);`	`undefined`
Original	Side-Effect Free Version

Figure 2: Side-effecting Statements with Side-Effect Free Versions

$$
\begin{array}{rcl}
E & ::= & \texttt{PreOp}\ I \\
 & | & I\ \texttt{PostOp} \\
 & | & E_1\ \texttt{BOp}\ E_2 \\
 & | & I = E \\
 & | & I \\
 & | & \mathrm{N} \\
 & | & I[E] \\
 & | & E_1, E_2 \\
 & | & E_1 ? E_2 : E_3 \\
\texttt{PostOp} & ::= & ++ \mid -- \\
\texttt{PreOp} & ::= & \texttt{PostOp} \mid ! \\
\texttt{BOp} & ::= & \texttt{LOp} \mid \texttt{AOp} \\
\texttt{AOp} & ::= & + \mid - \mid * \mid \% \mid == \mid != \\
\texttt{LOp} & ::= & \texttt{\&\&} \mid ||
\end{array}
$$

Figure 3: The Syntax of Expressions in `C--`.

$$
\begin{array}{rcl}
C & ::= & E; \\
 & | & \textbf{if}(E)\ C \\
 & | & \textbf{if}(E)\ C_1\ \textbf{else}\ C_2 \\
 & | & \textbf{while}(E)\ C \\
 & | & \textbf{do}\ C\ \textbf{while}\ (E) \\
 & | & \textbf{for}(E_1; E_2; E_3)\ C \\
 & | & \{C_1\ C_2\}
\end{array}
$$

Figure 4: The syntax of statements in `C--`.

with the full `C` language, numeric values are used to denote boolean values. When used as a boolean, any non–zero arithmetic value denotes 'true', with zero denoting 'false'.

In side-effect free programs defined variables are easy to describe. They are the left-hand-sides of assignment statements. In side-effecting programs, it is also necessary to describe the defined variables of expressions. These are the variables whose values may be altered by the evaluation of the expression. The function **DEF** of Figure 5 takes and expression and returns the set of variables whose value may be altered by the evaluation of the expression.

Definition 6 defines the referenced variables for the language `C--`. This is used in Definition 7 to define the expressions whose semantics is valid (that is, those whose semantics are 'defined' according to the C standard [1]).

The following definitions provide a more rigorous definition of Side Effect Removal Transformation.

$$
\begin{array}{rcl}
\textbf{DEF} & : & E \to \mathcal{P}(I) \\
\textbf{DEF}[\![++I]\!] & = & \{I\} \\
\textbf{DEF}[\![--I]\!] & = & \{I\} \\
\textbf{DEF}[\![I++]\!] & = & \{I\} \\
\textbf{DEF}[\![I--]\!] & = & \{I\} \\
\textbf{DEF}[\![!E]\!] & = & \textbf{DEF}[\![E]\!] \\
\textbf{DEF}[\![E_1\ \texttt{AOp}\ E_2]\!] & = & \textbf{DEF}[\![E_1]\!] \cup \textbf{DEF}[\![E_2]\!] \\
\textbf{DEF}[\![E_1 \texttt{\&\&} E_2]\!] & = & \textbf{DEF}[\![E_1]\!] \cup \textbf{DEF}[\![E_2]\!] \\
\textbf{DEF}[\![E_1 || E_2]\!] & = & \textbf{DEF}[\![E_1]\!] \cup \textbf{DEF}[\![E_2]\!] \\
\textbf{DEF}[\![I = E]\!] & = & \{I\} \\
\textbf{DEF}[\![I]\!] & = & \{\} \\
\textbf{DEF}[\![\mathrm{N}]\!] & = & \{\} \\
\textbf{DEF}[\![I[E]]\!] & = & \textbf{DEF}[\![E]\!] \\
\textbf{DEF}[\![E_1, E_2]\!] & = & \textbf{DEF}[\![E_1]\!] \cup \textbf{DEF}[\![E_2]\!] \\
\textbf{DEF}[\![E_1 ? E_2 : E_3]\!] & = & \textbf{DEF}[\![E_1]\!] \cup \\
 & & \textbf{DEF}[\![E_2]\!] \cup \textbf{DEF}[\![E_3]\!]
\end{array}
$$

Figure 5: Defined Variables of Expressions

$$
\begin{array}{rcl}
\mathbf{VALID} & : & E \to \{T, F\} \\
\mathbf{VALID}[\![{+}{+}I]\!] & & \\
\mathbf{VALID}[\![{-}{-}I]\!] & & \\
\mathbf{VALID}[\![I{+}{+}]\!] & & \\
\mathbf{VALID}[\![I{-}{-}]\!] & & \\
\mathbf{VALID}[\![!E]\!] & \text{iff} & \mathbf{VALID}[\![E]\!] \\
\mathbf{VALID}[\![E_1 \ \texttt{AOp} \ E_2]\!] & \text{iff} & \mathbf{VALID}[\![E_1]\!] \wedge \mathbf{VALID}[\![E_2]\!] \ \wedge \\
& & \mathbf{REF}[\![E_1]\!] \cap \mathbf{DEF}[\![E_2]\!] = \emptyset \ \wedge \\
& & \mathbf{DEF}[\![E_1]\!] \cap \mathbf{REF}[\![E_2]\!] = \emptyset \ \wedge \\
& & \mathbf{DEF}[\![E_1]\!] \cap \mathbf{DEF}[\![E_2]\!] = \emptyset \\
\mathbf{VALID}[\![E_1 \texttt{\&\&} E_2]\!] & \text{iff} & \mathbf{VALID}[\![E_1]\!] \wedge \mathbf{VALID}[\![E_2]\!] \\
\mathbf{VALID}[\![E_1 || E_2]\!] & \text{iff} & \mathbf{VALID}[\![E_1]\!] \wedge \mathbf{VALID}[\![E_2]\!] \\
\mathbf{VALID}[\![I = E]\!] & \text{iff} & \mathbf{VALID}[\![E]\!] \wedge I \notin \mathbf{DEF}[\![E]\!] \\
\mathbf{VALID}[\![I]\!] & & \\
\mathbf{VALID}[\![\mathrm{N}]\!] & & \\
\mathbf{VALID}[\![I[E]]\!] & \text{iff} & \mathbf{VALID}[\![E]\!] \\
\mathbf{VALID}[\![E_1, E_2]\!] & \text{iff} & \mathbf{VALID}[\![E_1]\!] \wedge \mathbf{VALID}[\![E_2]\!] \\
\mathbf{VALID}[\![E_1 ? E_2 : E_3]\!] & \text{iff} & \mathbf{VALID}[\![E_1]\!] \wedge \mathbf{VALID}[\![E_2]\!] \wedge \mathbf{VALID}[\![E_3]\!]
\end{array}
$$

Figure 7: Valid Expressions: Those Not Undefined According to the `C` Standard

$$
\begin{array}{rcl}
\mathbf{REF} & : & E \to \mathcal{P}(I) \\
\mathbf{REF}[\![{+}{+}I]\!] & = & \{I\} \\
\mathbf{REF}[\![{-}{-}I]\!] & = & \{I\} \\
\mathbf{REF}[\![I{+}{+}]\!] & = & \{I\} \\
\mathbf{REF}[\![I{-}{-}]\!] & = & \{I\} \\
\mathbf{REF}[\![!E]\!] & = & \mathbf{REF}[\![E]\!] \\
\mathbf{REF}[\![E_1 \ \texttt{AOp} \ E_2]\!] & = & \mathbf{REF}[\![E_1]\!] \cup \mathbf{REF}[\![E_2]\!] \\
\mathbf{REF}[\![E_1 \texttt{\&\&} E_2]\!] & = & \mathbf{REF}[\![E_1]\!] \cup \mathbf{REF}[\![E_2]\!] \\
\mathbf{REF}[\![E_1 || E_2]\!] & = & \mathbf{REF}[\![E_1]\!] \cup \mathbf{REF}[\![E_2]\!] \\
\mathbf{REF}[\![I = E]\!] & = & \mathbf{REF}[\![E]\!] \\
\mathbf{REF}[\![I]\!] & = & \{I\} \\
\mathbf{REF}[\![\mathrm{N}]\!] & = & \{\} \\
\mathbf{REF}[\![I[E]]\!] & = & \mathbf{REF}[\![E]\!] \cup \{I\} \\
\mathbf{REF}[\![E_1, E_2]\!] & = & \mathbf{REF}[\![E_1]\!] \cup \mathbf{REF}[\![E_2]\!] \\
\mathbf{REF}[\![E_1 ? E_2 : E_3]\!] & = & \mathbf{REF}[\![E_1]\!] \cup \\
& & \mathbf{REF}[\![E_2]\!] \cup \mathbf{REF}[\![E_3]\!]
\end{array}
$$

Figure 6: Referenced Variables of Expressions

Definition 1 (Side Effect)

An expression E contains a side–effect if it assigns a value to any variable.

More formally, E contains a side–effect iff $\mathbf{DEF}(E) \neq \emptyset$.

Definition 2 (Side–Effect Free)

A program is side–effect free iff the expressions of all statements are side–effect free, except for expression–statements of the form $I = E$;. For such expression statements, E must be a side–effect free expression.

Thus, in a side–effect free program, all variable updates take place using the familiar assignment statement, the right-hand side of which is a side–effect free expression. This 'special case' of an expression-statement arises because of the way `C` allows any expression to become a statement merely by virtue of the addition of a terminating semicolon. Thus, strictly speaking, the statement, $I = E$;, is an expression statement, the expression of which is the assignment expression $I = E$. This complicates the definition of 'side–effect free' slightly, because allowance must be made for this one special case of an expression, which *is* allowed to contain precisely one side effect.

Side effect removal can now be defined as the process of replacing a program with an equivalent side–effect free version, using some transformation procedure $\mathcal{A}$.

Definition 3 (Side Effect Removal)
A valid side effect removal algorithm, $\mathcal{A}$, is a transformation from program fragments to program fragments such that for all fragments p, $\mathcal{A}(p)$ behaves identically to p and $\mathcal{A}(p)$ is side–effect free.

3 Strategies for Side Effect Removal

There are three strategies for side effect removal, listed below. The first and third of these are applicable in all cases. The second, though desirable, is sadly not applicable in all cases. These three strategies are:

1. Place side effects **after** the evaluation of side-effect free version of the original expression.
   This has the advantage that there will always be a side-effect free equivalent of the original expression, so the transformation will always be possible. It has the disadvantage that it may require copying of the code which models the side-effects, creating a large increase in overall code size. This approach will be termed '**post-placement** of side effects'.

2. Place side effects **before** the evaluation of side-effect free version of the original expression.
   This has the disadvantage that it will not always be possible (as will be seen). This approach will be termed '**pre-placement** of side effects'. Where it is possible, pre-placement is often desirable, because post-placement of side-effects can lead to large increases in overall code size, which pre-placement can avoid.

3. Introduce **temporary variables** to capture the initial values of all variables used in the expression. This allows the placement of side effects **either before or after** the evaluation of side-effect free version of the original expression.
   This approach has the disadvantage that it introduces temporary variables and assignments to them. This may add to the burden of comprehension and will provide more work for a subsequent system which has to process the side-effect free code. However, the introduction of temporaries allows greater flexibility in the choice of where to place side effects.

Strategies 1 and 3 are related. Post-placement can cause introduction of 'too much code', whilst temporary variable introduction can cause the introduction 'too many variables'. However, post-processing the output of the side effect removal transformation can alleviate these problems by removing unnecessary temporaries and code.

An example will help clarify the difference between the three approaches.

3.1 Example

Consider the program fragment below

```
if (m!=x && (m=y,j)!=x) z = 1;
```

The expression `m!=x && (m=y,j)!=x` is evaluated using short circuit evaluation. This is modelled using the conditional expression `?:`, giving the equivalent (and still side–effecting) expression `(m!=x)?(m=y,j)!=x:0`. The side effect of this expression is captured by the statement:

```
if (m!=x) m=y;
```

However, the value of the expression `m!=x && (m=y,j)!=x` depends upon the original value of the variable `m`, which will be lost if the assignment `m=y;` precedes the evaluation of the side-effect free version of the expression. Thus, for the side-effect free version to evaluate correctly at some time after the side effects have taken place, the original value of the variable `m` must be stored in some other variable.

Alternatively, the original code fragment can be transformed into a side-effect free form, using the post-placement strategy (strategy 1). However, this example illustrates the way in which this strategy tends to increase the amount of code. The post-placement approach would yield the side-effect free form below:

```
if (m!=x ? j!=x : 0)
   { if (m!=x) m=y; z = 1; }
else if (m!=x) m=y;
```

The approach places the side effects in the `then` and `else` parts of the conditional. Because of the copying of the side effects (required by the post-placement approach) the side-effect free version of the code fragment is rather larger than one might prefer. Fortunately, standard simplification transformations can be used to simplify the code that results from side-effect removal.

In this case, the assignment to `m` in the `then` part can be moved to the end of the `then` part, giving:

```
if (m!=x && j!=x)
   { z = 1; if (m!=x) m=y; }
else if (m!=x) m=y;
```

An assignment which appears at the end of both `then` and `else` parts of a conditional can be factored out. Applying this rule gives:

```
if (m!=x && j!=x)  z = 1;
if (m!=x) m=y;
```

These post processing transformations can be automated using a transformation tool such as the Maintainers' Assistant [33].

4 A Worked Example

Linsert is an implementation of post-placement side effect removal. It is available (for non-commercial use) and can be downloaded from the web at

`http://www.brunel.ac.uk/~csstmmh2/linsert.`

Linsert uses a (very simple) symbolic executor to determine a side-effect free version of the expression. Since only expression operators, sequence points and conditionals are involved, the AST of expressions is an acyclic graph. The absence of cycles means that the usual problems which loops present to symbolic executors [7, 9, 10] disappear, and it becomes possible to completely determine symbolic values in all cases.

This section presents a worked example, showing how the output of Linsert is initially much longer than the original program. To some extent this merely reflects the 'hidden complexity' of the side effect-based semantics of the original. However, it also arises as a result of the copying inherent in the post placement strategy.

Fortunately, standard approaches to program simplification, (which also use transformation) become applicable to the output of Linsert because it is side-effect free. Using these standard transformations, it becomes possible to simplify Linsert's output, yielding a version of the original program which is both side-effect free and comparatively simple.

Consider the side-effect-ridden statement below:

```
if(a++,++a&&(b||++c)) x=c;
```

For this program, Linsert produces the side-effect free program fragment:

```
if(a+2&&(b||c+1)){
    a=a+1;
    if( a+1 ){
        a=a+1;
        if( b ){}
        else {c=c+1;}
    }
    else {a=a+1;}
    x=c;
}
else{
    a=a+1;
    if( a+1 ){
        a=a+1;
        if( b ){}
        else {c=c+1;}
    }
    else {a=a+1;}
}
```

At first glance, this may not appear to be much of an improvement over the original side-effecting version of the program fragment. However, it can be simplified using the ConSIT system [12]. ConSIT was designed to produce conditioned slices [14, 4]. That is, it strips out statements which are not reachable when user-selected initial conditions are imposed. However, if no conditions are imposed, ConSIT behaves like a dead-code eliminator, removing statements which are unreachable regardless of the initial conditions imposed. Although the problem of removing unreachable statements is not generally decidable [35], the unreachable code produced by side effect removal is a very simple case which can be handled quite easily by ConSIT.

For the fragment above, ConSIT produces the following simplified equivalent:

```
if(a+2&&(b||c+1)){
   a=a+1;
   a=a+1;
   if(b);
   else c=c+1;
   x=c;
}
else{
    a=a+1;
    if(a+1){
        a=a+1;
        c=c+1;
    }
    else a=a+1;
}
```

Finally, a sequence of standard simplification transformations used in amorphous program slicing [20, 19] can be applied to complete the simplification process. These rules are summarised in Figure 8. The term $\mathbf{SUB}(e_1, i, e_2)$ returns the expression that results from substituting all occurrences of the variable i in the expression e_1, with the expression e_2.

Using Rule 1, the two assignments `a=a+1; a=a+1;` are transformed to the single (unfolded) assignment `a=a+2;`. Also the statement `if(b); else c=c+1;` is simplified to `if(!b) c=c+1;` using Axiom 1.

Using Rule 2 and Axiom 2, the statement

```
if(a+1){a=a+1;c=c+1;} else a=a+1;
```

is transformed into

```
if(a+1) c=c+1; a=a+1;
```

This sequence of transformations yields:

Axiom 1 **if(e){} else c** $\Rightarrow$ **if(!e) c**

Axiom 2 **if(e){ $\mathbf{c_1}$ c } else{ $\mathbf{c_2}$ c }** $\Rightarrow$ **if(e){ $\mathbf{c_1}$ } else{ $\mathbf{c_2}$ } c**

Rule 1 (Unfold Assignment)

$$\frac{e3_{=}\mathrm{SUB}(e_2, i_1, e_1)}{[\![i = e_1;\ i = e_2;]\!] \Rightarrow [\![i = e_3;]\!]}$$

Rule 2 (Push Assignment)

$$\frac{i_1 \neq i_2, i_2 \notin \mathrm{REF}(e_1), e_3 = \mathrm{SUB}(e_2, i_1, e_1)}{[\![i_1 = e_1;\ i_2 = e_2;]\!] \Rightarrow [\![i_2 = e_3;\ i_1 = e_1;]\!]}$$

Rule 3 (Push If)

$$\frac{e_2' = \mathbf{SUB}(e_2, i, e_1), [\![i{=}e_1;\ c]\!] \Rightarrow [\![c'\ i{=}e_1;]\!]}{[\![i{=}e_1;\ \mathtt{if}\ (e_2)\ c]\!] \Rightarrow [\![\mathtt{if}\ (e_2')\ c'\ i{=}e_1;]\!]}$$

Figure 8: Transformation Axioms and Rules of Post Processing

```
if(a+2&&(b||c+1)){
    a=a+2;
    if(!b) c=c+1;
    x=c;
}
else{
    a=a+1;
    if(a+1) c=c+1;
    a=a+1;
}
```

Using Rule 3 and Rule 2 in the `then` part of `if(a+2&&(b||c+1))`, and Rules 3 and 1 on the `else` part yields:

```
if(a+2&&(b||c+1)){
    if (!b) c=c+1;
    x=c;
    a=a+2;
}
else{
    if(a+2) c=c+1;
    a=a+2;
}
```

Finally, factoring out common assignments using Axiom 2 yields:

```
if(a+2&&(b||c+1)){
   if (!b) c=c+1;
   x=c; }
else if(a+2) c=c+1;
a=a+2;
```

`if(`
`   a++,`
`   ++a&&(b||++c))`
`      x=c;`
	`if(a+2&&(b||c+1)){`
	`  if (!b) c=c+1;`
	`  x=c; }`
	`else if(a+2) c=c+1;`
	`a=a+2;`

Original	Side Effect Free Version

Figure 9: Side Effect Removal Example

Although this fragment is slightly longer than the original, it is potentially easier to understand and is also amenable to further analysis using other tools which operate solely upon side effect free programs.

For comparison, the original and the final, simplified, side-effect free version are presented in Figure 9.

5 Related and Future Work

Many authors have considered the problem of *detecting* the presence of side effects. To the authors knowledge, the present paper is the first to consider the problem of side effect *removal*. This section describes the relationship between previous work on side effect detection and the work reported here on side effect removal, indicating some directions of future work.

Side effects presented difficulties for work on Floyd/Hoare axiomatic characterisation of programming language semantics [15, 21, 22]. Using the axiomatic method, assertions effectively capture the set of possible states which may pertain at a particular program point. In the original formulation [15, 21], the assignment statement is treated as the paradigm of state change, with all expressions being considered to be side–effect free. Side effects complicate the exposition, which was initially attractive for its elegance and simplicity. There have been several attempts to capture the semantics of side effects in a simple and uncluttered way [3, 11, 25], but all involve additional and sometimes cumbersome, formal machinery.

By treating side effect removal as a source-level transformation, the side effect removal algorithm presented here could be considered to be an alternative characterisation of the semantics of side effects in the style of Landin [27]. In this approach, once the formal properties of the 'core language' are well defined, it is possible to define the semantics of a larger language by translation into the core. In this sense the constructs of the larger language are merely 'syntactic sugar' [27] for those in the core.

In the case of side effects, just such an approach is possible. The semantics of the core language (side-effect free `C--`) could be defined in a formal style (using the axiomatic method for example). The side-effecting programs of the superset language can be translated into longer programs which are side-effect free. Perhaps 'sugar' is a misnomer for side effects (often considered to be anything but sweet), but the essence of the approach remains the same.

The work presented here considered only the problem of removing side effects, and not the problem of detecting their presence. Interestingly, it appears that the problem of removing side effects may be easier than that of detecting them. The later problem is not computable, even in the simple intraprocedural case considered here.

Consider for example the code fragment:

```
x = f1(x) ;
if (x=f2(x)) s1 else s2
```

Suppose that both `f1` and `f2` are side effect free. The algorithm for post-placement presented here will transform this program fragment to

```
x = f1(x) ;
if (f2(x)) {x=f2(x);s1} else {x=f2(x);s2}
```

This transformed program is side-effect free. The original program is also side effect free if

$$\forall x \bullet \mathtt{f1}(x) = \mathtt{f2}(x)$$

However, this question is undecidable for arbitrary functions `f1` and `f2`. The problem of removing side effects may turn out to be computable however. This would be possible, because the problem of removing side effects is not necessarily reducible to the problem of detecting them. That is, the side effect removal transformation may transform side-effect free programs *as well as* side–effecting programs in an effort to remove side effects. However, it can guarantee that the result is side effect free, whereas determining whether or not a side effect is present is not generally decidable. The question of whether side effect removal is computable remains an open problem for future work.

The problem of determining whether or not a program contains a side effect is also more complex in the interprocedural paradigm. Work in this area such as that of Cooper and Kennedy in [8] and Spuler and Sajeev [31] attempts to produce a safe approximation to the set of side effects in a program. The approximation is conservatively safe in the sense that the algorithm will identify a superset of the side effects present, thereby guaranteeing to identify all side effects (but possibly including some 'false positives').

Side effect detection is useful in optimisation, for example Clausen [6] describes Cream, an optimizer for Java bytecode. The best side-effect analysis was found to improve efficiency by as much as 25%. Side-effect analysis is required to determine whether certain transformations are applicable. Such transformations include dead-code removal, constant propagation, common subexpression elimination. Removing side-effects by transformation presents an alternative to attempting to determine side effects in such situations. Side effect removal may provide an alternative route to side–effect based optimisation, by widening the range of applicable optimising transformation rules.

Other work on side effects has concerned characterisations of side effects and language design which takes account of side effects.

Banning side effects severely restricts a function's behaviour. However, some side effects, such as incrementing a count of the number of calls on a function, are important, yet relatively 'benign'. The problem is in finding a suitable characterization of 'benign'. Lamb [26] shows how to use trace specifications to prove that a side effect is benign.

Hall and O'Donnell [18] considered the problem of debugging in a side–effect free programming environment. They describe several debugging techniques for Daisy, a descendant of Lisp which is purely functional (i.e. prohibits all side effects) and uses lazy evaluation. Existing debugging techniques (like putting `print` statements in the suspected functions, or examining dumps) are inappropriate due to the absence of side effects, and the 'peculiar' evaluation order caused by lazy evaluation. In this work, the language considered is purely functional and so even the assignment statement is treated as a side effect. In the languages considered in the present paper, the source language is an imperative language and so the assignment statement is permitted, but side effects in expressions are to be removed.

It could be said that the algorithm for side-effect removal creates an 'almost functional' language, in which all state changes are expressed using the assignment statement. This would return the programming paradigm to that considered by the initial work on the Axiomatic Method, where axiomatic semantics is comparatively elegant and easy to define and use. This, perhaps, provides further additional anecdotal evidence that side–effects are hard to reason about and that their removal is worthwhile for comprehension.

It is widely accepted that side–effects impede comprehension. Despite this, side effects are widely used in programs. This may be the result of perceived or real improvements in execution speed which become possible using side effects. Previous work on optimisation using side effects, suggests that there is a trade off between speed of execution and speed and accuracy of comprehension. Fortunately Side Effect Removal Transformation allows the programmer to have two views of the program: A side–effecting version for execution and a side–effect free equivalent for comprehension.

6 Conclusion

It is widely thought that side effects hinder program comprehension. This paper introduces a new approach to the problem of side effects, called 'Side Effect Removal Transformation' (SERT).

The SERT approach consists of transforming a program which contains side effects into an equivalent program which is guaranteed to be side-effect free. This differs from previous approaches which have been focussed on the *detection* of side effects rather than upon their *removal*.

Three strategies for side effect removal are described: pre placement, post placement and temporary variable introduction. Pre- and post- placement differ in their choice of where to place the (side effect free) statements which replicate the behaviour of the side effects in the original program's expression. In post placement, these statements are placed after the side effect free version of the original expression, in pre placement they are placed before it. The third approach consists of introducing temporary variables.

It is shown that pre-placement may not always be possible. Unfortunately both post placement and temporary variable introduction can cause an increase in program size (through code copying or the introduction of temporary variable assignments). The paper illustrates the way in which post processing using standard transformation can be used to reduce the size of the program produced by one approach (post placement) which has been implemented in a simple SERT tool called Linsert.

References

[1] International standards origanisation: Programming languages — C. international standard, ISO/IEC 9899: 1990 (E), Dec. 1990.

[2] Bjørner, D., Ershov, A. P., and Jones, N. D. *Partial evaluation and mixed computation.* North–Holland, 1987.

[3] Boehm, H.-J. Side effects and aliasing can have simple axiomatic descriptions. *ACM Transactions on Programming Languages and Systems 7*, 4 (Oct. 1985), 637–655.

[4] Canfora, G., Cimitile, A., and De Lucia, A. Conditioned program slicing. In *Information and Software Technology Special Issue on Program Slicing*, M. Harman and K. Gallagher, Eds., vol. 40. Elsevier Science B. V., 1998, pp. 595–607.

[5] Cannon, L., Elliott, R., Kirchhoff, L., Miller, J., Milner, J., Mitze, R., Schan, E., Whittington, N., Spencer, H., Keppel, D., and Brader, M. Recommended C style and coding standards, 2000. http://www.cs.umd.edu/users/cml/cstyle/indhill-cstyle.html.

[6] Clausen, L. R. A java bytecode optimizer using side-effect analysis. *CONCURRENCY:PRACTICE AND EXPERIENCEPY 9*, 11 (Nov. 1997), 1031–1045.

[7] Coen-Porisini, A., and De Paoli, F. SYMBAD: A symbolic executor of sequential Ada programs. In *IFAC SAFECOMP'90* (London, 1990), pp. 105–111.

[8] Cooper, K. D., and Kennedy, K. Interprocedural side-effect analysis in linear time. In *Proceedings of the SIGPLAN '88 Conference on Programming Languages Design and Implementation* (1988), ACM, pp. 57–66.

[9] Coward, P. D. Symbolic execution systems - a review. *Software Engineering Journal 3*, 6 (Nov. 1988), 229–239.

[10] Coward, P. D. Symbolic execution and testing. *Information and Software Technology 33*, 1 (1991), 53–64.

[11] Cunningham, R. J., and Gilford, M. E. J. A note on the semantic definition of side effects. *Information Processing Letters 4*, 5 (Feb. 1976), 118–120.

[12] Danicic, S., Fox, C., Harman, M., and Hierons, R. M. ConSIT: A conditioned program slicer. In

IEEE International Conference on Software Maintenance (ICSM'00) (San Jose, California, USA, Oct. 2000), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 216–226.

[13] Danicic, S., Harman, M., and Sivagurunathan, Y. A parallel algorithm for static program slicing. *Information Processing Letters 56*, 6 (Dec. 1995), 307–313.

[14] De Lucia, A., Fasolino, A. R., and Munro, M. Understanding function behaviours through program slicing. In 4^{th} *IEEE Workshop on Program Comprehension* (Berlin, Germany, Mar. 1996), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 9–18.

[15] Floyd, R. W. Assigning meanings to programs. In *Mathematical Aspects of Computer Science*, J. T. Schwartz, Ed., vol. 19 of *Symposia in Applied Mathematics*. American Mathematical Society, Providence, RI, 1967, pp. 19–32.

[16] Futamura, Y., and Nogi, K. Generalized partial computation. In *IFIP TC2 Workshop on Partial Evaluation and Mixed Computation* (1987), D. Bjørner, A. P. Ershov, and N. D. Jones, Eds., North–Holland.

[17] Haahr, P. A programming style for java, Oct. 1999. http://www.webcom.com/˜haahr/essays/java-style/.

[18] Hall, C. V., and O'Donnell, J. T. Debugging in a side effect free programming environment. In *Proceedings of the ACM SIGPLAN '85 Symposium on Language Issues in Programming Environments* (New York, NY, 1985), ACM, pp. 60–68.

[19] Harman, M., and Danicic, S. Amorphous program slicing. In 5^{th} *IEEE International Workshop on Program Comprehesion (IWPC'97)* (Dearborn, Michigan, USA, May 1997), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 70–79.

[20] Harman, M., Sivagurunathan, Y., and Danicic, S. Analysis of dynamic memory access using amorphous slicing. In *IEEE International Conference on Software Maintenance (ICSM'98)* (Bethesda, Maryland, USA, Nov. 1998), IEEE Computer Society Press, Los Alamitos, California, USA, pp. 336–345.

[21] Hoare, C. A. R. An Axiomatic Basis of Computer Programming. *Communications of the ACM 12* (1969), 576–580.

[22] Hoare, C. A. R., and Wirth, N. An axiomatic definition of the programming language Pascal. *Acta Informatica 2*, 4 (Dec. 1973), 335–355.

[23] Kernighan, B. W., and Pike, R. *The practice of programming*. Addison-Wesley Longman, Reading, Massachusetts, 1999.

[24] Kernighan, B. W., and Ritchie, D. M. *The C programming language*. Prentice Hall, 1988. Second Edition, (ANSI C).

[25] Kowaltowski, T. Axiomatic approach to side effects and general jumps. *Acta Informatica 7*, 4 (Feb. 1977), 357–360.

[26] Lamb, D. A. Benign side effects. *Information Processing Letters 29*, 6 (Dec. 1988), 301–305.

[27] Landin, P. J. The next 700 programming languages. *Communications of the ACM 9*, 3 (Mar. 1966), 157–166.

[28] Mehlich, M., and Baxter, I. Mechanical tool support for high integrity software development. In *High Integrity Systems '97* (1997), IEEE Computer Society Press, Los Alamitos, California, USA.

[29] Meyer, B. *Object-oriented Software Construction*, second ed. Prentice Hall, New York, NY, 1997.

[30] Partsch, H. A. *The Specification and Transformation of Programs: A Formal Approach to Software Development*. Springer, 1990.

[31] Spuler, D. A., and Sajeev, A. S. M. Compiler detection of function call side effects. *Informatica 18*, 2 (1994), 219–227.

[32] Thomas, P. *Learning to program in C*, 2 ed. Plum Hall, Inc., 1989.

[33] Ward, M., Calliss, F. W., and Munro, M. The maintainer's assistant. In *Proceedings of the International Conference on Software Maintenance 1989* (1989), IEEE Computer Society Press, Los Alamitos, California, USA, p. 307.

[34] Weiser, M. Program slicing. *IEEE Transactions on Software Engineering 10*, 4 (1984), 352–357.

[35] Weyuker, E. J. *Program schemas with semantic restrictions*. PhD thesis, New Brunswick, New Jersey, 1977.