McGill University
School of Computer Science
ACAPS Laboratory

Advanced Compilers, Architectures and Parallel Systems

Extended SSA Numbering: Introducing SSA Properties to Languages with Multi-Level Pointers

Christopher Lapkowski
Laurie J. Hendren

ACAPS Technical Memo 102
April 19, 1996

Abstract

Static Single Assignment (SSA) intermediate representations have become quite popular in compiler development. One advantage of the SSA form is that each use of a variable corresponds to exactly one definition, and thus two references of the same SSA variable must denote the same value. To date, most SSA forms concentrate on scalar variables, and it is difficult to extend these intermediate representations to languages with multi-level pointers like C.

Translating a program into SSA form requires two steps. The first is inserting ϕ-nodes that join values from different flow of control paths, and the other renames variables such that each name has exactly one definition. When designing SSA forms to handle pointers, the insertion of ϕ-nodes is quite difficult. Thus, we propose a method that concentrates on the second step, in that we associate an SSA number to each variable reference, including references via pointers. Even without ϕ-nodes, we retain some of the properties of SSA such as "one value per name" when the SSA number is used as part of the name.

In addition to a primary SSA number for scalar variables we introduce a secondary SSA number for pointers. Given an indirect reference of the form `*p`, the primary number refers to the contents of `p` and the secondary number refers to the contents of `*p`. Using both primary and secondary SSA numbers we can tell when two indirect references denote the same value, as well as derive other SSA properties for indirect references.

In this paper we show how we have implemented Extended SSA Numbering in the McCAT C Compiler, and demonstrate its usefulness by strengthening reaching definitions analysis and developing a new optimizing transformation.

Contents

1 Introduction **1**
1.1 Difficulties with pointers 2

2 Compiler Framework **3**

3 Definition and Implementation of Extended SSA Numbering **7**
3.1 Implementation 8
3.2 Points of attention 9

4 Properties of Extended SSA Numbering **13**
4.1 Factored use-def and def-use chains 14
4.2 Definition point properties 14

5 Applications of Extended SSA Numbering **16**
5.1 Symbolic analyses 17
5.2 Induction variables 17
5.3 Recognition of `ref` parameters 17
5.4 Strengthening reaching definitions 17
5.5 Reduction of indirect references 18

6 Related Work **19**

7 Conclusions **20**

List of Figures

1 Example of SSA form vs. SSA Numbering 2
2 Difficulties with pointer for SSA form shown in original program, SSA form and SSA Numbered program 3
3 Structure of McCAT compiler 4
4 Example of SIMPLE intermediate representation 5
5 Data accessible via a pointer. 7
6 Sample program Extended SSA Numbers. 8
7 Partial algorithm for SSA Numbering analysis 10
8 Effects of `break` statements in while loops 11
9 Effects of `break` statements in do-while loops 12
10 Example program showing SSA numbers for structures using both methods . . . 13
11 Example of reaching definitions for pointer references. 18
12 Two transformations to reduce indirect references 19

List of Tables

1 Points-To relationships for SIMPLE program in Figure 4(b) 6
2 Read/Write Sets for SIMPLE-C program in Figure 4(b) 6
3 Description of benchmarks used to measure space requirements of DU, UD chains. 15
4 Average sizes of Defs and Uses sets stored at each variable reference. 16
5 List of benchmarks to which we applied our indirect reference reduction. 19
6 Results of applying our indirect reference reduction transformation. 20

1 Introduction

In modern compiler technology the role of analyses and intermediate representations has become very important. Powerful analyses facilitate effective optimizing transformations and well-designed intermediate representations ease the design and implementation of these analyses. Static Single Assignment (SSA) form is one such intermediate representation. A program in SSA form has some key properties that make this intermediate representation favorable. Each variable has only one static definition point and can therefore be used to provide factored use-def and def-use chains. The SSA form and its properties have been widely discussed. [12, 3, 4, 1, 11, 2]

Although, SSA form has become very popular in FORTRAN compilers, it has not enjoyed such popularity among languages with multi-level pointers such as C. We propose to introduce the properties of SSA form into compilers for languages with multi-level pointers such as C, using a new dataflow analysis which we call *SSA Numbering*.

To motivate the introduction of SSA Numbering instead of SSA form let's consider an example and the process of transforming a program into SSA form. Putting a program into SSA form requires two basic steps. The first, inserts ϕ-nodes that join two possible values from different flow of control paths, and the other renames variables such that each name has exactly one definition. It is the first step the causes most difficulty in the presence of pointers. We propose to concentrate on the second step, renaming variables by appending numbers.

Consider the code fragment example in Figure 1. Figure 1(a) contains the original program, Figure 1(b) contains the program in SSA form and Figure 1(c) contains the program with SSA Numbers. Notice several key differences between the original program and the program in SSA form. First, all variables are renamed and some variables, such as `a` that had several static definition points became several variables, (`a_1`, `a_2`, `a_3`, `a_4`) one for each definition point. Note that separate memory locations are associated with these variables. Second, a new statement is introduced, labeled *S2*. This statement, known as ϕ-node or join node, ensures that data correctly flows when two flow of control paths join. Depending from which branch flow of control came to the ϕ-node the corresponding value is returned.

Now let us compare SSA form to our propsed SSA Numbering in Figure 1(c). Notice that the SSA Numbered program does not add new variables. In the example, there is only one location for the variable `a` where as in SSA form there were four. Also, the SSA Numbered program does not have ϕ-nodes. With the exception of subscripts on all variables, the program looks exactly like the original. The subscripts are the same numbers as the ones appended to variables in the SSA form program (Figure 1(b)). For example, in statement *S3* and its counterpart *S1* we see that the left hand sides are `a_3` and $\mathtt{a}_3$. Similarly, on the right hand side we have `a_1` vs. $\mathtt{a}_1$ and `b_1` vs. $\mathtt{b}_1$. These subscript numbers are not part of the variable, instead just like information from a program analysis which is stored at statement points, the SSA numbers are stored at each variable reference to be used by compiler components that wish to access them.

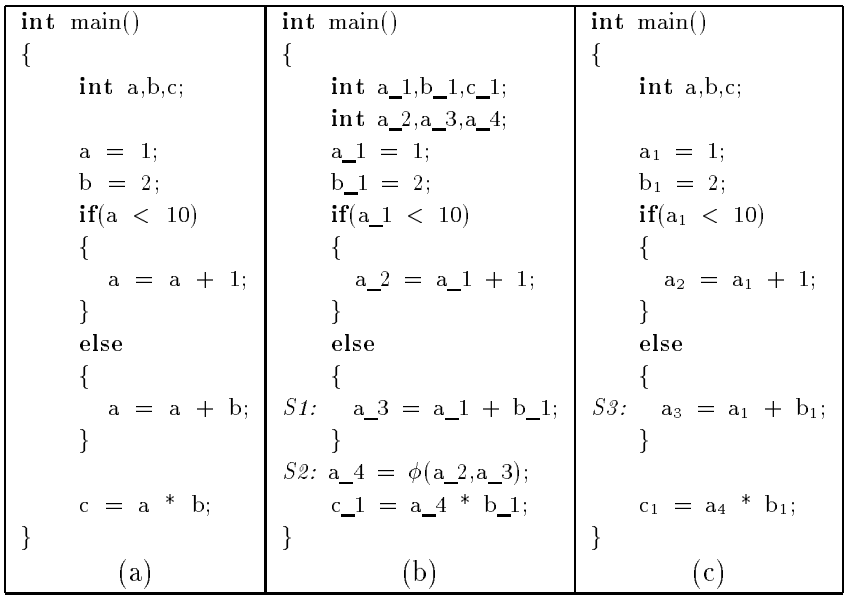

Figure 1: Example of SSA form vs. SSA Numbering

1.1 Difficulties with pointers

Seeing the differences between SSA form and SSA Numbering it is clear that there is fewer properties in the SSA Numbering intermediate representation then in SSA form. SSA Numbering still has the property that two variable references with the same SSA number must denote the same value. However, unlike SSA form, SSA numbers do not provide single definition point for each use. Why do we then propose SSA Numbering? The answer lies in three major problems which are easily solved in SSA Numbering, two of which are illustrated in Figure 2.

First consider the problem of taking the address of variables. This problem is illustrated in parts (a), (b), and (c) of Figure 2. Look at statement *S4* which takes the address of the variable `a`. In SSA form the variable `a` is represented by several memory locations, one for each static definition. Which location is being asked for? In the corresponding statement in the SSA form, *S5*, do we take address of `a_1`, `a_2` or even maybe something else? In the SSA Numbered program, statement *S6* does not have this problem because the variable `a` remains represented by only a single location.

Next consider the problem of an assignment via a pointer, illustrated in parts (d), (e), and (f) of Figure 2. The original statement is labeled *S7*. For the purpose of the example let us assume that the pointer `p` can point to either `a` or `b`. As we attempt to put the program segment into SSA form we see that statement *S8* has an implicit write to either `a` or `b`. We don't know which until run time. For the program to be in SSA form it is necessary to somehow expose this implicit write. Cytron and Gershbein [5] have done some work to correctly model this such that statement *S9* uses the correct variables with the correct values; however, as it can be seen

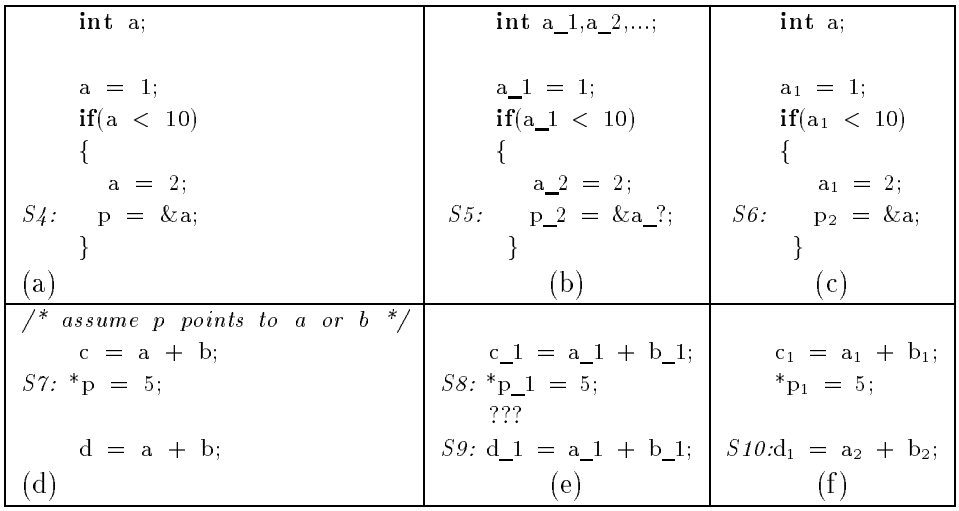

Figure 2: Difficulties with pointer for SSA form shown in original program, SSA form and SSA Numbered program

in Figure 2(f), such a construct is easily modeled in SSA Numbering. The SSA number of `a` and `b` after statement *S10* are different to reflect that they have a new possible static definition point.

The third problem is the lack of information about the value represented by `*p`. For two scalar references with the same name we are able to say that they share the same static definition; however, for two references of `*p` we can only say that they share the same static definition of `p` (ie. the address `p` holds), and nothing about the value returned by `*p`. Extended SSA Numbering will solve this.

The remainder of this paper is divided in the following way. Section 2 describes the compiler in which we implemented Extended SSA Numbering. Section 3 presents the Extended SSA Numbering algorithm. Section 4 defines the properties of Extended SSA Numbering as compared to the properties of SSA form. Section 5 describes some of the uses of Extended SSA Numbering. Section 6 relates our work to that done by others and section 7 presents our conclusions.

2 Compiler Framework

We have implemented Extended SSA Numbering in McCAT parallelizing/optimizing compiler which provides us with many analyses and an intermediate representation called SIMPLE[9]. The structure of McCAT is illustrated in Figure 3. McCAT accepts standard C programs in one or more files. For the purpose of interprocedural analyses, all the input C files are first symbolically linked into one, parsed and simplified to a simplified C called SIMPLE-C. High

level transformation and analyses are then performed at the SIMPLE level which is shown in Figure 3 as a box listing several of the analyses. Extended SSA Numbering is one of these analyses. Once finished high level analyses and transformations McCAT continues into lower level analyses and code generation.

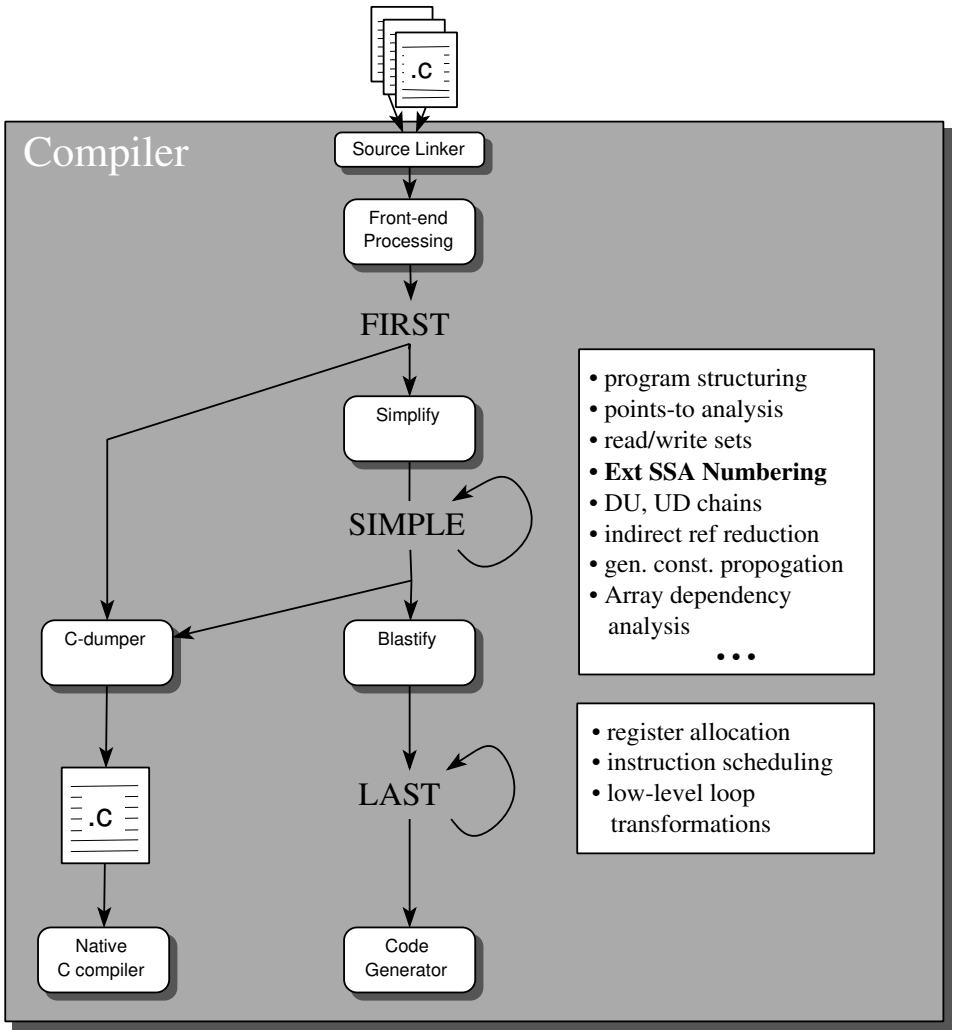

Figure 3: Structure of McCAT compiler

The SIMPLE intermediate representation is an Abstract Syntax Tree that represents a simplified C. Some key properties of SIMPLE are:

- Complex expressions are simplified to a series of assignments to temporaries reducing all assignment statements to at most 3 address form.
- All multi-level indirect references are simplified into several single-level indirect references.
- *Program structuring* removes all unstructured control flow branches [7, 8].

Program structuring and simplifying multi-level indirections are crucial to Extended SSA Numbering. Figure 4 shows an example of a program and its corresponding SIMPLE intermediate representation.

```
int g,h,**p,*q,*r;
int main()
{


  h = 2;
  q = &h;
  r = &g;
  p = &r;
  if(h < 10)
  {

    g = (h+2)*5;
    p = &q;
  }



  h = 2*g + **p;
}
```

(a)

```
    int g,h,**p,*q,*r;
S11:int main()
    {
      int temp_1,temp_0;
      int *temp_3,temp_2;
S12:  h = 2;
S13:  q = &h;
S14:  r = &g;
S15:  p = &r;
S16:  if(h < 10)
      {
S17:    temp_0 = h + 2;
S18:    g = temp_0 * 5;
S19:    p = &q;
      }
S20:  temp_1 = 2 * g;
S21:  temp_3 = *p;
S22:  temp_2 = *temp_3;
S23:  h = temp_1+temp_2;
    }
```

(b)

Figure 4: Example of SIMPLE intermediate representation

In addition to the SIMPLE intermediate representation Extended SSA Numbering requires two inter-procedural analyses:

- Points-to (similar to Alias analysis) analysis [6], that identifies for each point in the program, which named locations a pointer points to. Table 1 shows points-to relationships for each program point for the SIMPLE-C program in Figure 4(b). Each points-to relationship is shown as a pair (`p`,`v`) of pointer `p` and variable `v` to indicate that `p` can point to `v`. For example, at statement *S13* we introduce a new relationship namely that `q` points to `h`. This relationship shows up in the table starting at statement *S14*. Also, possible points to relationships are marked with a question mark following the pair.
- Read/Write sets (also known as Mod/Ref sets). This analysis summarizes at each statement which variables are read and written by this statement. This information is propagated in a hierarchical fashion such that compound statements that contain blocks of statements (eg. loops) describe which variables are read and which variables are written by the whole compound statement. Table 2 shows the Read and Write sets that are associated with each statement for the SIMPLE-C program in Figure 4(b). Notice that the

Stmt	Points-To Relations
S11	
S12	
S13	
S14	(q,h)
S15	(q,h) (r,g)
S16	(q,h) (r,g) (p,r)
S17	(q,h) (r,g) (p,r)
S18	(q,h) (r,g) (p,r)
S19	(q,h) (r,g) (p,q)
S20	(q,h) (r,g) (p,r)? (p,q)?
S21	(q,h) (r,g) (p,r)? (p,q)?
S22	(q,h) (r,g) (p,r)? (p,q)?
	(temp_3,g)? (temp_3,h)?
S23	(q,h) (r,g) (p,r)? (p,q)?
	(temp_3,g)? (temp_3,h)?

Table 1: Points-To relationships for SIMPLE program in Figure 4(b)

Read set for the `if` statement (*S16*) contains all the variables read by the condition and the body of the `if` statement.

Stmt	Read Set	Write Set
S11	h, g, p, q, r	h, g, p, q, r
S12		h
S13		q
S14		r
S15		p
S16	h, temp_0	temp_0, g, p
S17	h	temp_0
S18	temp_0	
S19		p
S20	g	temp_1
S21	p, q, r	temp_3
S22	temp_3, g, h	temp_2
S23	temp_1, temp_2	h

Table 2: Read/Write Sets for SIMPLE-C program in Figure 4(b)

3 Definition and Implementation of Extended SSA Numbering

We now describe the Extended SSA Numbering analysis by first defining the basic idea behind Extended SSA Numbering and then presenting our algorithm for computing it.

The goal of SSA Numbering analysis is to:

> Associate with each variable reference a number such that if the numbers were used as a renaming method the program would be in SSA form with all the ϕ-nodes removed.

Note that the major differences in generating SSA Numbering and SSA form are

- No ϕ-nodes are inserted for SSA Numbering where as in SSA form ϕ-nodes are a crucial part of the intermediate representation.
- When performing the renaming, in SSA Numbering, numbers are associated with variable references but no variables are actually renamed or created.

Let's take a closer look at what pointer variables represent and what data can be reached via a pointer. For example if we have a pointer `p` we can use it directly (`q = p;`) or we can dereference it (`x = *p;`). Each method gives a different piece of data. Figure 5 shows how we commonly draw pointers. It points out that there are two data accessible via the pointer, `p` and `*p`.

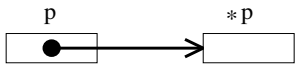

Figure 5: Data accessible via a pointer.

Seeing that pointers represent two data, modeling their behavior with one number will not be sufficient. We therefore introduce a second SSA number for pointer variables. We will call the two numbers *primary* and *secondary* SSA numbers. Recall from section 2 that one of the properties of SIMPLE is that multi-level pointer references are simplified to several single-level dereferences. Thanks to this simplification we will not need any more SSA numbers to handle multi-level pointers.

We use the primary SSA number to represent the address the pointer holds (data labeled `p` in Figure 5) and the secondary SSA number is associated with the data the pointer points to (data labeled `*p` in Figure 5). Therefore, if we define the value of the pointer we generate a new primary SSA number, and if we encounter an assignment via `*p` or via a variable `v` that `p` points to we generate a new secondary SSA number for `p`.

For example of how secondary SSA numbers work see Figure 6. In this example the statements of the form `x = *p + n;` (statements *S24, S25, S27, S28, S30, S31*) are used to let us see the primary and secondary numbers of `p`. Take for example statements *S25* and *S27*. Both of these statements reference `*p`. The primary numbers in both references of `*p` are the same because they both have the same static definition for `p`. On the other hand because there is a write to `n` at statement *S26* and `p` points to `n` the use of `*p` at statement *S27* has a different static definition point from that of statement *S25* thus producing a new secondary SSA number. Statement *S29* has a similar affect on references of `*p` in statements *S28* and *S30*. In addition to assignment statements we must also perform correct merging of control flow paths. Statement *S31* shows the new primary and secondary SSA numbers for `p` after a merge in control flow paths. We can imagine that there exists a ϕ-node just after the `if` statement which would look like $*p_{1,4} = \phi(*p_{1,2}, *p_{1,3})$.

```
int main()
{
      int a,x,n,*p;
      p1,1 = &n;
S24:  a1 = *p1,1 + n1;
      if (a1 <= 9)
      {
S25:    x1 = *p1,1 + n1;
S26:    n2 = x1;
S27:    x2 = *p1,2 + n2;
      }
      else
      {
        x3 = n1;
S28:    x4 = *p1,1 + n1;
S29:    *p1,3 = x4;
S30:    x5 = *p1,3 + n4;
      }
S31:  x6 = *p1,4 + n5;
}
```

Figure 6: Sample program Extended SSA Numbers.

3.1 Implementation

With the goal of Extended SSA Numbering analysis in mind we now describe our implementation of this analysis. To correctly SSA number all variable references we must know where ϕ-nodes would have to be inserted if we were putting the input program into SSA form. As noted in section 2, McCAT has a restructuring phase which guarantees the input program will be well structured (ie. no `goto`'s). With this in mind we refer the reader to the paper by Brandis and Mössenböck [2] for a description of the placement of ϕ-nodes in structured programs

for SSA form.

Figure 7 shows the steps that are taken for handling some of the key types of assignment statements, `if` statement, `for` loops, `while` loop, `do-while` loops, and function calls.

To assign SSA Numbers to variable references we maintain a look-up table containing the pairs (`v`, `n`) where `v` is a variable and `n` is the current SSA Number assigned to the variable `v`. The table is indexed by `v`. Each time an assignment to a variable `v` is encountered or a ϕ-node for `v` is expected, the pair (`v`, `n`) is replaced by (`v`, `n'`) where `n'` is an SSA number that has never appeared with the variable `v`. Selecting `n'` is easily accomplished by having a separate master table and each time a new number is needed for `v`, its SSA number is incremented by one in the master table. The function `new_SSA(v,t)` generates a new primary SSA number for the variable `v` and records that new number in the table `t`. For pointer variables we manage secondary numbers similarly using the function `new_secondary_SSA(v,t)`. We use the function `Store_SSA(v,t,s)` to look up the primary and secondary numbers of `v` in table `t` and store them in the reference of `v` at the statement `s`.

The algorithm makes use of a procedure `update_written()` defined in Figure 7. This procedure handles the generation of new SSA numbers for those variables that need it. For example, a simple assignment into `a` will increment the primary SSA number of `a` (`a` is in the write set) and any secondary SSA numbers of pointers that can point to `a`. For an assignment into `*a`, the write set contains all the variables that `a` can point to. `update_written()` will thus generate new primary SSA numbers for these variables and new secondary SSA numbers for all pointers that can point to them including `a`.

Note that even in the case of loops, the analysis is single-pass. This is possible thanks to the availability of Write Sets. Similarly, because Read/Write sets is an interprocedural analysis we can use write sets to accurately model the effect of function calls to SSA numbers.

Also, notice in the algorithm that when processing conditionals we must make a copy of `in_table`. At the end of processing both sides of the conditional the copy can be discarded. Instead of merging the two tables we can use the Write Set to know which variables would have to be merged and we generate new SSA numbers for them.

3.2 Points of attention

There are a few additional details in Extended SSA Numbering that must be addressed.

First, let's consider `break` and `continue` statements. Figure 8(a) shows a sample while loop and Figure 8(b) shows the control flow graph in SSA form that would be built for this loop. Figure 8(c) shows the same while loop but introduces a `break` statement and Figure 8 is its control flow graph. Notice that at statement *S36* we need to join two values of `x`, one from outside the loop and one from previous iteration. This shows up as `x2 =` ϕ`(x1,x4);` in the control flow graph. To help distinguish variable uses inside a loop and after the loop we decided to insert ϕ-nodes to join the value generated after the loop finished with the value

```
fun update_written(stmt,in_table)
{
    foreach v in WriteSet(stmt)
      new_SSA(v,in_table)
      foreach pointer p
        if(at stmt, p points to v)
           new_secondary_SSA(p,in_table)
}

fun process_statement(stmt,in_table)
{
    switch(stmt)
      < a = b op c >
        Store_SSA(b,in_table,stmt)
        Store_SSA(c,in_table,stmt)
        update_written(stmt,in_table)
        Store_SSA(a,in_table,stmt)

      < a = *b >
        Store_SSA(b,in_table,stmt)
        update_written(stmt,in_table)
        Store_SSA(a,in_table,stmt)

      < *a = b >
        Store_SSA(b,in_table,stmt)
        update_written(stmt,in_table)
        Store_SSA(a,in_table,stmt)

      < if(a op b) { then_block } else { else_block } >
        Store_SSA(a,in_table,stmt)
        Store_SSA(b,in_table,stmt)
        in_table2 = duplicate(in_table)
        process_statement(then_block,in_table)
        process_statement(else_block,in_table2)
        update_written(stmt,in_table)

      < for(INIT;a op b;INCR) { body } >
        process_statement(INIT,in_table)
        update_written(stmt,in_table)
        Store_SSA(a,in_table,stmt)
        Store_SSA(b,in_table,stmt)
        process_statement(body,in_table)
        process_statement(INCR,in_table)
        update_written(stmt,in_table)

      < while(a op b) { body } >
        update_written(stmt,in_table)
        Store_SSA(a,in_table,stmt)
        Store_SSA(b,in_table,stmt)
        process_statement(body,in_table)
        update_written(stmt,in_table)

      < do { body } while(a op b) >
        update_written(stmt,in_table)
        process_statement(body,in_table)
        Store_SSA(a,in_table,stmt)
        Store_SSA(b,in_table,stmt)

      < f(a,b,...) >
        foreach function argument v
           Store_SSA(v,in_table,stmt)
        update_written(stmt,in_table)

      ...

    process_statement(NEXT(stmt),in_table)
}
```

Figure 7: Partial algorithm for SSA Numbering analysis

before the loop started in the case that the loop does zero iteration. The insertion of the join node happens at statement point *S37*. The effect of inserting a `break` statement introduces a new possible path to the end of the loop. Adding this path introduces another argument to the ϕ-node changing it to `x5 =` ϕ`(x1,x2,x3)` from `x5 =` ϕ`(x1,x2)`. Because we concentrate on the SSA number, in this case 5, Extended SSA Numbering will not change.

On the other hand consider a do-while loop. Figure 9(a) shows a sample do-while loop and Figure 9(b) shows the control flow graph in SSA form that would be built for this loop. Figure 9(c) shows the same do-while loop but introduces a `break` statement and Figure 9 is its control

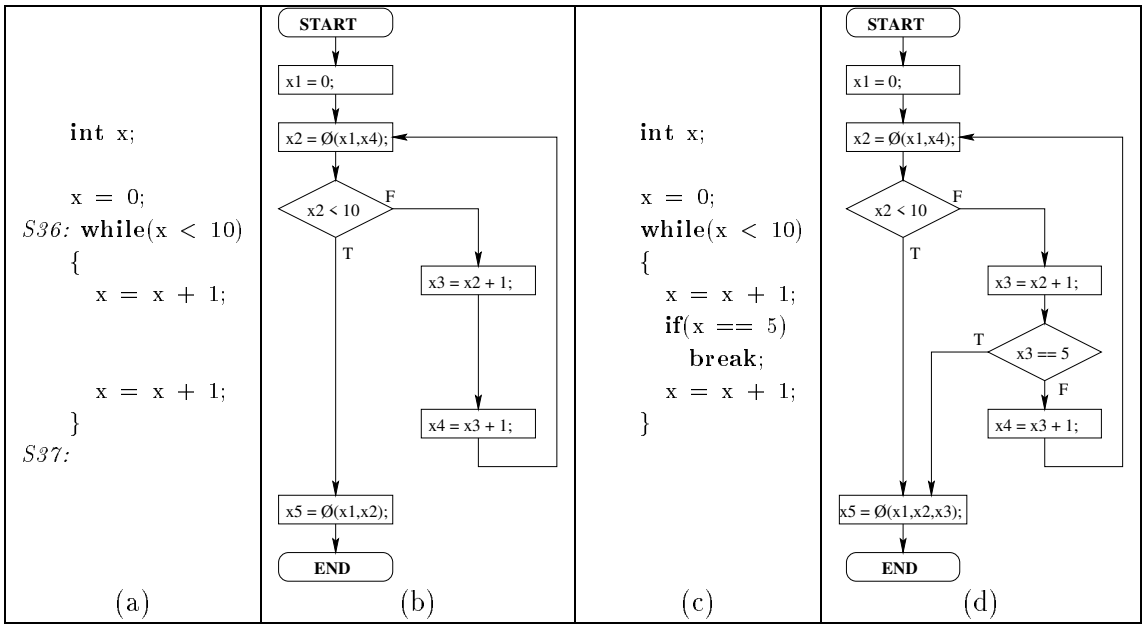

Figure 8: Effects of `break` statements in while loops

flow graph. Similarly to a while loop there is a need to join data coming from outside of the loop with data coming from the previous iteration at statement point *S38*. As opposed to while loop, do-while loops execute at least once. We therefore don't need to join data from before the loop with the data of the last iteration at the statement point *S39*; however, if we add a `break` statement we generate a short cut to the end of the loop requiring a join of data. This join appears in Figure 9(d) as `x5 =` ϕ`(x4,x3)`.

Now that we have pointed out the need for extra ϕ-nodes we must ask which variables need ϕ-nodes (or in our case, which variables need a new SSA number). The answer is those variables that are are modified "after"[1] the `break` statement. We can do that by checking the write sets of these statements and save some work using the hierarchical properties of the Read/Write sets. We chose a very simple conservative approach to generate new SSA numbers for all variables that are written by the whole loop.

We change the algorithm for do-while loops in the following way. `process_statement()` function returns `TRUE` or `FALSE` to signify if there exists a `break` statement in the passed statement block such that it can effect a loop containing this block. For example passed statement block S may contain a loop L. Loop L can have a `break` statement, but this `break` statement will not influence the control flow of anything outside of S; therefore, `process_statement()` will return `FALSE`. On the other hand if there is no loop or `switch` statement surrounding a `break` statement in S then `process_statement()` must return `TRUE`.

[1]The use of the word "after" should be taken to mean any statements that follow the last branch containing the `break` statement. This is because by definition any statement literally after a `break` are dead code.

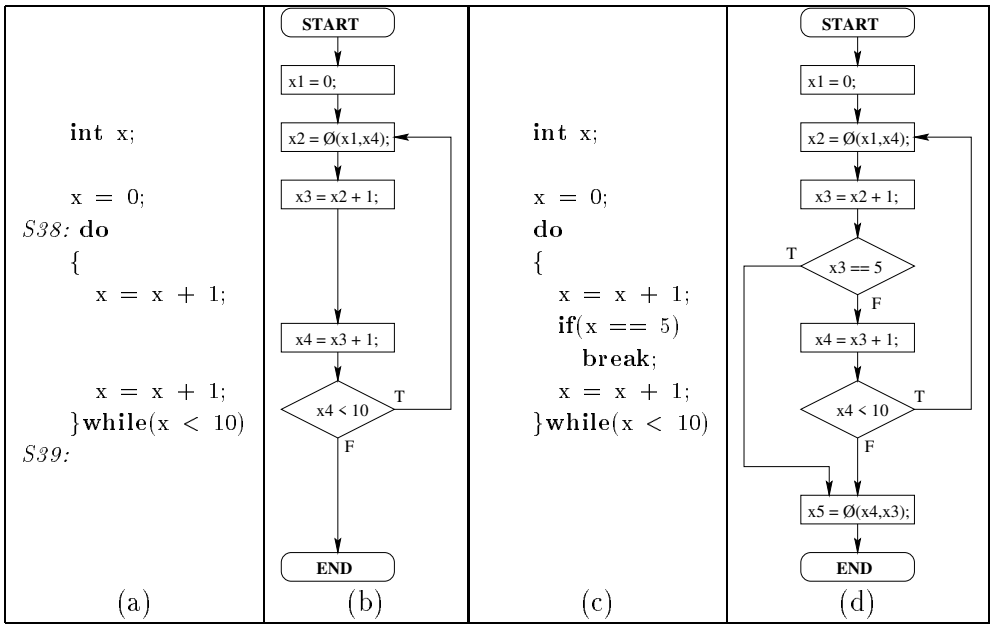

Figure 9: Effects of `break` statements in do-while loops

We now replace the code for do-while loops by

```
< do { body } while(a op b) >
  update_written(stmt,in_table)
  broken = process_statement(body,in_table)
  Store_SSA(a,in_table,stmt)
  Store_SSA(b,in_table,stmt)
  if (broken)
    update_written(stmt,in_table)
```

The only change is the additional `update_written()` call which generates new Extended SSA Numbers if they are needed. We must also do a very similar change for continue statements. A continue statement brings the control flow to just before the condition test. We would therefore insert the call to `update_written()` just before saving the SSA numbers of the variables in the condition.

Next, let us consider handling arrays in Extended SSA Numbering. Arrays can be treated as one entity. A write or read from different elements of the array need not be distinguished. Such methods are common (see [4, pages 460-461]) for handling arrays in SSA form as well as other data-flow analyses. Each update of an array element and each access or an array element are modeled using `A = update(A,ele_num,val);` and `x = access(A,ele_num);` functions. These functions hide partial access or update of the array and treat the array as one entity.

Lastly, we must address the issue how to SSA number structures and their fields. There are two approaches to doing this.

Method 1 Only scalar and array structure fields (ie. leaf fields) are assigned SSA numbers. In the case of structure copy assignments new SSA numbers must be generated for all such fields. For example a field reference may look like `x = a.b.c.d`$_5$`;` assuming that the field `d` is not of structure type.

Method 2 SSA numbers are associated with all the fields as well as with the structure itself. Assignment to a field deals with the SSA number associated with the field and a structure copy assignment changes only the SSA number associated with the structure as a whole. For example a field reference may look like `x = a`$_1$`.b`$_1$`.c`$_2$`.d`$_5$`;`. In this case the meaning of two references to having the same Extended SSA Numbers means that each corresponding structure/field have matching Extended SSA Numbers.

We chose to implement method 2. Method 2 is more cumbersome but for us it is important to have the primary SSA number for pointer variables to structures (eg. `x = (* p`$_{1,3}$`).b`$_4$`;`). Figure 10 shows a sample program manipulating points with SSA numbers assigned to structures for both methods. In method 1 at statement *S40* new SSA numbers for both fields `x` and `y` were generated. In method 2 at statement *S41* only the SSA number for the structure *a* was generated and the SSA numbers for its fields remained the same.

struct {	**struct** {
int x,y;	int x,y;
} a,b;	} a,b;
int v;	**int** v;
$a.x_1 = 0;$	$a_1.x_1 = 0;$
$a.y_1 = 0;$	$a_1.y_1 = 0;$
$b.x_1 = 1;$	$b_1.x_1 = 1;$
$b.y_1 = 1;$	$b_1.y_1 = 1;$
S40: a = b;	*S41:* a = b;
$v_1 = a.x_2 + a.y_2;$	$v_1 = a_2.x_1 + a_2.y_1;$
(Method 1)	(Method 2)

Figure 10: Example program showing SSA numbers for structures using both methods

4 Properties of Extended SSA Numbering

As mentioned in section 1 Extended SSA Numbering is not SSA form and does not have all the properties of SSA form. Some properties were lost and few properties for pointer variables have been introduced. Let us consider more closely the properties of Extended SSA Numbering.

4.1 Factored use-def and def-use chains

First, because Extended SSA Numbering has no ϕ-nodes we can not use it to get use-def and def-use chains. For this reason we need reaching definitions as a separate analysis. McCAT is equipped with this analysis where for each definition point a set of statements is stored indicating all the statements where this newly generated value may possibly be used. Also, for all variable references a set of statements listing the possible definitions points that may reach the reference is stored at the reference point.

It is commonly pointed out that using SSA form to provide use-def and def-use chains is space efficient, where as the space requirement for storing sets of possible uses and sets of possible definitions can grow very quickly. In theory this is very true; however, we have tested many benchmarks and on average programs do not require more then two or three elements per set. Considering that in SSA form one requires to insert ϕ-nodes, we feel the cost to be not significant.

Table 3 gives a short description of each benchmarks we measured as well as an idea of the size of the benchmark by giving the number of SIMPLE statements used to represent it. Table 4 shows the average size of DU and UD chains of the benchmarks in Table 3. For each benchmark we checked the reaching definitions set for each variable appearing on the right hand side of every assignment statement. Also for every assignment statement we checked the uses set for the variable appearing on the left hand side. Columns labeled *Avg # Defs per set* and *Avg # Uses per set* give the average size of the reaching definitions/uses set stored at each variable reference. Columns labeled *Sample size* state how many variables were checked to get the average sizes.

4.2 Definition point properties

Based on the "one definition per name" property of SSA form we can derive the following two properties for Extended SSA Numbering about the value returned by a variable or indirect reference.

Property 1 *Two references of variable* v *at statement* S *and statement* T *will produce the same value if SSA#(*v,S*) = SSA#(*v,T*)* [2].

Note that SSA#(v,*S*) returns the primary SSA number of v at statement *S*. Similarly for indirect references

Property 2 *Two indirect references* *p *at statement* S *and statement* T *will produce the same value if SSA#(*p,S*) = SSA#(*p,T*) and ExtSSA#(*p,S*) = ExtSSA#(*p,T*)* [2].

[2]Because Extended SSA Numbering as well as SSA form are static analyses we assume statements are within the same loop iteration and function call invocation.

Benchmark Name	Description	# SIMPLE stmts
test1.c	Calculate minimum spanning circle for set of points	271
test2.c	Bubble sort, Quick sort, Quick sort 3-median	145
test3.c	Compute determinant of matrix	129
test4.c	Knight's Tour problem	103
test5.c	Find complex roots of polynomial	352
test6.c	Heap sort	138
test7.c	Computes digits Mersenne prime $M = (2^p - 1)$	134
test8.c	Different sorting algorithms	114
test9.c	Graph clustering algorithms	706
travel.c	Traveling salesman problem	450
frac.c	Represent floating point as quotient of 2 integers	126
exptree.c	Build expression tree out of set of integers and operators to evaluate to given total	289
nsieve.c	Sieve of Eratosthenes	182
paging.c	Virtual memory simulation	276
nqueens.c	N Queens problem	333
sim.c	A time-efficient, linear-space local similarity algorithm	1821
unroll.c	Matrix multiply with 0 to 8 iterations unrolled	486
tomcatv.c	Mesh generation with Thompsons solver, vectorized version	608
nrcode2.c	One part of application suite for vectorizing C compilers	232
asuite.c	Test suite for vectorizing C compilers (Argonne)	449
pgm4.c	Hash table implementation with linked list buckets	114
pgm6.c	Construct binary trees as indexed data structure	140
dhrystone.c	Dhrystone benchmark	258
intmm.c	Integer matrix multiply	54
polymesh.c	Compute normals for a polygon mesh at each vertex	339
grid.c	Simple average processing on all matrix elements	208
lorenz.c	Solves the "Lorenz" equations using Runge-Kutta 4	154
matrix.c	Matrix multiplication	803
matrix2.c	BalanceMatrixt, ElmHes, Four1 from Numerical Recipes	500
newton.c	Newton method for finding the sqrt(2.01) given sqrt(2.0), sqrt(2.1), sqrt(2.2), ...	84
genetic.c	Genetic program, population size etc...	388

Table 3: Description of benchmarks used to measure space requirements of DU, UD chains.

Note that ExtSSA#(v,*S*) returns the secondary SSA number of v at statement *S*.

Further, using Extended SSA numbers as part of the name we can derive two more properties.

Property 3 *Two references of variable* v *at statement* S *and statement* T *will have identical sets of possible definition points if SSA#(*v,S*) = SSA#(*v,T*)*

Property 4 *Two indirect references* *p *at statement* S *and statement* T *will have identical sets of possible definition points if SSA#(*p,S*) = SSA#(*p,T*) and ExtSSA#(*p,S*) = ExtSSA#(*p,T*).*

Benchmark	Avg # Defs per set	Sample size	Avg # Uses per set	Sample size
test1.c	1.09	249	1.32	222
test2.c	1.25	59	1.88	92
test3.c	1.57	77	1.76	88
test4.c	1.56	48	1.90	58
test5.c	1.17	225	1.48	252
test6.c	1.53	72	1.84	98
test7.c	1.54	48	1.71	80
test8.c	1.76	45	2.37	70
test9.c	1.41	239	2.41	421
travel.c	1.64	210	1.85	313
frac.c	1.61	84	2.01	76
exptree.c	1.50	123	1.62	167
nsieve.c	2.19	78	2.07	97
paging.c	2.21	53	2.50	143
nqueens.c	4.81	142	1.57	205
sim.c	2.53	1180	3.14	1194
unroll.c	1.50	265	2.20	346
tomcatv.c	1.36	567	1.85	523
nrcode2.c	3.20	133	2.54	163
asuite.c	1.51	145	2.41	252
pgm4.c	1.55	33	1.57	70
pgm6.c	1.82	67	1.61	76
dhrystone.c	1.30	134	1.43	144
intmm.c	1.42	19	1.82	33
polymesh.c	1.68	117	1.31	210
grid.c	1.28	163	1.81	178
lorenz.c	3.13	108	2.19	93
matrix.c	1.52	461	2.59	552
matrix2.c	1.74	452	2.33	398
newton.c	1.37	41	1.96	55
genetic.c	2.07	129	1.86	217

Table 4: Average sizes of Defs and Uses sets stored at each variable reference.

It was our intention in designing this analysis to make these properties available to optimizing transformations and other analyses.

5 Applications of Extended SSA Numbering

Extended SSA Numbering appears to have many uses. Here are some analyses and transformations that use SSA numbers in McCAT.

5.1 Symbolic analyses

When studying methods for symbolically comparing expressions we encountered the need to recognize when two symbols represent the same value. For example, if in the process of Array Dependence Testing [10] we encounter the pair of index expressions, `i + *p` and `i + *p + 1`, we want to know if the value represented by `*p` is the same. If so, we can equivalently test `i` with `i + 1`. It is clear how property 2 allows us to test if `*p` denotes the same value in each expression. Property 1 is used to test scalar variables.

5.2 Induction variables

Wolfe [13] has presented a method for recognizing induction variables and computing their formulas. Their algorithm used SSA form. If one does not have SSA form or the programming language they are analyzing makes SSA form difficult to implement, then to recognize induction variables one must recognize when the pair (variable, set of reaching definitions) have been already visited when performing backward substitution. Properties 3 and 4 can perform this test without looking at the set of reaching definitions.

5.3 Recognition of `ref` parameters

In C there exists no syntax for passing parameters by reference and thus to accomplish this programmers use pointers. Optimizing compilers may benefit from knowing that some particular pointer parameters are really `ref` parameters. We can easily notice `ref` parameters by comparing the primary SSA number of each formal pointer parameter before and after applying Extended SSA Numbering to the body of a function. If the primary SSA numbers remain the same the pointer has never changed the place it points to; thus it can be treated as a `ref` parameter.

5.4 Strengthening reaching definitions

Consider the example in Figure 11. We are interested in the possible definitions of `*p`. When analyzing statement *S34* we have that `n` is defined at *S32* and `m` is defined at *S33*. Statement *S34* introduces a new definition but we don't know which variable it is defining until run time. We could be introducing a new definition for `n` or for `m`. Because we don't know which, we must be conservative and say that from now on the possible definitions for `n` are {*S32*, *S34*} and for `m` they are {*S33*, *S34*}. Then at statement *S35* we conclude that the reaching definitions for `*p` is the union of the reaching definitions for `n` and `m` which is {*S32*, *S33*, *S34*}.

Now consider using Extended SSA Numbering to help. If If SSA#(`p`,*S35*) = SSA#(`p`,*S34*) and ExtSSA#(`p`,*S35*) = ExtSSA#(`p`,*S34*) then the value returned by `*p` at statement *S35* will the same as the one produced by statement *S34*. We can thus state that the sole reaching definition is *S34*.

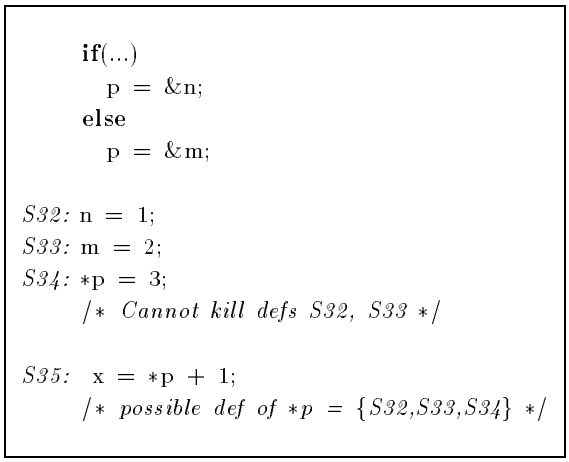

Figure 11: Example of reaching definitions for pointer references.

5.5 Reduction of indirect references

Right away from the conclusions of section 5.4 we can come up with the following optimizing transformation. Each time we recognize a definition and uses via indirection of the same pointer such that both primary and secondary SSA numbers match we introduce a scalar temporary variable to pass the value without indirection to all matching uses. Similarly if more then one indirect reference via the same pointer such that both primary and secondary SSA numbers match is recognized, the referenced value is stored in a temporary variable to be accessed directly in the future matching uses. The two transformations are illustrated in Figure 12. Care also has to be taken to ensure that the statement at which the value is stored into a temporary dominates[3] all the matching uses being replaced.

We have implemented this transformation in McCAT and we would like to stress the importance of providing SSA Numbers for structures because it is rare for this optimization to be applicable otherwise. It may be because as C programmers we are trained to perform this optimization ourselves.

We have applied this transformation to many benchmarks. Unfortunately many of our benchmarks had no opportunity to apply the transformation. Table 5 gives a list of benchmarks where our transformation was applied. Table 6 shows the effectiveness of our transformation. For each benchmark table 6 gives the number of times the transformation was applied and statically on average how many indirect references were saved by applying the transformation and finally how many indirect references were avoided at run time for one example input.

The effectiveness of this transformation varied greatly. For example two benchmarks `frac` and `vlsi` produced very little benefit at run time, even though the benchmark `frac` applied the transformation more times then any other. On the other hand a benchmark like `puzzle`

[3]Statement S dominates statement T if and only if S is in every execution path from the $START$ to T

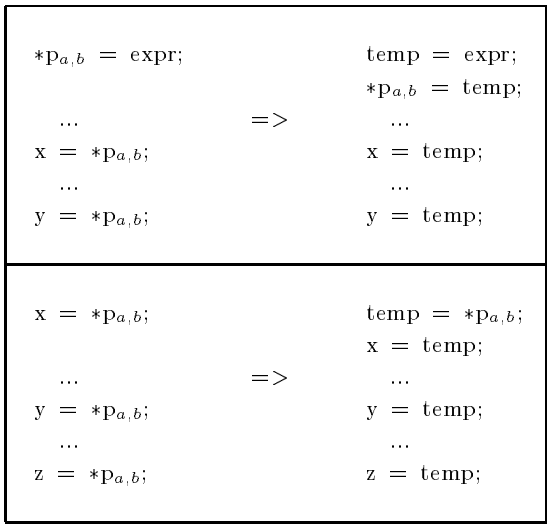

Figure 12: Two transformations to reduce indirect references

Name	Description
vlsi	VLSI chip testing problem
frac	Represents floating point number into a quotient of two integers
chomp	Simple game solver using game tree
queens	N Queens problem
stanford	Stanford baby benchmark suit from John Hennessy
travel	Traveling Salesman problem
puzzle	Solve 15 numbered square puzzle

Table 5: List of benchmarks to which we applied our indirect reference reduction.

produced the most benefit at run time even though there was only one place in the program were we could apply the transformation.

6 Related Work

We are not aware of any work done on approximating SSA form in situations where it is difficult to put a program into SSA form; however, Cytron and Gershbein [5] have worked on accommodating pointers into SSA form.

Their work concentrated on assignments via pointers. They devised a well behaved function (called `IsAlias(p,v)`) taking a pointer `p` and a variable `v` and it would perform the following:

`p` is tested if it actually points to `v`.

Benchmark	Num Transf.	Avg. Static Savings	Dynamic Savings
vlsi	2	1	1
frac	7	1.29	25
chomp	4	1.50	138
queens	5	1	12,513
stanford	2	1	38,450
travel	1	9	67,037
puzzle	1	1	339,579

Table 6: Results of applying our indirect reference reduction transformation.

- If it is, the function returns the value at `*p`.
- If it is not, the function returns the value in `v`.

After each assignment of the form `*p = ...` a series of statements of the form `v_2 = IsAlias(p,&v_1);` are inserted for each variable that `p` can point to. This has the effect of assigning new names to all variables that can point to `p`

However, Cytron and Gershbein have not addressed indirect references on the right hand side of the assignment statement. Initially one can ask "What needs to be done to `... = *p;`? It seems fine as it is." but the reference tells nothing about the value being returned and it may also have more then one definition point breaking the SSA form property.

Another issue not addressed by Cytron and Gershbein are statements of the form `p = &v;`. When SSA form changes the name of `v`, what do we set the pointer `p` to point to?

On the other hand, their work introduced an interesting idea. They proposed to insert new `IsAlias()` statements incrementally and stop when the outcome of a specific dataflow analysis will no longer change due to exposing of other aliases. This was important because the number of insertions of these `IsAlias()` statements can grow quickly.

Comparing our work with the work of Cytron and Gershbein is difficult, because our objectives were different. We concentrated on providing the properties of SSA form in languages with multi-level pointers where as Cytron and Gershbein concentrated on preserving the way that SSA form explicitly points out the flow of data to facilitate data flow analyses. We compromised on this point, but introduced more properties for indirect references.

7 Conclusions

In conclusion, we have introduced an analysis called Extended SSA Numbering that brings to us some properties of SSA form. The properties that are conserved deal with the fact that each variable use in SSA form has only on static definition point. With these properties now available for languages with multi-level pointers such as C, we can determine when expressions are

guaranteed to return same values. With the extension provided by Extended SSA Numbering indirect references such as `*p` also have these properties.

Directly visible applications of Extended SSA Numbering such as the indirect reference reduction transformation has not proved to be a break through in optimizing/parallelizing compilers; however, we are finding that many other analyses benefit from Extended SSA Numbering in many different ways, and so the cumulative effect of SSA numbers is significant.

SSA Numbers can be implemented in any intermediate representation without introducing the full SSA form.

References

[1] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 1–11, San Diego, California, January 13–15, 1988. ACM SIGACT and SIGPLAN.

[2] Marc M. Brandis and Hanspeter Mössenböck. Single-pass generation of static single-assignment form for structured languages. *ACM Transactions on Programming Languages and Systems*, 16(6):1684–1698, November 1994.

[3] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 25–35, Austin, Texas, January 11–13, 1989. ACM SIGACT and SIGPLAN.

[4] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[5] Ron Cytron and Reid Gershbein. Efficient accommodation of may-alias information in SSA form. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 36–45, Albuquerque, New Mexico, June 23–25, 1993. *SIGPLAN Notices,* 28(6), June 1993.

[6] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 242–256, Orlando, Florida, June 20–24, 1994. *SIGPLAN Notices,* 29(6), June 1994.

[7] Ana M. Erosa and Laurie J. Hendren. Taming control flow: A structured approach to eliminating goto statements. In *Proceedings of the 1994 International Conference on Computer Languages*, pages 229–240, Toulouse, France, May 16–19, 1994. IEEE Computer Society Press.

[8] Ana Marie Erosa. A goto-elimination method and its implementation for the McCAT C compiler. Master's thesis, McGill University, Montréal, Québec, May 1995.

[9] L. Hendren, C. Donawa, M. Emami, G. Gao, Justiani, and B. Sridharan. Designing the McCAT compiler based on a family of structured intermediate representations. In Uptal Banerjee, David Gelernter, Alex Nicolau, and David Padua, editors, *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, number 757 in Lecture Notes in Computer Science, pages 406–420, New Haven, Connecticut, August 3–5, 1992. Springer-Verlag. Published in 1993.

[10] Justiani and Laurie J. Hendren. Supporting array dependence testing for an optimizing/parallelizing C compiler. In Peter A. Fritzson, editor, *Proceedings of the 5th International Conference on Compiler Construction, CC '94*, number 786 in Lecture Notes in Computer Science, pages 309–323, Edinburgh, Scotland, April 7–9, 1994. Springer-Verlag.

[11] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global value numbers and redundant computations. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 12–27, San Diego, California, January 13–15, 1988. ACM SIGACT and SIGPLAN.

[12] Vugranam C. Sreedhar and Guang R. Gao. A linear time algorithm for placing ϕ-nodes. In *Conference Record of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 62–73, San Francisco, California, January 22–25, 1995.

[13] Michael Wolfe. Beyond induction variables. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 162–174, San Francisco, California, June 17–19, 1992. *SIGPLAN Notices*, 27(7), July 1992.