Increasing the scope and resolution of Interprocedural Static Single Assignment

Silvian Calman and Jianwen Zhu

Department of Electrical and Computer Engineering
University of Toronto, Toronto, Ontario, Canada
{calman,jzhu}@eecg.toronto.edu

Abstract. While intraprocedural Static Single Assignment (SSA) is ubiquitous in modern compilers, the use of interprocedural SSA, although seemingly a natural extension, is limited. We find that part of the impediment is due to the narrow scope of variables handled by previously reported approaches, leading to limited benefits in optimization.
In this study, we increase the scope of Interprocedural SSA (ISSA) to record elements and singleton heap variables. We show that ISSA scales reasonably well (to all MediaBench and most of the SPEC2K), while resolving on average 1.72 times more loads to their definition. We propose and evaluate an interprocedural copy propagation and an interprocedural liveness analysis and demonstrate their effectiveness on reducing input and output instructions by 44.5% and 23.3%, respectively. ISSA is then leveraged for constant propagation and dead code removal, where 11.8% additional expressions are folded.

Key words:SSA, interprocedural, dataflow, constant propagation

1 Introduction

When the *Intermediate Representation* (IR) is in *Static Single Assignment* (SSA) form, each use of a variable is associated with the point where it is defined. To convert the IR into SSA form an algorithm based on Cytron [1] can be used to replace loads and stores for a set of program variables, which we refer to as *SSA variables* and insert ϕ instructions at control flow merge points.

Not all program variables are SSA variables. Usually, they are limited to scalar stack variables, whose address is never taken. This scope can be extended to other stack variables, global variables, and variables allocated on the heap. This extension is usually referred to as *Interprocedural SSA* (ISSA) as it is required to trace the dataflow for SSA variables across procedure boundaries.

SSA form can simplify analysis and optimization algorithms. Due to ϕ instructions, we can distinguish values associated with incoming edges, a property utilized to apply constant propagation, dead code removal [2], and other transformations [3, 4]. Moreover, ϕ instructions can also be used to analyze cyclical dataflow, such as in induction variable analysis [5]. Beyond this, SSA is also

used to simplify other client applications [6–8] by decoupling the dataflow analysis from the implementation. Naturally, it can be expected that ISSA form can help extend these intraprocedural analysis and optimization algorithms to their interprocedural counterparts.

We review two recent ISSA construction algorithms. Liao [9] applied a unification based pointer analysis (Steensgaard's [10]), and renamed memory accesses to the corresponding alias set. Staiger et al. [11] used symbolic variables, called locators, to represent program variables at each procedure. In this work, values are passed interprocedurally by mapping locators to one another and SSA is generated in a traditional way [1], after a pointer analysis step maps all loads and stores to the corresponding locator. Staiger showed that an inclusion-based pointer analysis (Andersen's [12]) reduces memory consumption and considerably speeds up the formation of ISSA, compared to the unification-based pointer analysis (Steensgaard).

The ISSA construction described by Liao [9] and Staiger [11] has a number of shortfalls. First, their ISSA has limited resolution, as some SSA variables represent more than one program variable, creating *may def-use* relations. More specifically, full resolution was only available for scalar globals and as such, copy propagation (and strong updates, etc.) could only be applied to scalar globals. Furthermore, in contrast to traditional SSA form, client applications would have to distinguish between must def-use and may def-use relations. Second, neither Liao [9], nor Staiger [11] applied interprocedural copy propagation, which can fold false merge points or liveness analysis to reduce unnecessary dataflow propagation. The lack of a mechanism to remove redundant ϕ instructions and unused expressions during ISSA construction results in a much greater code size and less precise dataflow.

We propose and evaluate an ISSA form without may def-use relations, implemented in the compiler IR by extending the instruction set. Our implementation considers both scalars as well as scalar elements of records. We handle global and stack variables, as well as variables corresponding to a single dynamic memory location. Using this implementation, we contrast ourselves with previous work and make the following contributions:

- We quantify why the previous approach, in which a flow-insensitive pointer analysis is used and only strong updates to scalars globals are handled (similar to Staiger [11]), is less effective. By handling record elements and singleton heap locations, we replace 1.72 times (on average) more load instructions with their definition. In addition, we observed that the field-insensitive pointer analysis increases the input into procedures by a factor of 12.2, on average.
- We define the value of an instruction in terms of its parent's last invocation and we propose an interprocedural copy propagation algorithm, which reduces input and output instructions by 44.5%, on average. To the best of our knowledge, this is the first paper describing the challenges involved as well as proposing a solution.

– We incorporate a revised interprocedural liveness analysis to limit the variables propagated into and out of procedures, which reduces the input and output instructions by 23.3%, on average.
– We evaluate the benefit of ISSA by applying constant-folding and dead code elimination on the IR (ISSA form) and fold on average 11.8% more instructions than what was provided by the LLVM infrastructure.

In Section 2, we describe the IR extensions for ISSA and the challenges involved in copy propagation. In Section 3, we provide details regarding the implementation and present the algorithms used to identify heap allocated SSA variables, compute liveness analysis, and apply copy propagation. In Section 4, we provide experimental data for the performance and precision of our ISSA form and the improvement observed in constant propagation. Section 5 discusses related work and Section 6 summarizes the major conclusions.

2 Interprocedural SSA

In this section, we present the IR extensions used to handle dereferences and interprocedural value propagation and demonstrate ISSA form construction using an example. In Section 2.1, we describe the difference between intraprocedural and interprocedural copy propagation and outline our algorithm.

In ISSA, dereferences might correspond to multiple locations, including SSA variables. Similar to previous work [6, 13–15], we extend the IR with the conditional load and store instructions (ϕ^L and ϕ^S, respectively). Another issue is value passing at call instructions; we introduce two new instructions, ϕ^V and ϕ^C, to pass the value of a variable across procedure boundaries. These new instructions are discussed in more detail below:

ϕ^S: $pExpr.\phi^S(var, curr, val)$ is used to handle store instructions, where $pExpr$ is the pointer expression. If $pExpr$ is equal to var, then the value of this instruction becomes val, otherwise, the value is $curr$.

ϕ^L: $pExpr.\phi^L(\langle var_1, val_1\rangle, \ldots, \langle var_n, val_n\rangle)$ is used to handle load instructions, where $pExpr$ is the pointer expression. If $pExpr$ is equal to var_i, then the value of this instruction will be val_i.

ϕ^V: $\phi^V_{\langle var,p\rangle}(\langle ci_1, val_1\rangle, \langle ci_2, val_2\rangle, \ldots)$ is used to pass the value of variable var to the entry of procedure p, from a call instruction ci. When entering p from ci_i, the value of this instruction is val_i.

ϕ^C: $pExpr.\phi^C_{\langle var,ci\rangle}(\langle func_1, val_1\rangle, \langle func_2, val_2\rangle, \ldots)$ is used to pass the value of variable var, at the exit from a call instruction ci, where $pExpr$ is the pointer expression for ci, if ci is an indirect call. If $pExpr$ is equal to $func_i$, then the value of this instruction will be val_i. For direct calls, we omit the pointer expression.

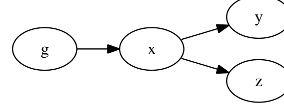
In Example 1(d), we show the ISSA form of Example 1(a). The ISSA form is derived by leveraging the pointer analysis result along with the new instructions (ϕ^L, ϕ^S, ϕ^V, ϕ^C). In Example 1(a), all four global variables g, x, y, and z are SSA

Example 1. Interprocedural SSA Example

```
int   y = 5, z = 10, *x, **g;        1
 C( ) { print( **g ); }              2
 B( ) { *g = &z; }                   3
 main( ) {                           4
  g = &x;                            5
  x = &y;                            6
S1: B( );                            7
  **g = 20;                          8
S2: C( );                            9
  }
```

(a) Code before SSA is applied. Point-to graph is shown in Example 1(b).

(b) Point-to graph for Example 1(a).




```
int   y = 5, z = 10,                 1
   *x, **g;                          2
 C( ) {                              3
  print( 20 );                       4
  }                                  5
 main( ) {                           6
  C( );                              7
  }
```

(c) Code after copy propagation.

```
int   y = 5, z = 10, *x, **g;                          1
 C( ) {                                                2
  x2 = φV⟨x,C⟩(CI2, x1);                               3
  y2 = φV⟨y,C⟩(CI2, y1);                               4
  z2 = φV⟨z,C⟩(CI2, z1);                               5
  print( x2.φL(⟨&y, y2⟩, ⟨&z, z2⟩) );                  6
  }                                                    7
 B( ) { }                                              8
 main( ) {                                             9
CI1:  B( );                                            10
  x1 = φC⟨x,CI1⟩(B, &z)                                11
  y1 = x1.φS(&y, 5, 20);                               12
  z1 = x1.φS(&z, 10, 20);                              13
CI2:  C( );                                            14
  }
```

(d) Code after ϕ^S, ϕ^L, ϕ^V, and ϕ^C instructions are inserted.

variables. A flow-insensitive pointer analysis indicates that x points to either y or z, and g points to x. Since the dereference in Example 1(a) on Line 8 corresponds to either y or z, we need to insert two ϕ^S instructions to handle the store, as illustrated in Example 1(d) on Lines 12–13. Similarly, due to the dereference in Example 1(a) on Line 2, we need to insert a ϕ^L instruction in Example 1(d) on Line 6. Note that procedure B produces the variable x, whereas procedure C uses the variable x and possibly the variables y or z. Hence, we generate the appropriate input and output mappings for procedure B on Line 11 and procedure C on Lines 3–5.

2.1 Copy Propagation

Copy propagation simplifies the IR, as we remove and fold ϕ, ϕ^S, ϕ^L, ϕ^V, and ϕ^C instructions. For instance, by applying copy propagation, we determine the value of $x1$ (Line 11 in Example 1(d)) to be $\&z$ and by folding the ϕ^S and ϕ^L instructions, we produce the code in Example 1(c).

The scope of a value in our framework is the whole program, enabling us to fold ϕ^V and ϕ^C instructions. The benefit of this approach is IR size reduction and the simplification of the def-use relation, as values passed in and out of procedures are masked by ϕ^V and ϕ^C instructions. To this end, we define the value of instruction I in procedure P as the value of I in the last call frame of

P, or otherwise (P is not on the stack) as the value of I in the last invocation of P. Under this definition, the value of I varies with its usage points, but at any program point in P, it is identical in both SSA and ISSA, and as such, ISSA can be constructed on IR in SSA form. Copy propagation is straight forward with the exception of ϕ^V and ϕ^C instructions, which we discuss in the rest of this section.

Let us consider a ϕ^V instruction I^V used in procedure P, merging a single value V. Under our definition, replacing I^V with V is legal as long as V is not located in P. However, V cannot be located in P since V must dominate I^V, and I^V dominates all instructions inside procedure P. Hence, under our definition, ϕ^V instructions merging a single value V, can always be substituted with V.

Example 2. Examples for invalid ϕ^C copy propagation

```
int Sum(                          1
   int a, int b, int c ) {        2
S1: return a + b + c; }           3
void main( ) {                    4
  int     e, f;                   5
                                  6
  e=Sum( 1,2,3 );                 7
  f=Sum( 20,21,315 );             8
  printf("%d,%d\n", f, e );       9
  }
```

(a) Interprocedural copy propagation.

```
StructPtr recursiveProc(                  1
   StructPtr a, StructPtr b ) {           2
   resA = recursiveProc(                  3
      a->right, b->right );               4
   resB = recursiveProc(                  5
      a->left, b->left );                 6
   ...                                    7
   if( resA == resB )                     8
      ...;                                9
   }
```

(b) Recursive procedure dataflow.

Let us consider a ϕ^C instruction I^C merging a single value V located in procedure P_V. Replacing I^C with V is not always legal, as the invocation of P_V which V corresponds to depends on the usage point of I^C. This is illustrated in Example 2, where we present two cases in which the same value can correspond to different instances of an instruction. In Example 2(a), both the first and second return values from procedure *Sum* would correspond to $S1$. If we propagate $S1$ through both ϕ^C instructions corresponding to it, then we would lose the reference to $S1$ returned from the first call. To further emphasize this, in Example 2(b), the values produced in two previous invocation of a recursive procedure are compared. Without distinguishing between such instances, we will erroneously conclude that the branch is always taken.

To discuss a solution for the substitution of ϕ^C instructions, let us assume I^C is in procedure P_{I^C} and defined at call instruction *ci*. First, if the value I^C merged was a constant (i.e. not V), then it could be substituted at all usage points of I^C. Otherwise, to prevent the propagation of values whose parent might still be on the stack, we make sure that **P_{I^C} and P_V do not belong to the same maximal Strongly Connected Component**. Then, we can substitute I^C with V if no other call to P_V is reachable between *ci* and the usage program point (V not redefined – thus it corresponds to value at *ci*). Note that our copy propagation algorithm must be flow-sensitive, since we need to determine the last instance of values substituted for ϕ^C instructions.

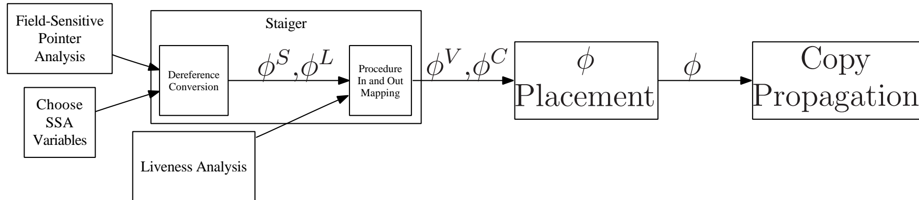


Fig. 1. Overall procedure for ISSA generation.

3 Interprocedural SSA Generation

ISSA is generated in a stepwise procedure, as is illustrated in Figure 1. First, in Section 3.1, we discuss the field-sensitive pointer analysis. In Section 3.2, we describe the SSA variables handled and present the algorithm used to identify singleton heap variables. Similar to Staiger [11], we convert load and store instructions as described in Section 3.3 and map input and output values at call instructions, as detailed in Section 3.4. In Section 3.4, we also describe the interprocedural liveness analysis used to constrain the variables propagated across procedures. Next, we place ϕ instructions to merge values, both interprocedurally and intraprocedurally. We treat the newly inserted ϕ^S, ϕ^V, and ϕ^C as storage instructions; ϕ^L as a load instruction; and we use Cytron's [1] algorithm, unmodified. Lastly, we apply interprocedural copy propagation, as described in Section 3.5.

3.1 Pointer Analysis

Pointer analysis is used to update the call graph and to resolve pointer dereferences into loads, stores, ϕ^S, and ϕ^L instructions (Section 3.3). From practical experimentation on a large number of C benchmarks, we observed that field-insensitivity results in a large number of spurious point-to edges and an increase in the number of variables passed across procedures.

We observed that distinguishing between heap objects reduced the number of spurious point-to edges and therefore, false loads and assignments. Moreover, many benchmarks use memory managers, and distinguishing between heap locations, allocated using interface functions, reduced both the runtime of the pointer analysis and spurious point-to edges (and hence, spurious memory accesses).

3.2 Choosing SSA Variables

Currently, we consider scalar variables and fields of aggregates as potential SSA variables. We handle local variables in non-recursive procedures and in addition to previous work on ISSA, we also handle SSA variables residing in allocation sites, executed at most once in a program, which we refer to as *singular*. If an allocation site A_i is executed at most once, then each dereference resolved to

A_i corresponds to the same memory location. Furthermore, since only a single instance of this instruction exists, each variable v, allocated at A_i, can be an SSA variable.

Singular allocation sites are identified by using the *maximal Strongly Connected Component* (SCC) partitioned call graph and control flow graph. Singular allocations can occur on mutually exclusive paths in the program. As such, we propose an algorithm called *Exclusive Path Singular Allocation Site Identifier* (**EPSASI**). Initially, EPSASI excludes from consideration all procedures in a SCC, or procedures called from a control flow graph SCC, as well as their descendants. In EPSASI, the procedures reached from every call instruction in non-excluded procedures are summed up (using bottom-up traversal). The rest of the algorithm is formulated as an intraprocedural dataflow analysis, identifying procedures invoked more than once on a given path. In the analysis, the domain is a mapping between every procedure p and the maximum number of times p can be invoked, at the entry to a basic block. Initially, at the entry block for a procedure, we initialize the map to 0. When encountering a call instruction ci, 1 is added to the number of possible invocations for each reachable procedure from ci (transfer function). The number of times a procedure p can be invoked at the entry to a basic block bb is the maximum number of invocations p can have in the predecessors of bb (meet operator). We exclude all procedures which have more than one path executing them, as well as their descendants.

When a fixed point is reached, all allocation sites in non-excluded procedures, not located in control flow graph SCCs, are singular.

3.3 Dereference Conversion

We convert dereferences to load and store instructions, which can reference SSA variables. If $pExpr$ is the pointer expression for a load or store instruction and it references a single memory location (according to pointer analysis) for SSA variable var, then we replace $pExpr$ with var.

Let us consider a store instruction, assigning value val, that references more than one memory location, including at least one SSA variable. We will replace this instruction with a series of ϕ^S instructions, with the form
$pExpr.\phi^S(var, curr, val)$, for each SSA variable var, with current value $curr$. If $pExpr$ can also reference a non-SSA variable, we also insert a default ϕ^S instruction, which is executed if none of the other ϕ^S instructions have $pExpr == var$.

In the case of a load instruction that can reference SSA variables $var_0 \ldots var_n$, and non-SSA variables $var_{n+1} \ldots var_{n+1+m}$, where $n + m > 1$, we insert a ϕ^L instruction. If $m == 0$, then the load is replaced with
$pExpr.\phi^L(\langle var_0, val_0\rangle, \ldots \langle var_n, val_n\rangle)$. If $m \neq 0$, then we also add a default value to the ϕ^L instruction, which is taken if none of the other addresses match.

The effect of external call instructions is captured by replacing the call using the load, store, ϕ^L, and ϕ^S instructions. In cases where this can't be done, we commit the value of the variable prior to the call and assign it afterwards.

Note that during copy propagation and constant folding, the pointer expression, *pExpr*, for various ϕ^L or ϕ^S instructions, is resolved. We can then fold these instructions to their corresponding value.

3.4 Procedure Input and Output Mapping

In the rest of this section (and in Section 3.5), we use the following terms:

- $\mathcal{PR} \subset [0, \infty)$ is the set of procedures in the program.
- $\mathcal{BB} \subset [0, \infty)$ is the set of basic blocks in the program.
- $\mathcal{INS} \subset [0, \infty)$ is the set of instructions in the program.
- $\mathcal{VR} \subset [0, \infty)$ is the set of SSA variables in the program.
- $\mathcal{RV} : \mathcal{PR} \mapsto 2^{\mathcal{VR}}$ is a mapping between a given procedure $p \in \mathcal{PR}$ and the set of variables $V \subseteq \mathcal{VR}$ possibly read in p or its descendants.
- $\mathcal{WV} : \mathcal{PR} \mapsto 2^{\mathcal{VR}}$ is a mapping between a given procedure $p \in \mathcal{PR}$ and the set of variables $V \subseteq \mathcal{VR}$ possibly written in p or its descendants.

We use the load, store, ϕ^S, and ϕ^L instructions to determine the initial values of $\mathcal{RV}$ and $\mathcal{WV}$, for each procedure. We then partition the call graph into SCCs and traverse the partitioned call graph using a postorder traversal (bottom-up pass), adding up the read and write sets in the program. The computation of $\mathcal{RV}$ and $\mathcal{WV}$ using this approach is very coarse and we refine it by using a revised liveness analysis. For every procedure $p \in \mathcal{PR}$, we compute $A\mathcal{RV}(p)$, which is the set of variables read after p exits. In addition, we compute the set of variables written before p is first invoked, $B\mathcal{WV}(p)$. Then, we constrain the set of variables passed in and out of procedure p using $B\mathcal{WV}(p)$ and $A\mathcal{RV}(p)$, respectively.

We compute these two sets by using the SCC partitioned call graph to derive a topological visitation order, $TopCG : \mathcal{Z} \mapsto 2^{\mathcal{PR}}$. Likewise, for each procedure $p \in \mathcal{PR}$, we derive a topological control flow graph visitation order $TopCFG : \mathcal{Z} \mapsto 2^{\mathcal{BB}}$. Next, we apply procedure *deriveLimitSets*, presented in Algorithm 1, which visits the call graph in topological order. When $TopCG(i)$ is a SCC, then $\mathcal{RV}$ and $\mathcal{WV}$ are added to $A\mathcal{RV}$ and $B\mathcal{WV}$, respectively, since each procedure in $TopCG(i)$ might be executed multiple times (Lines 9–15). Otherwise, a topological traversal over the SCC partitioned control flow graph of $TopCG(i)$ is applied, using $TopCFG$. During the pass, the set of variables written so far, $WritesSoFar$ (Line 28), and the set of procedures invoked so far, $ProcsSoFar$ (Line 29), are maintained. Conceptually, when visiting a call instruction where the callee is $cp \in \mathcal{PR}$, $WritesSoFar$ are added to $B\mathcal{WV}(cp)$, and when encountering a read to variable var, then var is added to $A\mathcal{RV}$ for each procedure in $ProcsSoFar$. In Algorithm 1, on Line 30, the routine *Summarize* is used to retrieve the set of procedures called in $TopCFG(i)$, along with the set of variables read and written. If $TopCFG(i)$ is not a SCC, then *Summarize* excludes variables written in procedures called from $TopCFG(i)$ and their descendants. In our algorithm $B\mathcal{WV}$ is updated for each called procedure $cp \in Callees$ with $WritesSoFar$, and $A\mathcal{RV}$ is updated for each procedure $psf \in ProcsSoFar$ with the current reads ($currR$). Lastly, if $TopCFG(i)$ is a SCC, then we also update $A\mathcal{RV}$ for psf (Line 35).

Algorithm 1. Top-down computation of $\mathcal{BWV}$ and $\mathcal{ARV}$

```
updateComp = func(                                      1
   Uset : PR ↦ 2^VR,                                    2
   update : 2^VR, callee : PR ) {                       3
   if( ∃i, callee ∈ TopCG(i) )                          4
      forall( p ∈ TopCG(i) )                            5
         Uset(p) = Uset(p) ∪ update;                    6
}                                                       7
deriveLimitSetsSCC = func( i : Z ) {                    8
 forall( p ∈ TopCG(i) ) {                               9
   forall( q ∈ TopCG(i) ) {                             10
     BWV(p) = BWV(q) ∪ WV(q);                           11
     ARV(p) = ARV(q) ∪ RV(q);}                          12
   forall( callee of p, cp ) {                          13
     updateComp(BWV, BWV(p), cp);                       14
     updateComp(ARV, ARV(p), cp);                       15
     } }                                                16
}                                                       17
deriveLimitSets = func( ) {                             18
  for(i = 0; i < |TopCG|; i++) {                        19
    if( |TopCG(i)| > 1 ∨
      TopCG(i) recursive )                              20
      deriveLimitSetsSCC(i);                            21
    else                                                22
      deriveLimitSetsNormal(i);                         23
    } }

deriveLimitSetsNormal = func(i : Z ) {                  24
  ProcsSoFar : 2^PR = ∅                                 25
  WritesSoFar : 2^VR = BWV(TopCG(i))                    26
  for(j = 0; j < |TopCFG|; j++) {                       27
    CurrW, CurrR : 2^VR;                                28
    Callees : 2^PR;                                     29
    ⟨CurrW, CurrR, Callees⟩ =
              Summarize( TopCFG(j));                    30
    WritesSoFar =
              WritesSoFar ∪ CurrW;                      31
    forall( cp ∈ Callees ) {                            32
     updateComp(BWV, WritesSoFar, cp);                  33
      if( TopCFG(i) recursive )                         34
         updateComp(ARV, CurrR, psf);                   35
      }                                                 36
    forall( psf ∈ ProcsSoFar )                          37
      updateComp(ARV, CurrR, psf);                      38
    ProcsSoFar ⋃= Callees;                              39
   WritesSoFar ⋃= ⋃_{cp∈Callees} WV(cp);                40
    }                                                   41
  forall( psf ∈ ProcsSoFar )                            42
    updateComp(
       ARV, ARV(TopCG(i)), psf);                        43
  }                                                     44
```

After $\mathcal{WV}$ and $\mathcal{RV}$ are computed, we insert ϕ^V and ϕ^C instructions. Let us assume that $ci \in \mathcal{INS}$ is the call instruction, $p \in \mathcal{PR}$ is the caller, and $ProcCallees \in 2^{\mathcal{PR}}$ is the set of callees. First, we compute the set of variables propagated to $cle \in ProcCallees$, which we refer to as $PropTo = \mathcal{RV}(cle) \cup \mathcal{WV}(cle)$. Then, for each variable $var \in PropTo$, we add the tuple $\langle ci, val\rangle$ to $\phi^V_{\langle var,cle\rangle}$, where val is the value of var prior to ci. Next, we compute the set of variables written in $ProcCallees$, which we refer to as $PropFrom = \bigsqcup_{cle \in ProcCallees} \mathcal{WV}(cle)$. Afterwards, for each variable $var \in PropFrom$, and each $cle \in ProcCallees$, we add the tuple $\langle cle, val\rangle$ to $\phi^C_{\langle var,ci\rangle}$, where val is the value of var at the exit from cle.

3.5 Interprocedural Copy Propagation

During ϕ-placement, the ϕ^V instructions merging a single value are substituted and we follow up by applying copy propagation to ϕ^C instructions. As described in Section 2.1, we substitute a $\phi^C_{\langle var,ci\rangle}(cle, val)$ instruction merging a single value $val \in \mathcal{INS}$ defined in procedure $pval \in \mathcal{PR}$, as long as $pval$ cannot be called on any path from the ϕ^C instruction, up to the respective use. In basic cases, where val is either a constant or if $pval$ is equal to cle and is called from only one call instruction (not in a SCC), we replace ϕ^C with val.

Otherwise, in order to determine where a ϕ^C can be replaced by val, we identify the call instruction corresponding to the last instance of $pval$, for each procedure at every basic block. To this end, in our implementation, described in Algorithm 2 (and illustrated, using an example in Appendix A), we construct a virtual SSA form, using a quasi variable p_v for each procedure $p \in \mathcal{PR}$, in

the program. The value of p_v will be the call instruction corresponding to the last invocation of p, or $\oslash$ otherwise. Prior to calling procedure *interCopyProp*, we use a bottom-up pass over the SCC partitioned call graph to summarize the set of procedures reached (in *ReachedProcedures*) from every call instruction, $ci \in \mathcal{INS}$. When visiting a procedure, we compute the iterated dominance frontier, $IDF \subset \mathcal{BB}$, for each call instruction, ci. We then add to the $\mathcal{VID}$ relation a mapping from each basic block $bb \in IDF$, to each procedure reached from ci ($ReachedProcedures(ci)$).

Algorithm 2. Interprocedural Copy Propagation

```
SCC : PR ↦ Z;                                        1
ReachedProcedures : INS ↦ 2^PR;                      2
VID : BB ↦ 2^PR;                                     3
Vals : Z ↦ BB × (PR ↦ INS);                          4
SP : Z;                                              5
replOuts = func(                                     6
  p : PR, ci : INS,                                  7
  currV : PR ↦ INS) {                                8
  forall( φ^C_⟨var,ci⟩(⟨pval, val⟩) ) {              9
    if( IsAConstant(val) )                           10
      replWithVal(φ^C_⟨var,ci⟩, val);                11
    else if( SCC(pval) ≠ SCC(p) ) {                  12
      if( IsCalledOnce(pval) )                       13
        replWithVal(φ^C_⟨var,ci⟩,val);               14
      else if( currV(pval) == ci )                   15
        replWithVal(φ^C_⟨var,ci⟩,val);               16
    } }                                              17
  forall( rp ∈ ReachedProcedures(ci) )               18
    currV(rp) = ci;                                  19
}

depthFirstVisit = func (                             20
  p : PR, bb : BB,                                   21
  currV : PR ↦ INS) {                                22
  forall( pdef ∈ VID(bb) )                           23
    currV(pdef) = ⊘;                                 24
  forall( ins ∈ bb.INS )                             25
    replOuts(p, ci, currV);                          26
  forall( succ ∈ getSuccs(bb) )                      27
    Vals(SP + +) = ⟨succ, currV⟩;                    28
}                                                    29
interCopyProp = func ( ) {                           30
  forall( p ∈ PR ) {                                 31
    SP = 0;                                          32
    deriveVID( );                                    33
    Vals(SP + +) = ⟨p.entry, ⊘⟩                      34
    while( SP! = 0 ) {                               35
      ⟨bb, currV⟩ = Vals(− − SP);                    36
      depthFirstVisit(p, bb, currV);                 37
    }                                                38
  }                                                  39
}
```

After $\mathcal{VID}$ is computed, we begin a depth-first traversal of the control flow graph for p, to perform copy propagation for the quasi variables. We treat a call instruction ci as an assignment to each reachable procedure's ($pv \in ReachedProcedures(ci)$) quasi variable, indicating ci was the last call instruction to reach pv. We use $\mathcal{VID}$ to identify basic blocks where a procedure pv might be reached through more than one call instruction and we invalidate the value stored in the quasi variable for pv (on Line 24). Through the copy propagation of the quasi variables, we identify the call instruction associated with the last invocation of each reachable procedure. We can substitute a reference to $\phi^C_{\langle \ldots, ci \rangle}$ (passed out at call instruction ci), with its value val (derived in procedure p), if $Vals(p) == ci$, at the site of the use (see procedure *replOuts* in Algorithm 2).

4 Experiment

In this study, we report on the performance of the interprocedural SSA construction algorithm and we contrast the design choices to previous work. Aside from the runtime, we present and discuss the impact of increasing the scope and resolution of ISSA, applying copy propagation and liveness analysis, pointer analysis, and lastly, the impact on constant propagation.

4.1 Setup, Benchmarks, and Runtime

We implemented the interprocedural SSA in the LLVM [16] compiler infrastructure. The experiments were performed on an Intel CORE 2 Duo 1.66 GHz processor, with 4 GB memory, and running 64-bit ubuntu. These results were collected on IR in intraprocedural SSA form, with constant propagation and dead code removal already applied.

We evaluated our work on a set of MediaBench [17] and SPEC2K [18] benchmarks. In Table 1, we list the various benchmarks used and their lines of code, along with the number of call sites present in the benchmarks.

MediaBench				SPEC2K				SPEC2K			
Name	Lines	Call Sites	T(s)	Name	Lines	Call Sites	T(s)	Name	Lines	Call Sites	T(s)
GSM	4626	258	1.4	164.gzip	8218	306	0.95	197.parser	10932	1691	21.52
JPEG	26173	942	10.8	175.vpr	16984	1902	5.88	254.gap	59493	9773	91.17
MPEG2 [0]	7283	654	2.3	181.mcf	1913	81	1.02	256.bzip2	4665	299	0.74
G721	1476	53	0.3	186.crafty	19478	2252	8.32	300.twolf	19756	1883	38.63

Table 1. Benchmark characteristics and the runtime (column labeled T(s)), in seconds.

In Table 1, we also present the runtime for ISSA generation (does not include pointer analysis runtime). All the MediaBench [17] benchmarks complete within a few seconds and we handle a very large number of variables in them. In comparison, the runtime takes longer for the SPEC2K [18] benchmarks, which is understandable as the benchmarks have more lines of code and more call sites. Furthermore, SPEC2K benchmarks use a greater set of the C language features, including recursion, indirect calls, and cast accesses, which increase the number of inputs and output across call sites.

4.2 Impact of Increasing Scope and Resolution

We evaluate the impact of increasing the scope and resolution of ISSA using the number of SSA variables and the number of load instructions resolved to the corresponding definition. A greater number can provide a higher benefit to clients of ISSA.

In Table 2 we compare our ISSA construction algorithm (columns labeled *All*) to an algorithm which is similar to Staiger [11] as it only considers scalar globals (columns labeled *Globals*), and provide the ratio between them (columns labeled *X*).

We first present the number of SSA variables (heading) for the two algorithm in Table 2. As indicated, we are handling on average *5.17* times more variables than an ISSA formation similar to Staiger [11]. Second, this shows a precise

[0] decoder

Benchmark	SSA Variables			Loads Replaced			Allocation Sites	
	All	Globals	X	All	Globals	X	Singular	%
GSM	73	20	3.65	191	164	1.16	0	0.0
JPEG	249	7	35.57	1564	588	2.66	33	55.0
MPEG2	186	133	1.4	814	650	1.25	1	7.1
G721	14	5	2.8	43	15	2.87	0	0.0
164.gzip	151	100	1.51	575	530	1.08	1	20.0
175.vpr	280	96	2.92	2471	2008	1.23	31	30.4
181.mcf	39	6	6.5	140	15	9.33	3	100.0
186.crafty	403	266	1.52	3406	1501	2.27	5	41.7
197.parser	229	82	2.79	570	520	1.10	2	1.8
254.gap	222	207	1.07	1412	1409	1.00	1	50.0
256.bzip2	41	41	1	478	478	1.00	5	50.0
300.twolf	378	293	1.29	6808	6669	1.02	0	0.0
Average			5.17			1.72		

Table 2. Number of variables handled, load instructions replaced, and singular allocation sites identified.

field-sensitive analysis is useful in increasing the scope of ISSA, making it more useful for structure intensive benchmarks.

Furthermore, in Table 2, we also illustrate the impact of increasing the scope and resolution of ISSA construction on the number of load instructions substituted with their definition (columns underneath *Loads Replaced*). On average, we substituted *1.72* times more load instructions with their definition, than Staiger [11], increasing the scope of the dataflow analysis and its potential benefit.

Lastly, we present the number of singular allocation sites and their percentage (of total allocation sites), in the last two columns (*Allocation Sites* heading). While a large percentage of singular allocation sites were identified in a number of benchmarks, only in *JPEG* this translated to a substantial increase in SSA variables. In other benchmarks, such memory was primarily used for arrays, which we currently do not handle.

4.3 Impact of Copy Propagation and Liveness Analysis

To evaluate the impact of copy propagation and liveness analysis we compute the the sum of ϕ^V and ϕ^C instructions. A lower number indicates both performance (less instructions) and precision improvement, as a lower number results from folding various instructions (i.e. propagation through ϕ^V or ϕ^C instructions), associating additional uses with the corresponding definition.

We apply copy propagation as described in Section 3.5, and fold ϕ^V and ϕ^C instructions. As shown in Table 3, copy propagation reduced the number of ϕ^V and ϕ^C instructions at call sites, and procedure entries, by 44.5% on average. In addition, during copy propagation we folded 30% of the ϕ^V instructions merging values from multiple call sites, as well as a number of ϕ^L and ϕ^S instructions. This

demonstrates a significant improvement over previous work, as copy propagation reduced both the size of the IR as well as the number of spurious merge points.

Benchmark	Liveness Analysis			Copy Propagation			Constant Propagation		
	Total ϕ^V,ϕ^C		Δ	Total ϕ^V,ϕ^C		Δ	Extra Folded	Δ	Extra Dead Blocks
	Before	After		Before	After				
GSM	494	319	35.4 %	319	136	57.4%	9	2.23 %	1
JPEG	10261	9115	11.2 %	9115	4600	49.5%	35	13.01 %	15
MPEG2	6279	5408	13.9 %	5408	3418	36.8%	9	3.59 %	11
G721	133	100	24.8 %	100	66	34.0%	0	0 %	1
164.gzip	2606	2074	20.4 %	2074	1037	50.0%	105	23.6 %	4
175.vpr	5702	4457	21.8 %	4457	2412	45.9%	15	1.81 %	10
181.mcf	262	181	30.9 %	181	12	93.4%	3	7.32 %	2
186.crafty	20935	16373	21.8 %	16373	13276	18.9%	119	2.13 %	3
197.parser	23037	22015	4.4 %	22015	17109	22.3%	133	19.97 %	0
254.gap	100678	61684	38.7 %	61684	48332	21.6%	29	0.55 %	5
256.bzip2	942	614	34.8 %	614	269	56.2%	115	59.28 %	7
300.twolf	5211	4106	21.2 %	4106	2130	48.1%	113	8.67 %	10
Average			23.3 %			44.5%		11.8 %	

Table 3. Impact of liveness analysis and copy propagation measured by the reduction of the read and write sets, along with the effectiveness of constant propagation.

In Table 3, we detail the impact of the liveness analysis, presented in Section 3.4, on reducing the read and write sets into various procedures. The second and third columns contain the sum of ϕ^V and ϕ^C instructions before and after liveness analysis, respectively. The average number of ϕ^V and ϕ^C instructions removed was 23.3%, demonstrating the benefit of liveness analysis in reducing the size of the IR, thus making ISSA construction more efficient.

4.4 Impact of Pointer Analysis

In Table 4, we illustrate the difference between the input sets derived using the field-insensitive pointer analysis available in LLVM and our field-sensitive pointer analysis. The size of the input sets is on average *12.2* times higher in the field-insensitive version, mainly because of the greater point-to set size. Furthermore, since the pointer analysis is used to resolve indirect calls, the field-insensitive version usually contains spurious paths in the call graph. This increases the size of the input sets, as data must be propagated to various unreachable destinations. Larger sets result in increased code size and runtime and hence, by using the field-sensitive pointer analysis, we are able to reduce code size and runtime, in addition to handling more variables.

Benchmark	Field-Sensitive	Field-Insensitive	X
GSM	214	818	3.82
JPEG	330	2480	7.52
MPEG2	1256	12185	9.7
G721	10	83	8.3
164.gzip	1024	4348	4.25
175.vpr	2265	18341	8.1
181.mcf	49	136	2.78
186.crafty	2660	11236	4.22
197.vpr	8239	21398	2.6
300.twolf	581	40806	70.23
Average			12.15

Table 4. Size of input and output sets for a field-insensitive and field-sensitive pointer analysis.

4.5 Impact on Constant Propagation

We implemented a pass that performs constant propagation and dead code removal using ISSA, based on the Wegman and Zadeck algorithm [19]. In Table 3 we show the effectiveness of ISSA based constant propagation in comparison to the LLVM [16] constant propagation and dead code removal passes (*-instcombine, -adce, -ipconstprop*). In the last three columns, we present the number of additional constant folded expressions (and their percentage in relation to LLVM), along with the number of dead basic blocks in the benchmarks. On average, excluding all expressions folded during dereference conversion and copy propagation, we fold an additional 11.8% of instructions, on top of the LLVM passes.

5 Related Work

The challenge in handling pointers in SSA form is that pointer dereferences are not always resolved to a singular memory location and as such, merge points have to sometimes be inserted for pointer dereferences. One way to handle this challenge is to use an aliasing query, as was done by Cytron [13] and others [14, 15].

For interprocedural SSA, dereferences must be handled. In Liao's [20] ISSA form, SSA variables are alias sets (equivalence classes) computed by applying Steensgaard's unification-based pointer analysis [10]. Such derivation creates more merge points than an inclusion-based pointer analysis [11], due to the relatively lower precision which impacts the construction in two ways. First, more spurious assignments are inserted due to a greater point-to set size, and second, the call graph which is used to propagate definitions and uses is less precise as well (in programs with indirect calls). Staiger [11] considered each variable individually, in a manner similar to Horwitz [21]. When encountering

unresolvable dereferences, Staiger merged dataflow by assigning a common allocator to aliased objects. Staiger showed that using more precise pointer analysis would result in a drastically lower number of ϕ instructions; Andersen's pointer analysis had 20× less ϕ instructions than Steensgaard's in some benchmarks. However, Staiger does not apply copy propagation and the analysis outputs its results in graph form – making it harder to directly apply traditional clients of SSA. Along the same lines, the representation is may def-use (e.g. locators correspond to recursive data structures), where only accesses to scalar globals are marked with must use edges. Lastly, Staiger did not evaluate ISSA using a target application.

As shown in Section 4, our approach reduces input and output instructions, while we handle more SSA Variables and replace more load instructions than Staiger [11]. In addition, we demonstrate the benefit of ISSA to constant propagation.

6 Conclusion

SSA can be used for various analysis and optimization algorithms and this paper presents an extension of SSA to the scope of a whole program. We have shown that while handling a large number of variables, we are still able to construct ISSA in seconds. ISSA improves precision by handling a large percentage of load instructions, and by resolving a few pointer dereferences. We have also demonstrated the benefit of liveness analysis and interprocedural copy propagation on ISSA, as well as an improvement in constant propagation and dead code removal, due to ISSA.

From our experiment, ISSA usually performed better in the MediaBench [17] benchmarks, in terms of runtime and precision improvement. This occurred because there was little use of recursive data structures, recursive procedures, and hashtables in MediaBench. Such features make it difficult to resolve dereferences to singular objects and propagate values interprocedurally.

Acknowledgments

We would like to thank the anonymous reviewers for their helpful comments.

References

[1] Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. ACM Transactions on Programming Languages and Systems **13**(4) (Oct 1991) 451–490

[2] Wegman, M.N., Zadeck, F.K.: Constant propagation with conditional branches. ACM Transactions on Programming Languages and Systems **13**(2) (Apr 1991) 181–210

[3] Gal, A., Probst, C.W., Franz, M.: HotpathVM: an effective JIT compiler for resource-constrained devices. In: VEE '06: Proceedings of the 2nd international conference on Virtual execution environments, New York, NY, USA, ACM (2006) 144–153

[4] Stoutchinin, A., Gao, G.: If-conversion in SSA form. In: Euro-Par 2004 Parallel Processing. Volume 3149. (August 2004) 336–345

[5] Wolfe, M.: Beyond induction variables. In: Proceedings of the Conference on Programming Language Design and Implementation (PLDI). Number 7 in 27, New York, NY, ACM Press (1992) 162–174

[6] Hasti, R., Horwitz, S.: Using static single assignment form to improve flow-insensitive pointer analysis. In: Proceedings of SIGPLAN Conference on Programming Language Design and Implementation. (1998) 97–105

[7] Kennedy, R., Chan, S., Liu, S.M., Lo, R., Tu, P., Chow, F.: Partial redundancy elimination in SSA form. ACM Trans. Program. Lang. Syst. **21**(3) (1999) 627–676

[8] Brisk, P., Verma, A.K., Ienne, P.: Optimal polynomial-time interprocedural register allocation for high-level synthesis and asip design. In: ICCAD '07: Proceedings of the 2007 IEEE/ACM international conference on Computer-aided design, Piscataway, NJ, USA, IEEE Press (2007) 172–179

[9] Liao, S.W.: SUIF Explorer: An interactive and interprocedural parallelizer. PhD thesis, Stanford University, CA, USA (2000) Adviser-Monica S. Lam.

[10] Steensgaard, B.: Efficient context-sensitive pointer analysis for C programs. In: Proceedings of the 1996 International Conference on Compiler Construction. (April 1996) 136–150

[11] Staiger, S., Vogel, G., Keul, S., Wiebe, E.: Interprocedural Static Single Assignment Form. In: Proceedings of the 14th Working Conference on Reverse Engineering. (2007) 1–10

[12] Andersen, O.: Program Analysis and Specialization for the C Programming Language. PhD thesis, Computer Science Department, University of Copenhagen (1994)

[13] Cytron, R., Gershbein, R.: Efficient accommodation of may-alias information in SSA form. In: Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation. (1993) 36–45

[14] Chow, F.C., Chan, S., Liu, S.M., Lo, R., Streich, M.: Effective representation of aliases and indirect memory operations in SSA form. In: CC '96: Proceedings of the 6th International Conference on Compiler Construction, London, UK, Springer-Verlag (1996) 253–267

[15] Choi, J.D., Cytron, R., Ferrante, J.: On the efficient engineering of ambitious program analysis. IEEE Trans. Softw. Eng. **20**(2) (1994) 105–114

[16] Lattner, C.: LLVM : An infrastructure for multi-stage optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign (December 2002)

[17] Lee, C., Potkonjak, M., Mangione-Smith, W.H.: Mediabench: A tool for evaluating and synthesizing multimedia and communications systems. In: Micro 30. (1997)

[18] Standard Performance Evaluation Corporation: SPEC CPU2000 benchmarks. `http://www.specbench.org/cpu2000/`

[19] Wegman, M.N., Zadeck, F.K.: Constant propagation with conditional branches. ACM Trans. Program. Lang. Syst. **13**(2) (1991) 181–210

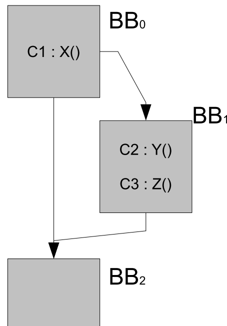
[20] Liao, S.W., Diwan, A., Bosch, Jr., R.P., Ghuloum, A., Lam, M.S.: SUIF Explorer: An interactive and interprocedural parallelizer. In: Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. (1999) 37–48

[21] Horwitz, S., Reps, T., Binkley, D.: Interprocedural slicing using dependence graphs. In: PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation, New York, NY, USA, ACM (1988) 35–46

A Example for Copy Propagation Algorithm

In this section, we illustrate the solution process for Algorithm 2, shown in Section 3.5. Let us consider the example shown in Figure 2 (a). First, we assign identifiers for functions starting at 0 and call sites, starting at 1. For each call site CI, we compute its reachable functions, shown in Figure 2 (b), in the fifth column and add it to $\mathcal{VID}(bb)$, where bb is a basic block in CI's iterated dominance frontier (eighth column).

The reference solution for Figure 2 (a) is shown in the sixth and ninth columns of Figure 2 (b), where the vector index corresponds to the function. After CI_1 the latest call to X and Z is CI_1 and to Y is undefined (similar reasoning applies to CI_2,CI_3 with different functions). Since functions Y and Z are in $\mathcal{VID}(BB_2)$, their value gets invalidated when entering BB_2. The actual replacement of ϕ^C instructions occurs during the traversal, as we query the table to determine whether substitution is possible at various program points.



(a) Structure for currently analyzed procedure.

Function	ID	Call Site	ID	Reachable Functions	Values	Basic Block	$\mathcal{VID}$	Values
X	0	$C1$	1	X,Z	[1,0,1]	BB_0	$\oslash$	[0,0,0]
Y	1	$C2$	2	Y,Z	[1,2,2]	BB_1	$\oslash$	[1,0,1]
Z	2	$C3$	3	Z	[1,2,3]	BB_2	Y, Z	[1,0,0]

(b) Table with data computed prior and during the algorithm. It shows the identifiers for call sites and procedures, as well as the values of quasi variables.

Fig. 2. Example illustrating Algorithm 2, from Section 3.5