Compiler Detection of Function Call Side Effects

D.A. SPULER and A.S.M SAJEEV

Technical Report 94/01
January 1994

Compiler Detection of Function Call Side Effects

David A. Spuler *and* A. S. M. Sajeev
Department of Computer Science
James Cook University
Townsville, QLD 4811
Australia

Abstract

The determination of whether an operation in a procedural language such as Pascal causes side effects has important applications in static error detection, program optimization, program verification and other software tool issues. For simple operations, such as assignment, there is obviously a side effect. However, to detect whether a function call causes a side effect, the body of the function must be analyzed. A function call can produce a side effect in a number of ways, such as: modifying a global variable, modifying an internal static variable, modifying one of its pass-by-reference arguments, producing output, consuming input etc. This paper explains the algorithms and data structures used to determine if a function call causes a side effect.

1. Introduction

Static analysis of computer programs is used in a variety of applications. Static analysis techniques provide means to study the properties of a program without executing it. For example, an optimizing compiler can employ data flow analysis to remove redundant code (dead-code elimination), to replace expensive operations by cheaper ones (reduction in strength) etc. [1]. Static analysis is extensively used in proving programs correct [2]. Error checkers like Lint [8] also use static analysis techniques to detect bugs and obscurities in programs.

Side effects are used as a method of communication among program units. However, the method is not considered very elegant because of its adverse effect on clarity of programs. Besides, side effects can prevent a compiler from generating optimized code. For example, in the loop below:

```
while ( i < test(j) ) do
        i := i+1
```

an optimizing compiler would prefer to move the evaluation of the of the function call `test` to before the loop. (This optimization is called code motion.) This is possible only if it can be shown that `test` is free of side effects (and that `j` is not an alias for `i`).

This paper describes algorithms that can be used by an optimizing compiler to detect functions (and procedures) with side effects; it is also useful in error checkers to issue warnings on the problems side effects can create.

Algorithms have been proposed for side effect detection [3]. However, previous efforts were mostly on considering side effects through global variables and reference parameters. If optimizations such as in the example above are to succeed, other types of side effects must also be considered.

In the next section we define the term *side effect* and identify the different causes of side effects. In Section 3, we describe in detail the algorithms and data structures used to detect functions with side effects. In Section 4 we point out some of the limitations of static analysis techniques in detecting side effects. Section 5 gives the conclusions.

2. What is a Side Effect?

The term *Side effect* refers to the modification of the nonlocal environment [7]. Generally this happens when a function (or a procedure) modifies a global variable or arguments passed by reference parameters. But there are other ways in which the nonlocal environment can be modified.

We consider the following causes of side effects through a function call:

1) Performing I/O
2) Modifying global variables
3) Modifying local permanent variables (like `static` variables in C)
4) Modifying an argument passed by reference
5) Modifying a local variable, either automatic or `static`,
of a function higher up in the function call sequence (usually via a pointer).

All but item 3 obviously affect the nonlocal environment. We shall briefly describe these effects. Any function that performs I/O, whether it be console I/O or file I/O, is causing a side effect. Such functions do affect the environment outside the program. Failing to get the effect of I/O will obviously change the meaning of the program.

Another important way that a function can cause a side effect is by modifying a global variable that is used by some other function. Such a side effect can be caused by an explicit assignment to a global variable, or in less obvious ways such as calling a library routine to set a global clock (effectively a global variable).

A modification to an internal permanent variable (eg. a `static` variable in C) can also be a side effect if a later call to the same function uses the old value of the `static` variable. This is because a static variable, though declared within a function, retains its value after the the function terminates. Normally, the local environment of a function is destroyed at the termination of the function. Since static variables are not destroyed we can consider them as belonging to the nonlocal environment with the restriction that they are accessible only through the statements in the current function.

Modification of an argument passed by reference to a function is a form of side effect. In C, this is applicable to array variables only. However, languages like Pascal allows parameter passing by reference. Since pointers can set up *aliases*, modification of a variable through a pointer dereference can also cause a side effect. For example, the pointer could be pointing to a global variable.

3. Detection of Side Effects

To be certain that a function call produces no side effects, the following conditions are sufficient (but not necessary):

(1) The function does not perform any I/O
(2) No global variables are modified
(3) No local permanent variables are modified
(4) No pass-by-reference parameters are modified
(5) No modification is made to nonlocal/static variables via pointers
(6) Any function called also satisfies these conditions

The detection of side effects is done in two phases. In the first phase, each function in the program is considered separately (intraprocedural analysis) and all except condition 6 are checked. This will result in two lists: a list of functions which do not satisfy one or more of conditions 1 to 5, and therefore have the potential[1] to cause side-effects, and another list of functions which satisfies conditions 1 to 5. The second list will be side effect free only if condition 6 is satisfied; this is checked in Phase 2 (Interprocedural analysis).

The algorithms are introduced here in an incremental fashion. First we consider the simple case (ignoring permanent variables and pointers) and later in Sections 3.4 and 3.5, we modify the algorithm to consider the effect of pointers and permanent variables.

3.1. Intraprocedural Analysis

In the intraprocedural analysis, statements in each function are analysed separately. Information about library functions is used to identify functions that perform i/o. A function that has statements which make calls to the i/o functions is identified as having the i/o side-effect. A function can access nonlocal data objects through nonlocal variables (i.e., variables declared outside the function) and parameters passed by reference. The modification is detected by examining every assignment statement and input statement within the function body. If any of these variables appear in the left hand side of the assignment statement then the function is marked to produce a side effect. Note that in languages like C, short hand forms like i++ can be written to mean i = i+1; these forms are taken care of while scanning for assignments.

3.1.1. The Method The statements in the function are scanned and the information collected is stored in a table called the side-effect table (*se table*). The *se table* has a row for each function in the program. Columns are titled *Nonlocal Variables, Reference Parameters, I/O, Indirect Calls* and *Side Effect*. If a function is found by analysis to cause a side effect by modifying the nonlocal variables, reference parameters or through i/o, then the *Side Effect* entry for that function will have the value *true*. Moreover, if the side effect is caused by modifying one or more nonlocal variables, then the set of nonlocal variables which are modified in the function is given in the column titled *Nonlocal Variables*. The case for reference parameters is similar. For i/o, an input statement will be coded as R and any output statement will be coded as W. The column *Indirect Calls* is left null in the first phase; it is later used to record the set of side-effect causing functions which are called by the current function.

The algorithm to create the *se table* through intraprocedural analysis is given below.

[1]The analysis is flow insensitive. That is, if a function can cause side effects in any of its execution path, then it will be flagged to have the potential to cause side effects.

```
Initialize the entries in the side effect table to ∅ (null set).
For each function f do
    For each statement s in f do
        If s modifies reference parameters r1,..,rn of f then
            seTable[f, ref] := seTable[f, ref] ∪ [r1,..,rn];
        If s modifies nonlocal variables v1,..,vn of f then
            seTable[f, nonlocal] := seTable[f, nonlocal] ∪ [v1,..,vn];
        If s is an input statement
            seTable[f, io] := seTable[f, io] ∪ [R];
        If s is an output statement
            seTable[f, io] := seTable[f, io] ∪ [W]
    end;
end;
For each function f in the seTable do
    if an entry in that row is non null then
        seTable[f, side effect] :=  true
end;
```

Algorithm 1: Intraprocedural analysis to detect side effects

3.2. An Example

Consider the Pascal program given in Figure 1.

```
program demo;
var b : integer;

function f3 : integer;
var x : integer;
begin
      x := f1 (x);
      x := f2
end;

function f2 : integer;
var x : integer;
begin
      x  := 20;
      f2 := x
end;

function f1 (var a : integer) : integer;
begin
      read (a);
      a := f2;
      a := f3
end;

begin
      write (f1(b));
end.
```

Figure 1. An example program for side-effect detection

The *se table* for this program is shown below.

Function	Nonlocal	Ref	I/O	Indirect	Side effect
f1	∅	[a]	[R]	∅	true
f2	∅	∅	∅	∅	?
f3	∅	∅	∅	∅	?

Table 1: *Se Table* for the program in Figure 1.

The table shows that *f1* has side effects through the reference parameter a as well as through i/o. No other function is determined to cause side effects at this stage.

3.3. Interprocedural analysis

The next phase is to do the interprocedural analysis to detect side effects created by calls made to other functions. For this, a *called-by graph* for the program is constructed. A called-by graph is similar to a *call graph* [1]. A call graph is a directed graph with a node for each function and an edge from the node for function A to the node for function B, if and only if, the body of function A contains a call to function B. In a called-by graph, the direction of the edges are reversed. Figure 2 shows the call graph and called-by graph of the program in Figure 1.

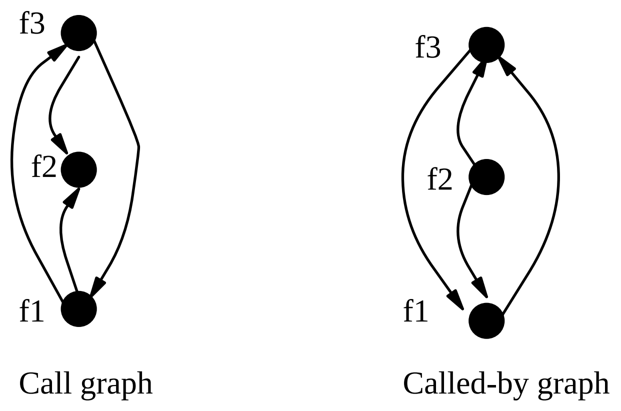

Figure 2. Call and Called-by graphs for the example program

The partially complete *se table* from phase 1 and the called-by graph can be used to fill the remaining entries in the *se table*. This is done using Algorithm 2. Intuitively, the algorithm starts with a function *f*, which is already found to cause side-effect, and denotes all functions that call *f* as causing a side effect. This is repeated for all functions which cause side-effects. If no function is found to cause a side effect at the end of phase 1, there is no need to do phase 2 because all functions are side-effect free in the sense of Section 2.

```
Unmark all rows in the se table;
While unmarked rows with true in the side-effect column exist in the table do
      Let f be the function name of an unmarked entry with true in the side-effect column;
      Mark the row corresponding to f;
      For every function g to which there is an edge from f in the called-by graph do
            Add f to the set of functions in the Indirect entry of g;
            Enter true in the side-effect entry of g;
      end;
end;
Replace question marks with false in the side-effect entry for
      all functions with question marks.
```

Algorithm 2. An algorithm to find side-effects caused by function calls

Applying this algorithm to Table 1, gives the table:

Function	Nonlocal	Ref	I/O	Indirect	Side effect
f1	∅	[a]	[R]	[f3]	true
f2	∅	∅	∅	∅	false
f3	∅	∅	∅	[f1]	true

Table 2. The *se* table after the interprocedural analysis.

All the functions found to be side-effect free can be guaranteed to be side-effect free in the sense of Section 2. However, as in the case of any static analysis technique, the converse need not hold. For example, our analysis will show that function *f* in the following program is likely to cause side effects, but whether it will actually do so can be decided only at run time; it will depend on the input value of the variable `x`.

```
program demo0;
var x : integer;

function f (a:integer) : integer;
begin
      if a = 0 then
            x := 10;
      f := 20;
end;

begin
      read(x);
      write (f(x))
end.
```

In some cases, more complicated analysis can be performed to check whether some of the functions which are flagged to have side effects in the initial analysis indeed have a side effect. Consider the program below.

```
program demo1;
var x : integer;

function f1 : integer;
var y : integer;
      function f2 : integer;
      begin
            y := 10;
            f2 := 10;
      end;
begin
      y := 20;
      y := f2
end;

begin
      x := f1
end.
```

Our static analysis will flag both `f1` and `f2` as having side effects. The *se table* created after both the intra and interprocedural analyses is:

Function	Nonlocal	Ref	I/O	Indirect	Side effect
f1	∅	∅	∅	[f2]	true
f2	[y]	∅	∅	∅	true

Table 3. The *se* table for program demo1.

Function `f2` causes side effect because it modifies a non local variable (`y`). Function `f1` causes side effect because it calls another function (`f2`) which has side effect. However, a look at `f1` reveals that `f1` does not have side effects because, `f2` modifies a local variable of `f1`. The intrafunctional algorithm can be modified as in Algorithm 3 to incorporate this.

This algorithm uses two functions. Function *nl(g, nonlocal_set)* returns a set of variables which are elements of *nonlocal_set* but are also nonlocal to the function *g*. The function *ap(f, g, parameter_set)* returns a set of actual parameters of the function *g* corresponding to the formal parameters in the *parameter_set* for the function *f*.

```
Unmark all rows in the se table;
While unmarked rows with the side-effect entry true exist in the se table do
    Let f be the function name of an unmarked entry with the side-effect entry true;
    Mark the row corresponding to f;
    For every function g to which there is an edge from f in the called-by graph do
        nlset := seTable[g, Nonlocal];
        rpset := seTable[g, Ref];
        seTable[g, Nonlocal] := seTable[g, Nonlocal] $\cup$ nl(g, seTable[f, Nonlocal]);
        actuals := ap(f, g, seTable[f, Ref]);
        seTable[g, Nonlocal] := seTable[g, Nonlocal] $\cup$ (actuals $\cap$ nonlocal variables of g)
        seTable[g, Ref] := seTable[g, Ref] $\cup$ (actuals $\cap$ reference parameters of g)
        If (nlset $\cap$ seTable[g, Nonlocal] $\neq \emptyset$ ) or
            (rpset $\cap$ seTable[g, Ref] $\neq \emptyset$) then
            seTable[g, Indirect] := seTable[g, Indirect] $\cup$ [f];
            seTable[g, side-effect] := true;
        end;
    end;
end;
Replace question marks with false in the side-effect entry for
    all functions with question marks.
```

Algorithm 3. Modified algorithm to find side-effects caused by function calls

Intuitively, if a side-effect causing function *f* is called by *g*, then *g* will cause side-effect only if non-local variables or reference parameters modified in *f* are also nonlocal variables or reference parameters of *g*. (Note that a nonlocal variable of *f* could be a reference parameter of *g* or a reference parameter of *f* could be a nonlocal variable of *g*.)

After applying the modified algorithm to the `demo1` program, we get the *se table* as:

Function	Nonlocal	Ref	I/O	Indirect	Side effect
f1	∅	∅	∅	∅	false
f2	[y]	∅	∅	∅	true

Table 4. The *se table* for the `demo1` program after applying Algorithm 3.

The table shows that only function `f2` has the potential to cause side effect.

3.4. Detecting Side Effects Due to Pointer Dereferences

Nonlocal pointer variables can cause side effects. While other nonlocal variables cause side effects by direct modification, pointer variables can cause side effects by dereferencing. However, a number of side effects can occur in a non-obvious fashion by modification of a value via a pointer:

- Pointer variables passed as *value* parameters can also cause side effects by dereferencing.

- Nonlocal pointers can be assigned to local pointer variables; the latter can then cause side effects by dereferencing.

In the earlier analysis we did not consider pointer dereferencing. To do this analysis, the *se table* is added one more column named *Pointers*. An entry in *Pointers* column shows the set of pointers that can cause a side effect through dereferencing. For every nonlocal pointer used in a function, a set of variables (local or nonlocal) which has the same pointer value is maintained. Any modification by dereferencing any of the pointers in this set is flagged to cause a side effect during the intraprocedural analysis (i.e., phase 1). The modification to Algorithm 1 is shown below. (The algorithm must be modified in the initialization, iteration and the post-iteration parts.)

```
Initialization:
For every pointer type parameter or nonlocal pointer p appearing in the function f
      alias[p] :=  [p];  {Comment: alias is an associative array with
                                    elements of set type}

Iteration:
For every statement s in the function f do
      If any element of alias[p] is assigned to q then
            Remove q from all alias sets;
            alias[p] := alias[p] ∪ [q];
      For any p, if s modifies a data object by dereferencing through an element of alias[p] then
            seTable[f, Pointers] := seTable[f, Pointers] ∪ [p];
end;

Post iteration:
For every function f do
      if seTable[f, Pointers] ≠ ∅ then
            seTable[f, side-effect] := true
end;
```

As an example of the application of the algorithm, consider the analysis of the the procedure `f` in the following program.

```
program demo2;
type R = record
      value : integer
      end;
     pointer = ˆR;

var a, b : pointer;

procedure f (c : pointer);
var d : pointer;
begin
      d := c;
      d := a;
      dˆ.value := 25
end;

begin
      aˆ.value := 15;
      bˆ.value := 20;
      f(b)
end.
```

Figure 1. An example program for side-effect detection

Initially,

$$alias[a] = [a];\ \ alias[c] = [c];\ \ seTable[f, Pointers] = \emptyset$$

After analysing the first assignment in the procedure f, the set values become

$$alias[a] = [a];\ \ alias[c] = [c, d];\ \ seTable[f, Pointers] = \emptyset$$

After the second assignment, they change to:

$$alias[a] = [a, d];\ \ alias[c] = [c];\ \ seTable[f, Pointers] = \emptyset$$

After the third assignment, we have:

$$alias[a] = [a, d];\ \ alias[c] = [c];\ \ seTable[f, Pointers] = [a]$$

Therefore, at the termination of the analysis,

$$seTable[f, side\text{-}effect] = true$$

3.5. Detecting Side Effect Due to Permanent Local Variables

Some programming languages allow permanent local variables. *Static* variables in C and *own* variables in Algol are examples. Their values are retained from one function call to another, thus causing a side effect. Detecting this type of side effect is not a problem. The *se table* needs one more column for permanent local variables. Any modification to such variables will change the entries in that column during the intraprocedural analysis, just as in the case of other columns in the *se table*.

4. Limitations of the Method

There are situations where a function has the potential to cause a side effect, and yet a particular call to that function will not cause the side effect to occur. For example, the arguments to a function call may mean that the execution path containing the statements causing the side effect is not executed, as below:

```
function F (i : integer) : integer;
begin
      if i > 0 then
              writeln ('Hello');
      F := 0
end;

F(0);
```

The call to `F()` does not produce output, although `F()` must still be classed as an output-producing side effect function. The general problem of determining if a particular call to a function will invoke a particular execution path is very difficult. In fact, it is non-computable in general (because being able to solve it would provide a solution to the halting problem), although many special cases are solvable, such as the simple one above. Analysis of the function body and possibly the entire program would be able to identify function calls that do or do not cause a side effect. However, the advanced methods necessary to detect which paths will be executed at run-time are beyond the scope of this paper. Thus, a limitation of our method is that, while it can identify function calls that definitely cause no side effect, any function call that is considered to cause a side effect may not actually do so at run-time. More extensive global analysis would be necessary to prove that a function call will cause a particular path inside the function to be executed. Fortunately, this limitation is not too damaging, since error detection of side effect related errors will, at worst, produce a spurious warning message, and program optimizations tend to rely on proving that a function call does *not* cause a side effect, rather than proving that it does.

5. Discussion

Our interest in function calls that cause side effects arose during the development of a static checker for C. It was important to identify side effect operations so as to warn about expression statements that produced no side effects (null effect statements) and order of evaluation problems (where the order of occurrence of side effects was ambiguous). Lint also checks for such errors [8]. At the time the problem was not considered too important and it was decided to consider function calls as side effects for the first test, and not as side effects for the second test — both choices aim to reduce spurious warnings about correct code.

Detecting that a function call does not produce any side effects is important in compiler error detection to warn about "null effect" function call statements. Static detection of programming errors is not the only area where it is useful to know if a function call causes a side effect. It is also useful in program verification (side effects do make program verification difficult) and more importantly for compiler optimization (to remove redundant function calls, for example).

The algorithms developed analyse functions and detect those functions which do not cause side effects due to any of the reasons identified in Section 2. For those functions with potential side effects, the algorithms also identify the causes of those side effects. We are not aware of any side effect tools in actual use. Banning [3], Burke [4], and Cooper and Kennedy [5] have proposed algorithms to detect side effects but, their works were based only on side effects caused by global variables and parameter passing by reference.

Even when a function has a side effect, it may not adversely affect the calling function. For example, if the value of the global variable is not used at any time after the function returns, but instead the old value is overwritten, the change to the value of the global variable has had no effect. An example of this would be a function that uses a global variable, or even a global data structure, to perform some temporary calculations (e.g. a treesort routine uses a global binary tree as a temporary data structure to sort an array, and then destroys the binary tree). However, this does not mean that the called function does not cause a side effect— it simply means that the calling function is not affected by it.

Similarly a local static variable can only be accessed within the current function, and hence if it does not use the old value of the `static` variable, a modification to that variable does not affect the function. (Interestingly, any local `static` variable that is set before used inside a function need not be declared as "static", and hence any such instances probably deserve a compiler warning.) This situation can be detected by algorithms to determine if the variable is *set before it is used*. A number of algorithms have already been developed for the similar problem of variables *used before set* and these algorithms could easily be modified: Fosdick and Osterweil [6] discuss the algorithms in general and an example of their implementation is the Omega checker [9].

The analysis is more difficult for global variables because there is no scope restriction that only one function can use the variable (as there is for local `static` variables). It becomes necessary to perform global analysis to determine which functions actually modify or use the global variable. It is possible to generalize the algorithm for detection of set-before-used local `static` variables to global variables. A side effect due to the modification to a global variable within a function *F* is harmless if:

1) The global variable is set-before-used in the function *F*
2) Any other function setting or accessing this global variable must be called via *F*.

The second condition guarantees that any call to a function using the global variable must first pass through *F*, which sets the global variable and ignores any previous value. This condition can be detected by examination of the call graph. The lower level functions using the global variables form one or more subgraphs of the call graph, where the only entry points to this sub-graph are directed edges originating at the node for *F*.

Procedural parameters (i.e., procedures passed as parameters to other procedures) are not included in our algorithms. In our experience, procedural parameters are not very common in programs, and the utility of the algorithms are not lost by leaving them out.

References

[1] A.V. Aho, R Sethi, and J.D. Ullman, *Compilers, Principles, Techniques and Tools*, Addison Wesley, Reading, MA, 1986

[2] K.R. Apt, "A Static Analysis of CSP Programs", Logics of Programs: Proceedings 1983, Lecture Notes in CS, No. 164, pp. 1-17, Springer Verlag, New York, 1983

[3] J.P. Banning, "An Efficient Way to Find the Side Effects of Procedure Calls and the Aliases of Variables", 6th Annual ACM ACM Symposium on Principles of Programming Languages, Jan., 1979.

[4] M. Burke, "An Interval-Based Approach to Exhaustive and Incremental Interprocedural Data-Flow Analysis", ACM Transactions on Programming Languages and Systems, Vol. 12, No. 3, pp. 341-395, July, 1990.

[5] K.D. Cooper, K. Kennedy, "Interprocedural Side-Effect Analysis in Linear Time", ACM SIGPLAN Notices, Vol. 23, No. 7, pp. 57-66, July, 1988.

[6] L.D. Fosdick, L.J. Osterweil, "Data Flow Analysis in Software Reliability", ACM Computing Surveys, Vol. 8, No. 3, pp. 305-330, 1976.

[7] C. Ghezzi, M. Jazayeri, *Programming Language Concepts*, John Wiley & Sons, New York, NY, 1987.

[8] S.C. Johnson, "Lint, A C Program Checker", Technical Report, AT&T Bell Labs, Murray Hill, New Jersey, July, 1978

[9] C. Wilson, L.J. Osterweil, "Omega—A Data Flow Analysis Tool for the C Programming Language", IEEE Trans. on Software Eng., Vol. 11, No. 9, pp. 832-838, 1985.