Chapter 1

SSA Reconstruction —(*S. Hack*)

1 Introduction

Some optimizations break the single-assignment property of the SSA form by inserting additional definitions for a single SSA value. A common example is live-range splitting by inserting copy operations or inserting spill and reload code during register allocation. Other optimizations, such as loop unrolling or jump threading, perform similar operations. Let us first go through two examples before we present algorithms to properly repair SSA.

motivations

should talk about CFG modifications also

1.1 Live-Range Splitting

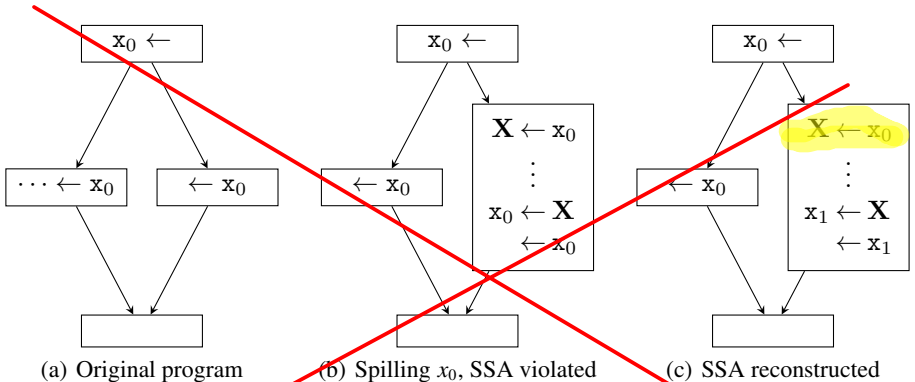
this is not exactly live-range splitting as you remove parts of the live-range (you do not impose your mem var to be under SSA)

introduce the notation X here (I would prefer to see a load & a store)

In Figure 1(b), our spilling pass decided to spill a part of the live range of the variable x_0 ~~in the right block~~. Therefore, it inserted a store and a load instruction. The load however is a second definition of x_0, hence SSA is violated and has to be reconstructed as shown in Figure 1(c).

motivating example with no CFG changes

Example of Fig 2 is sufficient



(a) Original program (b) Spilling x_0, SSA violated (c) SSA reconstructed

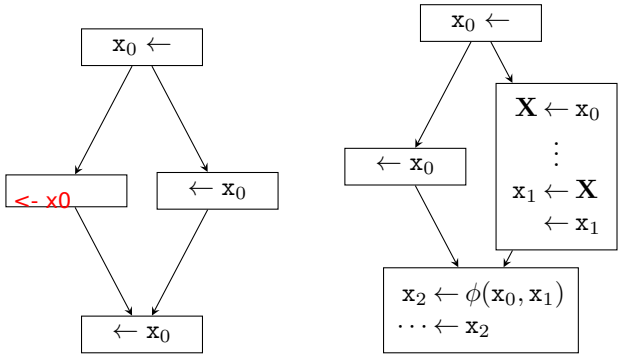
Fig. 1. Adding a second definition

In a more complicated configuration, adding a second definition to an SSA variable can cause ϕ-functions to be inserted. Figure 2 shows the same program fragment with the left use of $\mathbf{x}_0$ being moved to the bottom. As there are two definitions reaching the use in the lower block, a ϕ-function has to be inserted.

Such program modifications are done by many optimizations. Not surprisingly, maintaining SSA is often one of the more complicated and error-prone parts in such

out the SSA violated part
(as in fig 1)

motivating example.
no CFG change (cont)



(a) Original program (b) Second definition of x_0, ϕ-function inserted

Fig. 2. SSA repaired

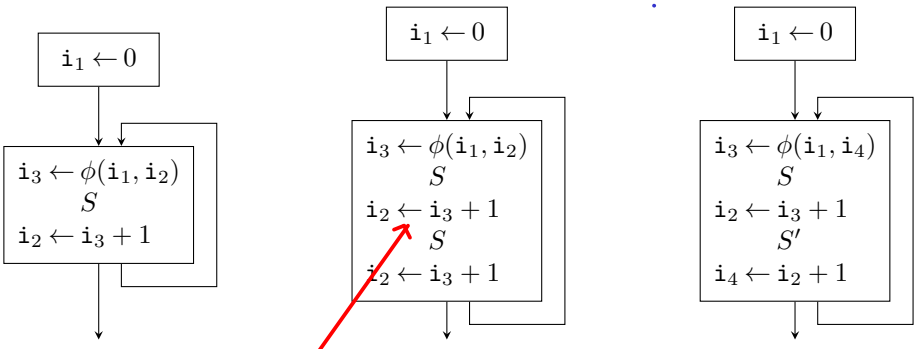
optimizations; owing to the insertion of additional ϕ-functions and the correct redirection of the variable's uses.

1.2 Loop Unrolling

Loop unrolling is a transformation to increase instruction-level parallelism inside the body of a loop. To this end, the body of the loop is duplicated n times. Similar to the example above, this adds several new definitions *and* uses to an existing SSA variable. Consider following example and assume that the loop's body is duplicated once, i.e. the loop is unrolled once.

loop unrolling example
(no change of the CFG)

I would have prefered a code
the changes the CFG. When
you do not know the bound of
the loop (parameter) you need
to change the test
(eg i+1<n instead of i<n)
and add a new test and a new
BB (for odd number of
iterations).



(a) Original loop (b) Unrolled loop, code duplicated, SSA broken (c) SSA repaired

here you duplicate the code so you copy also the phi and have i3 <- i2 (non SSA but correct code)

Fig. 3. Loop Unrolling

There is an important discussion on the different kind of transformations that you can do and when you require the call to the "reconstruction" black box.
I see two kind of transformations:
- no CFG transformations. add / remove some def uses for a set of variables (but ensure that the code is correct eg your example on the loop unrolling). Then you have a precise list of variables (x0 in fig 2; i2 & i3 in fig 3) that have been impacted.
- CFG transformations without any more definitions/uses introduced (remove an empty block, change some edges, add an empty block). Then you need to have a list of BBs for which the set of dominators have changed, and a list of BB for which the dominants have changed. Again, the transformation should be valid. ie it can be non SSA code, but it should be semantically correct (eg you need to put the copy i3:=i2 in your unrolling example as you would do for non SSA code. We do not deal with problems that you already face for non SSA programs here).

The interesting remark is that after this second transformation, if the code is correct, you are still under SSA, you just potentially broke the dominance property. You need to list the variables that may have been impacted by this transformation (this is only what your algo required, a list of variables. You can do better but this might be discussed in the conclusion. See further). A variable that has a use in a BB which dominator set has changed AND which has a def in a BB which dominated set has been changed might be impacted.
Being able to list the impacted variables and hence for CFG changes the impacted BBs is a requirement for your algorithm. And this is sufficient! This set of BB can be provided by the client (who knows what he is doing), if he really knows, he can provide only the set of variables, if he does know anything then he can rely on some sophisticated techniques related on dom-tree incremental update (that you can discuss in the conclusion).

These examples show that naive code duplication and live-range splitting in an SSA program can violate the single assignment property of the SSA form. To repair these effects, we seek for a simple, black box like algorithm. Ideally, we provide a set of definitions and uses of which we claim that they correspond to one single non-SSA variable. Afterwards, the algorithm establishes the SSA form for these live ranges. This includes re-assigning the uses to the correct definition and inserting ϕ-functions on demand. Next, we investigate an algorithm based on the classical dominance frontier construction algorithm.

I think I understand now: because of the way you do unrolling you want to consider several SSA variables as a unique variable (i2 & i3 in your example). But this is really confusing and not clear how you should identify i2 & i3 (more complex than SSA webs).

2 General Considerations

say somewhere that you consider your code to be scheduled.

We consider the following scenario: The program is represented as a control-flow graph (CFG) and is in SSA form. For the sake of simplicity, we assume that each instruction in the program only writes to a single variable. Due to the single-assignment property of the SSA form, we can then identify the program point of the instruction and the variable. An optimization/transformation now violates SSA by inserting additional definitions for an existing SSA variable, like in the examples above. The original variable and the additional definitions can be seen as a single non-SSA variable that has multiple definitions and uses.

to simplify the notations SSA has been broken! Don't think you need this simplification.

and uses

instance of the pb a variable v with multiple defs and multiple uses

In the following, v will be a such a non-SSA variable. D is a set of SSA variables being the definitions of v. A use of a variable is a pair consisting of a program point (a variable) and an integer denoting the index of the operand at the using instruction.

awkward. Why don't you consider D as a set of defs of v. Identify the operands (def & uses) with prog point.

Both algorithms which we are going to present share the same driver routine (Figure 1). First, every basic block `b` is equipped with a list `b.defs` that contains all instructions in the block which define one of the variables in D. This list is sorted according to the schedule of the instructions in the block from back to front. Hence, the latest definition is the first in the list.

for each use, check if exists a def in the BB that reach it otherwise traverse the dom-tree backward to find a reaching def (find_def_begir

meaning of back & front for a list?

I would prefer a set that you enumerate in a given order (the property is stated where you use it)

awkward

Then, all uses of the variables in D are scanned. We search the use's block for a definition of the used variable. By scanning the block's list from back to front, we find the latest such use if one exists. If the variable has no definition in the block, we have to find the definition that reaches this block from *outside*. Here we have to differentiate whether the user is a ϕ-function or not. If it is a ϕ-function, the block of the use is the corresponding predecessor block. We use two functions `find_def_from_begin` and `find_def_from_end` that find the reaching definition at the beginning and end of a block, respectively. As can seen from Algorithm 1, `find_def_from_end` can be expressed in terms of `find_def_from_begin`. `find_def_from_end` additionally considers definitions in the block, while `find_def_from_begin` does. Our two approaches only differ in the implementation of the function `find_def_from_begin`. The differences are described in the next two sections.

for each use u, we consider its block b and scan the list of defs that are in this block (if any) from its end to its beginning so that we find the latest first.

def

the definition that reaches.. otherwise can be understood in two != ways.

3 Reconstruction based on the Dominance Frontier

put all the references in the conclusion. Refer to the book when in the book

This algorithm, proposed by Sastry and Ju [4], follows the same principles as the classical SSA construction algorithm by Cytron et al. [3]. We compute the iterated dominance frontier (IDF) of D (cf. Appel's book [1] for an algorithm on how to compute the IDF).

v.defs

need to say somewhere that you want SSA with dominance property

Algorithm 1 SSA Reconstruction Driver

```
proc ssa_reconstruct(set of var D):
  for d in D:
    b = d.block
    insert d in b.defs according to schedule
    # latest definition is first in list

  for each use u of D:
    v = get_used_var(u)
    b = v.block
    d = None

    # search for a local definition in the block
    for e in b.defs:
      if v == e and is_later(u, e):
        d = e
        break

    # no local definition was found, search in the predecessors
    if d == None:
      # if the user is a phi we have to start the search
      # at the end the corresponding predecessor block
      if u.is_phi():
        d = find_def_from_end(u.block.pred(u.index), ...)
      else:
        d = find_def_from_begin(b, ...)

    rewrite use at u to d

proc find_def_from_end(block b, ...):
  if not b.defs.empty():
    return b.defs[0]
  return find_def_from_begin(b, ...)
```

var v

confusing: one var @ a time

v.defs

v

u.block #predecessor block if u is a use operand of a phi

duplicated

in reverse order of BB's schedule

e.order < u.order

keep only this

this def will be the returned reaching def. The freshly created uses of the phi
will query for their reaching def. This is done using a recursive call to find_def_from_end

traverse backward dom-tree
insert phi on the fly
if no def and in iDF(v.defs)

v.defs

This is because you build SSA with dom property.

???

This set is a sound overapproximation on the set where ϕ-functions must be placed (it might contain blocks where a ϕ-function would be dead). Then, we search for each use the corresponding reaching definition. This search starts at the block of u. If that block is in the IDF of D a ϕ-function needs to be placed at its entrance. The operands of that ϕ-function are then (recursively) searched in the predecessors of the block. If the block is not in the IDF, the search continues in the block's immediate dominator. This is because in SSA, every use of a variable must be dominated by its definition[1]. If the block is not in the IDF, the reaching definition is the same for all predecessors and hence for the immediate dominator of this block. Note that by rewiring the uses of several variables, some variables defined by ϕ-functions may not be used anymore. A dead code elimination pass after SSA reconstruction will remove these. Algorithm 2 shows this procedure in pseudo-code.

Algorithm 2 SSA Reconstruction based on Dominance Frontiers

```
proc find_def_from_begin(block b, set of blocks F):
  if b in F:
    d = new_phi(b)
    i = 0
    for p in b.preds:
      o = find_def_from_end(p, F)
      set i-th operand of d to o
      i = i + 1
  else:
    d = find_def_from_end(b.idom, F)
  b.def = d
  return d
```

Say that F is iDF(v.defs)

let vi new version of v; create d: vi=phi(...) in b; add d into b.defs

the corresponding operand of d to o

4 Search-based Reconstruction

data flow like with recursive algo.
Insert phi on the fly

on

and as we will see later more phi than the minimum required can be inserted.

?

The second algorithm we present here is similar to the construction algorithm that Click describes in his thesis [2]. Although his algorithm is designed to construct SSA from the abstract syntax tree, it also works well on control flow graphs. Its major advantage over the algorithm presented in the last section is that it does neither require dominance information nor dominance frontiers. Thus it is well suited to be used in transformations that change the control flow graph. Its disadvantage is that potentially more blocks have to visited during the reconstruction. The principle idea is to start a search from every use to find the corresponding definition inserting ϕ-functions on the fly while caching the found definitions at the basic blocks. This is similar to the implementation of a data-flow analysis, that places the ϕ-functions. As in the last section, we only consider the reconstruction for a single variable. If multiple variables have to be reconstructed, the algorithm can be applied to each variable separately.

[1] The definition of an operand of a ϕ-function has to dominate the according predecessor block.

we do not know if the definitions are at the block or the definitions that reach the block.

suppose first thqt the CFG is a DAG...

for each block the definitions that can reach it.

you do not check it in the pseudo-code.

We perform a backward depth-first search in the CFG to collect the reaching definitions at each block. To mark a block as visited, the reaching definition of that block is inserted into the `defs` list of that block. If the CFG is a DAG, all predecessors of a block can be visited before the block itself is processed (post-order traversal). Hence, all reaching definitions at a block b can be computed before we decide whether to place a ϕ-function in b or not. If more than one definition reaches the block, we need to place a ϕ-function.

computes recursively the defs that reach each block. Insert a phi if more than 1 reaching def.

if there are more than one reaching def then a phi has to be inserted (and refer to phi_necessary in pseudo code)

If the CFG has loops, there are blocks for which not all reaching definitions can be computed before we can decide whether a ϕ-function has to be placed. Recursively computing the reaching definitions for a block b can end up at b itself. To avoid infinite recursion, we create a ϕ-function without operands in the block before descending to the predecessors. Hence, if a variable has no definition in a loop, the ϕ-function placed in the header eventually reaches itself (and can later be eliminated). When we return to b we decide whether a ϕ-function has to be placed in b by looking at the reaching definition for every predecessor. If the set of reaching definitions is a subset of $\{a, x\}$ where x is the ϕ-function inserted at b, then ϕ-function is not necessary and we can propagate a further downwards. Otherwise, we place a ϕ-function.

insert a phi if more than one reaching def than the inserted phi itself.

phi

awkward

if #reach_def=1 or #reach_def={a,x} with x... (and refer to phi necessary in pseudo code)

Algorithm 3 Search-based SSA Reconstruction

pseudo code

```
proc find_def_from_begin(block b):
  phi = make_phi()        same notations as for algo 3
  if b.defs.empty():
    b.defs = [ phi ]
  else return phi
  reaching_defs = []
  for p in b.preds:
    reaching_defs += find_def_from_end(p)   recursion does not stop here
  if phi_necessary(reaching_defs):
    set_arguments(phi, reaching_defs)   same notations as algo3
    d = phi
  else:
    reaching_defs.remove(phi)   b.defs.remove(phi)
    d = reaching_defs[0]
  return d
```

Give an example for both algo 2 & 3

5 Conclusions

the style is not good. Not a paper. No conclusion as a summary.

Some optimizations, such as loop unrolling or live-range splitting destroy the single-assignment property of the SSA form. In this chapter we presented two generic algorithms to reconstruct SSA. The algorithms are independent of the transformation that violated SSA and can be used as a black box: For every variable for which SSA was violated, a routine is called that restores SSA. The presented algorithms differ in the prerequisites and their runtime behavior:

You should put some references concerning the update of dominance tree & dominance frontier here (Sreedhar, Ramlingam). Update of dominance tree techniques might be used to find the set of BB impacted by the modification of the CFG (see discussion above). Finally, you can mention the technique I forwarded you (mail of Francois) that corresponds to isolate a region by phi functions (or copies) so that you can restrict the changes within that region (of course this is not minimal SSA but who cares...)

1. The first is based on the iterated dominance frontiers like the classical SSA construction algorithm by Cytron et al. [3]. Hence, it is less suited for optimizations that also change the flow of control since that would require recomputing the iterated dominance frontiers. On the other hand, by using the iterated dominance frontiers, the algorithm can find the reaching definitions quickly by scanning the dominance tree upwards.
2. The second algorithm does not depend on additional analysis information such as iterated dominance frontiers or the dominance tree. Thus, it is well suited for transformations that change the CFG because no information needs to be recomputed. On the other hand, it might find the reaching definitions slower than the first one because they are searched by a depth-first search in the CFG.

yes

Both approaches constructed *pruned* SSA (TODO: cite other chapter), i.e. no ϕ-function is dead. One can also show that the first approach produces minimal SSA in the sense of Cytron et al. [3] whereas the second approach might create superfluous ϕ-functions in the case of irregular control flow.

This citation is to make chapters without citations build without error. Please ignore it: [**?**].

References

1. Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, second edition, 2002.
2. Clifford Click. *Combining Analyses, Combining Optimizations*. PhD thesis, Rice University, February 1995.
3. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadek. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.
4. A. V. S. Sastry and Roy D. C. Ju. A new algorithm for scalar register promotion based on SSA form. In *PLDI '98: Proceedings of the conference on Programming language design and implementation*, pages 15–25, New York, NY, USA, 1998. ACM.