# Extended SSA Numbering: Introducing SSA properties to languages with multi-level pointers

Christopher Lapkowski and Laurie J. Hendren
School of Computer Science
McGill University
[lapkow,hendren]@cs.mcgill.ca

**Abstract**

Static Single Assignment (SSA) intermediate representations have become quite popular in compiler development. One advantage of the SSA form is that each 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.

In this paper we introduce a *Extended SSA Numbering*, a simple analysis that concentrates on the "same name, same value" property of SSA form. The analysis handles all variable references, including those via pointer indirections. For each scalar variable, Extended SSA Numbering associates a primary SSA number, whereas for each pointer variable, it associates both a primary and secondary SSA number. Extended SSA Numbering can be easily implemented in any compiler that supports pointer analysis and side-effect analysis. There is no need to change the intermediate form used in the compiler since SSA numbers can be captured as dataflow attributes.

In this paper we present our implementation of the technique in the McCAT optimizing/parallelizing C compiler. Further, we demonstrate the usefulness of Extended SSA Numbers by describing several typical applications.

## 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. [SG95, CFR$^+$89, CFR$^+$91, AWZ88, RWZ88, BM94]

Although, SSA form neatly handles scalar variables, there is no natural way to handle multi-level pointers and explicit pointer dereferences like those present in C. We propose a new flow analysis technique, called *Extended SSA Numbering*, that provides the property "same name - same value", for both scalar and pointer variables. However, Extended SSA Numbering does not include any notion of $\phi$-nodes, and thus it sacrifices the idea of exactly one static definition point. The technique is simple to implement, simple to use, and can be easily implemented in any compiler that supports pointer analysis and side-effect analysis.

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 $\phi$-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 $\phi$-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 $\phi$-node the corresponding value is returned.

```
int main()
{
    int a,b,c;

    a = 1;
    b = 2;
    if(a < 10)
    {
      a = a + 1;
    }
    else
    {
      a = a + b;
    }

    c = a * b;
}
```
(a)

```
int main()
{
    int a_1,b_1,c_1;
    int a_2,a_3,a_4;
    a_1 = 1;
    b_1 = 2;
    if(a_1 < 10)
    {
      a_2 = a_1 + 1;
    }
    else
    {
S1:   a_3 = a_1 + b_1;
    }
S2: a_4 = φ(a_2,a_3);
    c_1 = a_4 * b_1;
}
```
(b)

```
int main()
{
    int a,b,c;

    a1 = 1;
    b1 = 2;
    if(a1 < 10)
    {
      a2 = a1 + 1;
    }
    else
    {
S3:   a3 = a1 + b1;
    }

    c1 = a4 * b1;
}
```
(c)

Figure 1: Example of SSA form vs. SSA Numbering

Now let us compare SSA form to our proposed 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 $\phi$-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, rather they are just stored as attributes for each variable reference.

### 1.1 Difficulties with pointers

Given the differences between SSA form and SSA Numbering it is clear that the SSA Numbering intermediate representation supports fewer properties than the 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.

| | | |
|---|---|---|
| **int** a;<br><br>a = 1;<br>**if**(a < 10)<br>{<br>a = 2;<br>*S4:* p = &a;<br>}<br>(a) | **int** a_1,a_2,...;<br><br>a_1 = 1;<br>**if**(a_1 < 10)<br>{<br>a_2 = 2;<br>*S5:* p_2 = &a_?;<br>}<br>(b) | **int** a;<br><br>$a_1$ = 1;<br>**if**($a_1$ < 10)<br>{<br>$a_1$ = 2;<br>*S6:* $p_2$ = &a;<br>}<br>(c) |
| */* assume p points to a or b */*<br>c = a + b;<br>*S7:* *p = 5;<br><br>d = a + b;<br>(d) | c_1 = a_1 + b_1;<br>*S8:* *p_1 = 5;<br>???<br>*S9:* d_1 = a_1 + b_1;<br>(e) | $c_1$ = $a_1$ + $b_1$;<br>*$p_1$ = 5;<br><br>*S10:*$d_1$ = $a_2$ + $b_2$;<br>(f) |

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

First consider the problem of taking the address of variables. This problem is illustrated in parts (a), (b), and (c) of Figure 2. Specifically, consider 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 [CG93] 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 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`, we cannot infer anything about the value returned by `*p`. Extended SSA Numbering solves this problem.

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 many analyses and an intermediate representation called SIMPLE[HDE+92]. 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 transformations 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. The transformed SIMPLE-C program can be used to produce a new C program, or it can be used as input for the lower-level representation, analyses and code generation.

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 [EH94, Ero95].

Simplifying multi-level indirections is crucial for Extended SSA Numbering. Program structuring is not necessary, but it allows us to develop simple and efficient structure-based algorithms. Figure 4 shows an example of a program and its corresponding SIMPLE intermediate representation.

In addition to the SIMPLE intermediate representation, Extended SSA Numbering requires two other analyses:

**Points-to analysis [EGH94]:** identifies for each point in the program, which named locations a pointer points to. Figure 5(a) 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):** 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. Figure 5(b) shows the Read and Write sets that are associated with each statement for the SIMPLE-C program in Figure 4(b). Notice that the Read set for the `if` statement (*S16*) contains all the variables read by the condition and the body of the `if` statement.

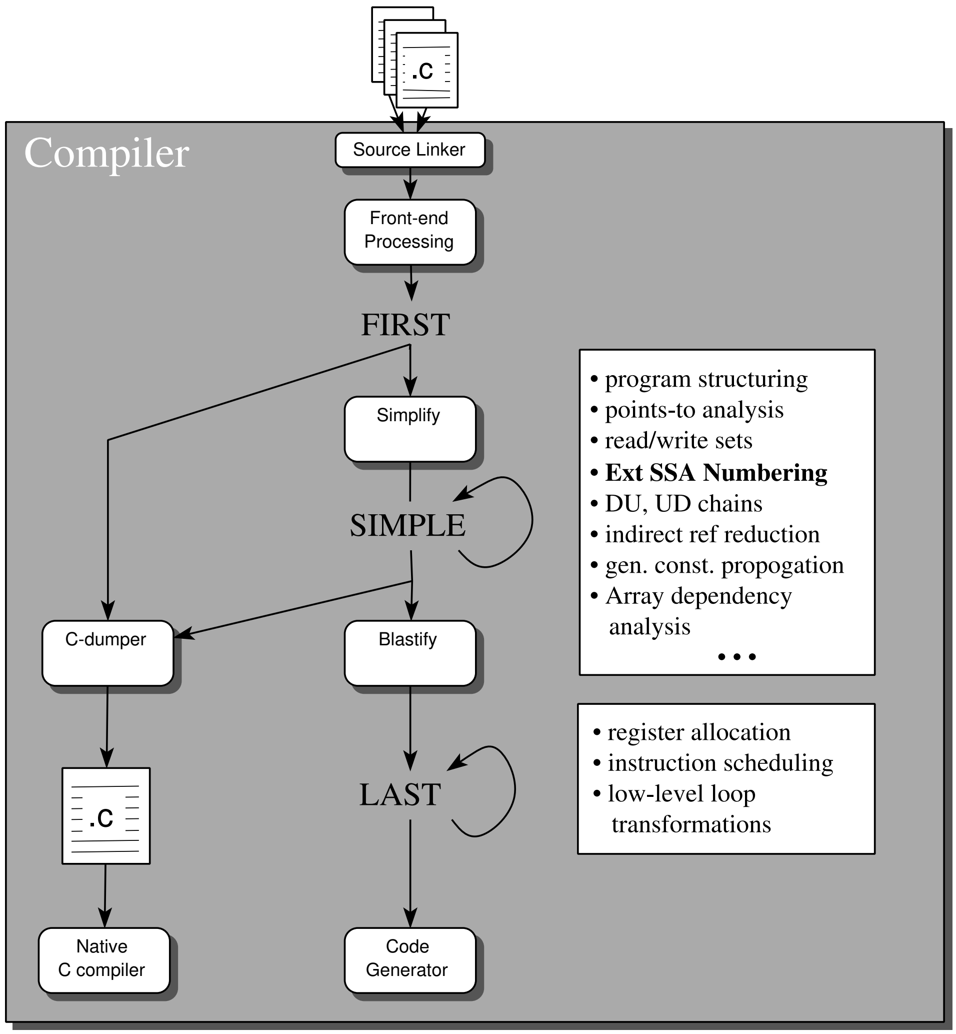


Figure 3: Structure of McCAT compiler

## 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 $\phi$-nodes removed.

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

- No $\phi$-nodes are inserted for SSA Numbering where as in SSA form $\phi$-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.

```
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

| 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)? |

(a) Points-to Pairs

| 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 |

(b) Read/Write Sets

Figure 5: Points-to Pairs and Read/Write sets for the SIMPLE program in Figure 4(b)

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;`). Figure 6 shows how we commonly draw pointers. It points out that there are two values accessible via the pointer, `p` and `*p`.

Since pointers represent two memory locations, modeling their behavior with one number is

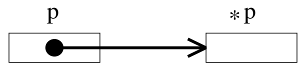


Figure 6: Data accessible via a pointer.

not sufficient. We therefore introduce a second SSA number for pointer variables. We 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 do not require 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 6) and the secondary SSA number is associated with the data the pointer points to (data labeled `*p` in Figure 6). 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 an example of how secondary SSA numbers work consider Figure 7. In this example the statements of the form `x = *p + n;` (statements *S24, S25, S27, S28, S30, S31*) are used to illustrate 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 $\phi$-node just after the `if` statement which would look like $*p_{1,4} = \phi(*p_{1,2}, *p_{1,3})$.

### 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 $\phi$-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 [BM94] for a description of the placement of $\phi$-nodes in structured programs for SSA form.

Figure 8 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 $\phi$-node for `v` is expected, the pair (`v`, `n`) is replaced by (`v`, `n'`) where `n'` is an SSA number that has never been used by 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

```
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 7: Sample program Extended SSA Numbers.

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 8. 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 it also increments the secondary SSA numbers of all pointers that can point to `a`. For an assignment into `*a`, the write set contains all the variables that `a` can point to, call this set $S$. Thus, `update_written()` generates new primary SSA numbers for all variables in $S$. In addition, new secondary SSA numbers must be generated for all pointers that can point to any pointer in $S$ (which of course includes `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.

Details on implementing Extended SSA Numbering in the presence of `break` and `continue` statements can be found in the technical memo version of this paper [LH96].

```
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 8: Partial algorithm for SSA Numbering analysis

### 3.1.1 Handling Arrays

Extended SSA Numbering treats each array as one entity. A write or read from different elements of the array need not be distinguished, since array dependence testers perform this task. However, Extended SSA Numbers can be used to determine when two names denote the same array. Such methods are common (see [CFR+91, 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.

### 3.1.2 Handling Structures

There are two approaches for assigning SSA Numbers to structures.

**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.`$\mathtt{d}_5$`;` 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 =` $\mathtt{a}_1.\mathtt{b}_1.\mathtt{c}_2.\mathtt{d}_5$`;`. 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 = (*` $\mathtt{p}_{1,3}$`).`$\mathtt{b}_4$`;`). Figure 9 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 9: 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 some new 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 $\phi$-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. The McCAT compiler 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 a significant number of 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 $\phi$-nodes, we feel the cost to be not significant. We refer the reader to the longer version of this paper as a technical memo[LH96] for some empirical data.

### 4.2 "Same Name - Same Value" Properties

Even though SSA Numbers does not support factored use-def and def-use changes, we can state the following important properties.

**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*)* [1].

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*)* [1].

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*).*

## 5 Applications of Extended SSA Numbering

Based on the properties presented in the previous section, we can use Extended SSA Numbering in many subsequent analyses and transformations. In this section we outline some of the uses that have been implemented in the McCAT compiler.

### 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, in Array Dependence Testing

[1]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.

[JH94, Lap97] it is often useful to perform symbolic simplification before applying a dependence tester. Consider the case of comparing the two index expressions, `i + *p` and `i + *p + 1`, where `i` is the loop induction variable. If we can determine that `*p` denotes the same value in both index expressions, then we can subtract `*p` from both expressions, can reduce the test to comparing the expressions `i` and `i + 1`. According to property 2, we can just test if both the primary and secondary SSA numbers are the same for both uses of `*p`.

## 5.2 Induction variables

Wolfe [Wol92] has presented a method for recognizing induction variables and computing their formulas based on an 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) has already already visited when performing backward substitution. Properties 3 and 4 indicate that SSA numbers can be used to detect when the set of reaching definitions are the same.

## 5.3 Recognition of `ref` parameters

In C there exists no syntax for passing parameters by reference and thus to accomplish this programmers use explicit pointers. Optimizing compilers may benefit from knowing that some particular pointer parameters are really `ref` parameters. Programmers can also be aided by a tools that automatically indicates which pointers are being used to implement `ref` paramemters.

Extended SSA numbers can be used to detect `ref` parameters. If the primary SSA number of a formal pointer parameter remains constant throughout the body of the function, then we can guarantee that the pointer itself is never updated, and thus it behaves like a `ref` parameter.

## 5.4 Strengthening reaching definitions

Consider the example in Figure 10. 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. It 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 give a more precise reaching definition set. 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*.

## 5.5 Reduction of indirect references

Based on the result from section 5.4, we can note that some definitions and uses via indirections could be replaced by definitions and uses of scalars. This improves the program by reducing the number of memory accesses, promoting more accesses to registers, and conveying more detailed dependence analysis to the instruction scheduler.

```
      if(...)
        p = &n;
      else
        p = &m;

S32: n = 1;
S33: m = 2;
S34: *p = 3;
      /* Cannot kill defs S32, S33 */

S35:  x = *p + 1;
      /* possible def of *p = {S32,S33,S34} */
```

Figure 10: Example of reaching definitions for pointer references.

The transformation operates as follows. 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 there is more than 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 11. Care also has to be taken to ensure that the statement at which the value is stored into a temporary dominates[2] all the matching uses being replaced.

| | | | | | |
|---|---|---|---|---|---|
| $*p_{a,b}$ = expr;<br>...<br>x = $*p_{a,b}$;<br>...<br>y = $*p_{a,b}$; | => | temp = expr;<br>$*p_{a,b}$ = temp;<br>...<br>x = temp;<br>...<br>y = temp; | x = $*p_{a,b}$;<br>...<br>y = $*p_{a,b}$;<br>...<br>z = $*p_{a,b}$; | => | temp = $*p_{a,b}$;<br>x = temp;<br>...<br>y = temp;<br>...<br>z = temp; |

Figure 11: Two transformations to reduce indirect references

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. Table 1 gives a list of benchmarks where our transformation was applied. Table 2 shows the effectiveness of our transformation. For each benchmark table 2 gives the number of times the transformation was applied, 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` produced

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

| 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 1: List of benchmarks to which we applied our indirect reference reduction.

| 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 2: Results of applying our indirect reference reduction transformation.

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

### 5.6 Improving Heap Dependence Tests

The McCAT compiler supports a wide variety of heap analyses, including *connection analysis*[GH96, Ghi96]. Connection analysis determines, for each program point, which heap-directed pointers can point to the same data structure. In order to make connection analysis useful for dependence testing one must introduce the idea of *anchor handles*, or symbolic pointers that capture the value of a heap-directed pointer at a specific program point. Without Extended SSA Numbers, one must create a new anchor handle for each indirect assignment in the program. However, using Extended SSA Numbers, anchor handles can be reused if the primary SSA number does not change. This optimization reduces the number of anchor handles by over 50% [Ghi97].

### 5.7 Summary

Overall, we have found that SSA numbers are useful for many of our subsequent analyses and transformations, and SSA numbers are now an integral part of our compiler.

# 6 Related Work

Most previous work in this area has concentrated on extending SSA form to accommodate pointers[CG93, CCL+96]. In both these cases it is assumed that the compiler uses SSA form for its intermediate representation, and SSA form must be extended to handle pointers.

One of the first approaches, by Cytron and Gershbein [CG93] concentrated on assignments via pointers. They devised a well behaved function (called `IsAlias(p,v)`) that takes a pointer `p` and a variable `v` and performs the following operation. If pointer `p` actually points to `v`, it returns the value of `*p`, otherwise it returns the value of `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`. This is an interesting idea, but only solves part of the problem. It does not completely address indirect references on the rhs of assignments (`... = *p`), nor does it handle statements of the form `p = &v`.

A more complete, and more complicated, approach was suggested by Chow et. al. [CCL+96]. In this case new sorts of operators: the $\mu$ operator to model MayUses, and the $\chi$ operator to model MayDefs. In order to reduce the number of versions required in this approach, they suggest the use of zero versioning which restricts the number of versions for variables that do not correspond to real variables. The technique appears to handle all of the complexities of aliases in C and Fortran, and has been implemented in in the WOPT global optimizer.

Our approach does not attempt to incorporate pointers into SSA form. Rather we take some of the nice properties of SSA form, and capture these via SSA numbers, and then give a very simple method for handling pointers via Extended SSA numbers. Thus, our approach is useful for compilers using a non-SSA intermediate form (for example, any structured Abstract Syntax Tree, or control flow graphs). Our approach is simple, and it does not require changing the structure of the intermediate form. We merely store Extended SSA numbers with variable uses and definitions.

# 7 Conclusions

We have introduced an analysis called Extended SSA Numbering which provides some of the properties of SSA form, even in the presence of pointers. A primary SSA number is associated with each scalar variable, and primary and secondary SSA numbers are associated with each pointer variable. Variables with the same SSA numbers denote the same value, and have the same set of reaching definitions. These properties are useful for a wide-range of subsequent analyses and transformations.

We have provided a simple and efficient algorithm for computing Extended SSA Numbers. Our algorithm operates on a structured intermediate representation, the SIMPLE form of the McCAT compiler. However, the general idea holds for any intermediate representation that has points-to and read/write set information.

Extended SSA Numbers have been implemented in the McCAT C compiler, and they are used in many subsequent phases in the compiler. We demonstrated several of these applications including their use in symbolic analyses and a transformation to reduce the number of indirect references.

## References


[AWZ88] Bowen Alpern, Mark N. Wegman, and F. Kenneth Zadeck. Detecting equality of variables in programs. In *Conf. Rec. of the Fifteenth Ann. ACM Symp. on Principles of Programming Languages*, pages 1–11, San Diego, Calif., Jan. 1988.

[BM94] Marc M. Brandis and Hanspeter Mössenböck. Single-pass generation of static single-assignment form for structured languages. *ACM Trans. on Programming Languages and Systems*, 16(6):1684–1698, Nov. 1994.

[CCL+96] Fred Chow, Sun Chan, Shin-Ming Liu, Raymond Lo, and Mark Streich. Effective representation of aliases and indirect memory operations in ssa form. In *Proceedings of the International Conference on Compiler Construction*, pages 253–267, 1996.

[CFR+89] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *Conf. Rec. of the Sixteenth Ann. ACM Symp. on Principles of Programming Languages*, pages 25–35, Austin, Tex., Jan. 1989.

[CFR+91] 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 Trans. on Programming Languages and Systems*, 13(4):451–490, Oct. 1991.

[CG93] Ron Cytron and Reid Gershbein. Efficient accommodation of may-alias information in SSA form. In *Proc. of the ACM SIGPLAN '93 Conf. on Programming Language Design and Implementation*, pages 36–45, Albuquerque, N. Mex., Jun. 1993.

[EGH94] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proc. of the ACM SIGPLAN '94 Conf. on Programming Language Design and Implementation*, pages 242–256, Orlando, Flor., Jun. 1994.

[EH94] Ana M. Erosa and Laurie J. Hendren. Taming control flow: A structured approach to eliminating goto statements. In *Proc. of the 1994 Intl. Conf. on Computer Languages*, pages 229–240, Toulouse, France, May 1994.

[Ero95] Ana Marie Erosa. A goto-elimination method and its implementation for the McCAT C compiler. Master's thesis, McGill U., Montréal, Qué., May 1995.

[GH96] Rakesh Ghiya and Laurie J. Hendren. Connection analysis: A practical interprocedural heap analysis for C. *Intl. J. of Parallel Programming*, 24(6):547–578, 1996.

[Ghi96] Rakesh Ghiya. Practical techniques for interprocedural heap analysis. Master's thesis, McGill U., Montréal, Qué., Jan. 1996.

[Ghi97] Rakesh Ghiya. *Putting Pointer Analysis to Work.* PhD thesis, McGill U., Montréal, Qué., Nov. 1997.

[HDE+92] 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 *Proc. of the 5th Intl. Work. on Languages and Compilers for Parallel Computing*, number 757 in Lec. Notes in Comp. Sci., pages 406–420, New Haven, Conn., Aug. 1992. Springer-Verlag. Publ. in 1993.

[JH94] Justiani and Laurie J. Hendren. Supporting array dependence testing for an optimizing/parallelizing C compiler. In *Proc. of the 5th Intl. Conf. on Compiler Construction*, number 786 in Lec. Notes in Comp. Sci., pages 309–323, Edinburgh, Scotland, Apr. 1994. Springer-Verlag.

[Lap97] Christopher Lapkowski. A practical symbolic array dependence analysis framework for c. Master's thesis, McGill U., Montréal, Qué., Jun. 1997.

[LH96] Christopher Lapkowski and Laurie J. Hendren. Extended SSA numbering: Introducing SSA properties to languages with multi-level pointers. ACAPS Tech. Memo 102, Sch. of Comp. Sci., McGill U., Montréal, Qué., Apr. 1996. In ftp://ftp-acaps.cs.mcgill.ca/pub/doc/memos.

[RWZ88] Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Global value numbers and redundant computations. In *Conf. Rec. of the Fifteenth Ann. ACM Symp. on Principles of Programming Languages*, pages 12–27, San Diego, Calif., Jan. 1988.

[SG95] Vugranam C. Sreedhar and Guang R. Gao. A linear time algorithm for placing $\phi$-nodes. In *Conf. Rec. of the 22nd ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 62–73, San Francisco, Calif., Jan. 1995.

[Wol92] Michael Wolfe. Beyond induction variables. In *Proc. of the ACM SIGPLAN '92 Conf. on Programming Language Design and Implementation*, pages 162–174, San Francisco, Calif., Jun. 1992.