Simple and Efficient Construction of Static Single Assignment Form

Matthias Braun[1], Sebastian Buchwald[1], Sebastian Hack[2], Roland Leißa[2], Christoph Mallon[2], and Andreas Zwinkau[1]

[1] Karlsruhe Institute of Technology
{matthias.braun,buchwald,zwinkau}@kit.edu
[2] Saarland University
{hack,leissa,mallon}@cs.uni-saarland.de

Abstract. We present a simple SSA construction algorithm, which allows *direct* translation from an abstract syntax tree or bytecode into an SSA-based intermediate representation. The algorithm requires no prior analysis and ensures that even during construction the intermediate representation is in SSA form. This allows the application of SSA-based optimizations during construction. After completion, the intermediate representation is in minimal and pruned SSA form. In spite of its simplicity, the runtime of our algorithm is on par with Cytron et al.'s algorithm.

1 Introduction

Many modern compilers feature intermediate representations (IR) based on the static single assignment form (SSA form). SSA was conceived to make program analyses more efficient by compactly representing use-def chains. Over the last years, it turned out that the SSA form not only helps to make analyses more efficient but also easier to implement, test, and debug. Thus, modern compilers such as the Java HotSpot VM [14], LLVM [2], and libFirm [1] exclusively base their intermediate representation on the SSA form.

The first algorithm to efficiently construct the SSA form was introduced by Cytron et al. [10]. One reason, why this algorithm still is very popular, is that it guarantees a form of *minimality* on the number of placed ϕ functions. However, for compilers that are entirely SSA-based, this algorithm has a significant drawback: Its input program has to be represented as a control flow graph (CFG) in non-SSA form. Hence, if the compiler wants to construct SSA from the input language (be it given by an abstract syntax tree or some bytecode format), it has to take a detour through a non-SSA CFG in order to apply Cytron et al.'s algorithm. Furthermore, to guarantee the minimality of the ϕ function placement, Cytron et al.'s algorithm relies on several other analyses and transformations: To calculate the locations where ϕ functions have to be placed, it computes a dominance tree and the iterated dominance frontiers. To avoid placing dead ϕ functions, liveness analyses or dead code elimination has to be performed [7]. Both, requiring a CFG and relying on other analyses, make it inconvenient to use this algorithm in an SSA-centric compiler.

K. De Bosschere and R. Jhala (Eds.): CC 2013, LNCS 7791, pp. 102–122, 2013.
© Springer-Verlag Berlin Heidelberg 2013

Modern SSA-based compilers take different approaches to construct SSA: For example, LLVM uses Cytron et al.'s algorithm and mimics the non-SSA CFG by putting all local variables into memory (which is usually not in SSA-form). This comes at the cost of expressing simple definitions and uses of those variables using memory operations. Our measurements show that 25% of all instructions generated by the LLVM front end are of this kind: immediately after the construction of the IR they are removed by SSA construction.

Other compilers, such as the Java HotSpot VM, do not use Cytron et al.'s algorithm at all because of the inconveniences described above. However, they also have the problem that they do not compute minimal and/or pruned SSA form, that is, they insert superfluous and/or dead ϕ functions.

In this paper, we

- present a simple, novel SSA construction algorithm, which does neither require dominance nor iterated dominance frontiers, and thus is suited to construct an SSA-based intermediate representation directly from an AST (Section 2),
- show how to combine this algorithm with on-the-fly optimizations to reduce the footprint during IR construction (Section 3.1),
- describe a post pass that establishes minimal SSA form for arbitrary programs (Section 3.2),
- prove that the SSA construction algorithm constructs pruned SSA form for all programs and minimal SSA form for programs with reducible control flow (Section 4),
- show that the algorithm can also be applied in related domains, like translating an imperative program to a functional continuation-passing style (CPS) program or reconstructing SSA form after transformations, such as live range splitting or rematerialization, have added further definitions to an SSA value (Section 5),
- demonstrate the efficiency and simplicity of the algorithm by implementing it in Clang and comparing it with Clang/LLVM's implementation of Cytron et al.'s algorithm (Section 6).

To the best of our knowledge, the algorithm presented in this paper is the first to construct minimal and pruned SSA on reducible CFGs without depending on other analyses.

2 Simple SSA Construction

In the following, we describe our algorithm to construct SSA form. It significantly differs from Cytron et al.'s algorithm in its basic idea. Cytron et al.'s algorithm is an eager approach operating in forwards direction: First, the algorithm collects all definitions of a variable. Then, it calculates the placement of corresponding ϕ functions and, finally, pushes these definitions down to the uses of the variable. In contrast, our algorithm works backwards in a lazy fashion: Only when a variable is used, we query its reaching definition. If it is unknown at the current location, we will search backwards through the program. We insert ϕ functions at join points in the CFG along the way, until we find the desired definition. We employ memoization to avoid repeated look-ups.

This process consists of several steps, which we explain in detail in the rest of this section. First, we consider a single basic block. Then, we extend the algorithm to whole CFGs. Finally, we show how to handle incomplete CFGs, which usually emerge when translating an AST to IR.

2.1 Local Value Numbering

When translating a source program, the IR for a sequence of statements usually ends up in a single basic block. We process these statements in program execution order and for each basic block we keep a mapping from each source variable to its current defining expression. When encountering an assignment to a variable, we record the IR of the right-hand side of the assignment as current definition of the variable. Accordingly, when a variable is read, we look up its current definition (see Algorithm 1). This process is well known in literature as *local value numbering* [9]. When local value numbering for one block is finished, we call this block *filled*. Particularly, successors may only be added to a filled block. This property will later be used when handling incomplete CFGs.

(a) Source program	(b) SSA form
a ← 42;	v_1: 42
b ← a;	
c ← a + b;	v_2: $v_1 + v_1$
	v_3: 23
a ← c + 23;	v_4: $v_2 + v_3$
c ← a + d;	v_5: $v_4 + v_?$

Fig. 1. Example for local value numbering

```
writeVariable(variable, block, value):
  currentDef[variable][block] ← value

readVariable(variable, block):
  if currentDef[variable] contains block:
    # local value numbering
    return currentDef[variable][block]
  # global value numbering
  return readVariableRecursive(variable, block)
```

Algorithm 1. Implementation of local value numbering

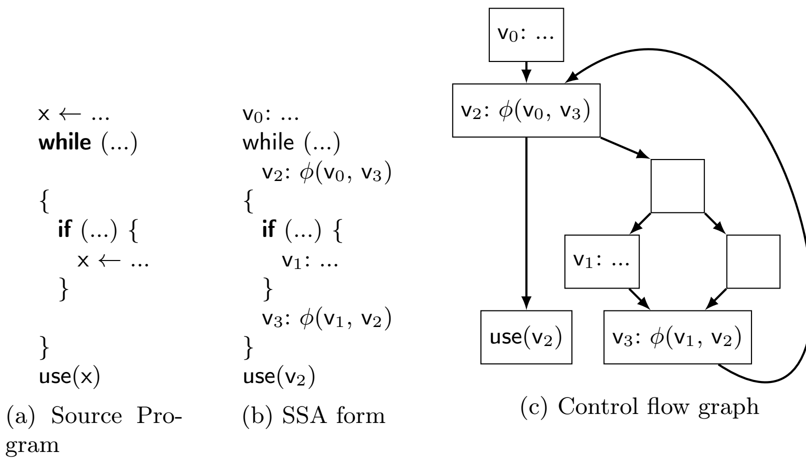
A sample program and the result of this process is illustrated in Figure 1. For the sake of presentation, we denote each SSA value by a name v_i.[1] In a concrete implementation, we would just refer to the representation of the expression. The names have no meaning otherwise, in particular they are not local variables in the sense of an imperative language.

[1] This acts like a `let` binding in a functional language. In fact, SSA form is a kind of functional representation [3].

Now, a problem occurs if a variable is read before it is assigned in a basic block. This can be seen in the example for the variable d and its corresponding value $\mathsf{v}_?$. In this case, d's definition is found on a path from the CFG's root to the current block. Moreover, multiple definitions in the source program may reach the same use. The next section shows how to extend local value numbering to handle these situations.

2.2 Global Value Numbering

If a block currently contains no definition for a variable, we recursively look for a definition in its predecessors. If the block has a single predecessor, just query it recursively for a definition. Otherwise, we collect the definitions from all predecessors and construct a ϕ function, which joins them into a single new value. This ϕ function is recorded as current definition in this basic block.



(a) Source Program (b) SSA form (c) Control flow graph

Fig. 2. Example for global value numbering

Looking for a value in a predecessor might in turn lead to further recursive look-ups. Due to loops in the program, those might lead to endless recursion. Therefore, *before recursing*, we first create the ϕ function without operands and record it as the current definition for the variable in the block. Then, we determine the ϕ function's operands. If a recursive look-up arrives back at the block, this ϕ function will provide a definition and the recursion will end. Algorithm 2 shows pseudocode to perform global value numbering. Its first condition will be used to handle incomplete CFGs, so for now assume it is always false.

Figure 2 shows this process. For presentation, the indices of the values v_i are assigned in the order in which the algorithm inserts them. We assume that the loop is constructed before x is read, i.e., v_0 and v_1 are recorded as definitions for x by local value numbering and only then the statement after the loop looks up x. As there is no definition for x recorded in the block after the loop, we perform a recursive look-up. The block has only a single predecessor, so no ϕ function is needed here. This predecessor is the loop header, which also has no definition

```
readVariableRecursive(variable, block):
  if block not in sealedBlocks:
    # Incomplete CFG
    val ← new Phi(block)
    incompletePhis[block][variable] ← val
  else if |block.preds| = 1:
    # Optimize the common case of one predecessor: No phi needed
    val ← readVariable(variable, block.preds[0])
  else:
    # Break potential cycles with operandless phi
    val ← new Phi(block)
    writeVariable(variable, block, val)
    val ← addPhiOperands(variable, val)
  writeVariable(variable, block, val)
  return val

addPhiOperands(variable, phi):
  # Determine operands from predecessors
  for pred in phi.block.preds:
    phi.appendOperand(readVariable(variable, pred))
  return tryRemoveTrivialPhi(phi)
```

Algorithm 2. Implementation of global value numbering

```
tryRemoveTrivialPhi(phi):
  same ← None
  for op in phi.operands:
    if op = same || op = phi:
      continue # Unique value or self−reference
    if same ≠ None:
      return phi # The phi merges at least two values: not trivial
    same ← op
  if same = None:
    same ← new Undef() # The phi is unreachable or in the start block
  users ← phi.users.remove(phi) # Remember all users except the phi itself
  phi.replaceBy(same) # Reroute all uses of phi to same and remove phi

  # Try to recursively remove all phi users, which might have become trivial
  for use in users:
    if use is a Phi:
      tryRemoveTrivialPhi(use)
  return same
```

Algorithm 3. Detect and recursively remove a trivial ϕ function

for x and has two predecessors. Thus, we place an operandless ϕ function v_2. Its first operand is the value v_0 flowing into the loop. The second operand requires further recursion. The ϕ function v_3 is created and gets its operands from its direct predecessors. In particular, v_2 placed earlier breaks the recursion.

Recursive look-up might leave redundant ϕ functions. We call a ϕ function v_ϕ *trivial* iff it just references itself and one other value v any number of times: $\exists v \in V : v_\phi : \phi(x_1, \ldots, x_n) \quad x_i \in \{v_\phi, v\}$. Such a ϕ function can be removed and the value v is used instead (see Algorithm 3). As a special case, the ϕ function might use no other value besides itself. This means that it is either unreachable or in the start block. We replace it by an undefined value.

Moreover, if a ϕ function could be successfully replaced, other ϕ functions using this replaced value might become trivial as well. For this reason, we apply this simplification recursively on all of these users.

This approach works for all acyclic language constructs. In this case, we can fill all predecessors of a block before processing it. The recursion will only search in already filled blocks. This ensures that we retrieve the latest definition for each variable from the predecessors. For example, in an if-then-else statement the block containing the condition can be filled before the then and else branches are processed. Accordingly, after the two branches are completed, the block joining the branches is filled. This approach also works when reading a variable after a loop has been constructed. But when reading a variable within a loop, which is under construction, some predecessors—at least the jump back to the head of the loop—are missing.

2.3 Handling Incomplete CFGs

We call a basic block *sealed* if no further predecessors will be added to the block. As only filled blocks may have successors, predecessors are always filled. Note that a sealed block is not necessarily filled. Intuitively, a filled block contains all its instructions and can provide variable definitions for its successors. Conversely, a sealed block may look up variable definitions in its predecessors as all predecessors are known.

```
sealBlock(block):
  for variable in incompletePhis[block]:
    addPhiOperands(variable, incompletePhis[block][variable])
  sealedBlocks.add(block)
```

Algorithm 4. Handling incomplete CFGs

But how to handle a look-up of a variable in an unsealed block, which has no current definition for this variable? In this case, we place an operandless ϕ function into the block and record it as proxy definition (see first case in Algorithm 2). Further, we maintain a set incompletePhis of these proxies per block. When later on a block gets sealed, we add operands to these ϕ functions (see Algorithm 4). Again, when the ϕ function is complete, we check whether it is trivial.

Sealing a block is an explicit action during IR construction. We illustrate how to incorporate this step by the example of constructing the while loop seen in Figure 3a. First, we construct the while header block and add a control flow edge from the while entry block to it. Since the jump from the body exit needs to be added later, we cannot seal the while header yet. Next, we create the body entry

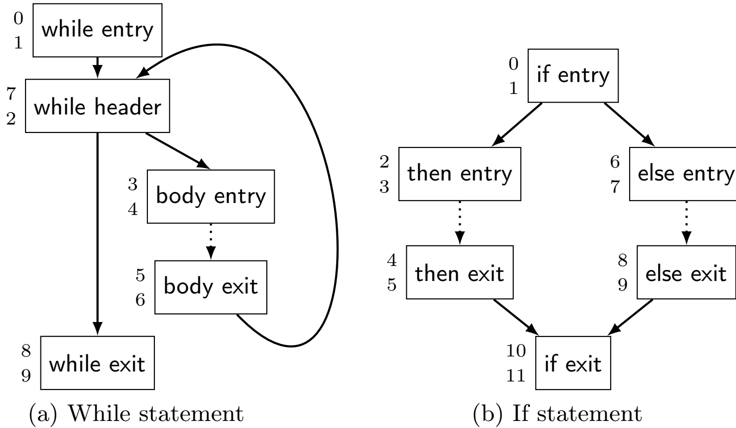


Fig. 3. CFG illustration of construction procedures. Dotted lines represent possible further code, while straight lines are normal control flow edges. Numbers next to a basic block denote the order of sealing (top) and filling (bottom).

and while exit blocks and add the conditional control flow from the while header to these two blocks. No further predecessors will be added to the body entry block, so we seal it now. The while exit block might get further predecessors due to break instructions in the loop body. Now we fill the loop body. This might include further inner control structures, like an if shown in Figure 3b. Finally, they converge at the body exit block. All the blocks forming the body are sealed at this point. Now we add the edge back to the while header and seal the while header. The loop is completed. In the last step, we seal the while exit block and then continue IR construction with the source statement after the while loop.

3 Optimizations

3.1 On-the-Fly Optimizations

In the previous section, we showed that we optimize trivial ϕ functions as soon as they are created. Since ϕ functions belong to the IR, this means we employ an IR optimization during SSA construction. Obviously, this is not possible with all optimizations. In this section, we elaborate what kind of IR optimizations can be performed during SSA construction and investigate their effectiveness.

We start with the question whether the optimization of ϕ functions removes all trivial ϕ functions. As already mentioned in Section 2.2, we recursively optimize all ϕ functions that have used a removed trivial ϕ function. Since a successful optimization of a ϕ function can only render ϕ functions trivial that use the former one, this mechanism enables us to optimize all trivial ϕ functions. In Section 4, we show that, for reducible CFGs, this is equivalent to the construction of minimal SSA form.

Since our approach may lead to a significant number of triviality checks, we use the following caching technique to speed such checks up: While constructing

a ϕ function, we record the first two distinct operands that are also distinct from the ϕ function itself. These operands act as witnesses for the non-triviality of the ϕ function. When a new triviality check occurs, we compare the witnesses again. If they remain distinct from each other and the ϕ function, the ϕ function is still non-trivial. Otherwise, we need to find a new witness. Since all operands until the second witness are equal to the first one or to the ϕ function itself, we only need to consider operands that are constructed after both old witnesses. Thus, the technique speeds up multiple triviality checks for the same ϕ function.

There is a more simple variant of the mechanism that results in minimal SSA form for most but not all cases: Instead of optimizing the users of a replaced ϕ function, we optimize the unique operand. This variant is especially interesting for IRs, which do not inherently provide a list of users for each value.

The optimization of ϕ functions is only one out of many IR optimizations that can be performed during IR construction. In general, our SSA construction algorithm allows to utilize conservative IR optimizations, i.e., optimizations that require only local analysis. These optimizations include:

Arithmetic Simplification. All IR node constructors perform peephole optimizations and return simplified nodes when possible. For instance, the construction of a subtraction `x−x` always yields the constant 0.

Common Subexpression Elimination. This optimization reuses existing values that are identified by local value numbering.

Constant Folding. This optimization evaluates constant expressions at compile time, e.g., `2*3` is optimized to `6`.

Copy Propagation. This optimization removes unnecessary assignments to local variables, e.g., `x = y`. In SSA form, there is no need for such assignments, we can directly use the value of the right-hand side.

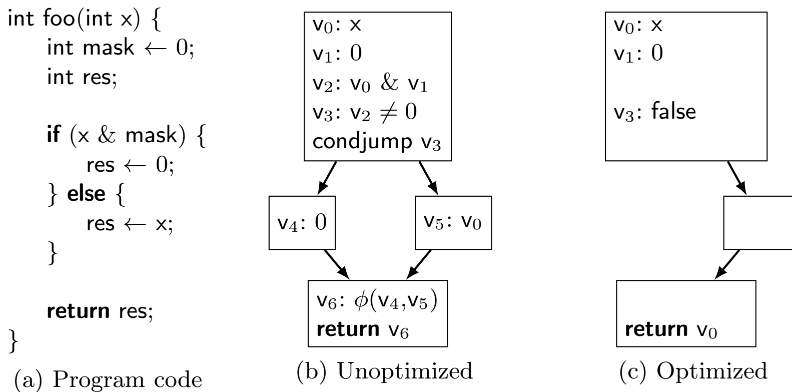


Fig. 4. The construction algorithm allows to perform conservative optimization during SSA construction. This may also affect control flow, which in turn could lead to a reduced number of ϕ functions.

Figure 4 shows the effectiveness of these optimizations. We want to construct SSA form for the code fragment shown in Figure 4a. Without on-the-fly optimizations, this results in the SSA form program shown in Figure 4b. The first difference with enabled optimizations occurs during the construction of the value v_2. Since a bitwise conjunction with zero always yields zero, the arithmetic simplification triggers and simplifies this value. Moreover, the constant value zero is already available. Thus, the common subexpression elimination reuses the value v_1 for the value v_2. In the next step, constant propagation folds the comparison with zero to `false`. Since the condition of the conditional jump is `false`, we can omit the `then` part.[2] Within the `else` part, we perform copy propagation by registering v_0 as value for res. Likewise, v_6 vanishes and in the end the function returns v_0. Figure 4c shows the optimized SSA form program.

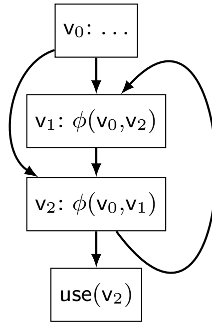
The example demonstrates that on-the-fly optimizations can further reduce the number of ϕ functions. This can even lead to fewer ϕ functions than required for minimal SSA form according to Cytron et al.'s definition.

3.2 Minimal SSA Form for Arbitrary Control Flow

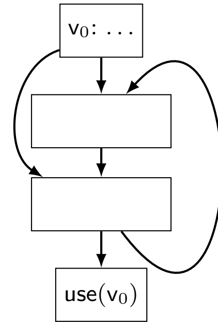
So far, our SSA construction algorithm does not construct minimal SSA form in the case of irreducible control flow. Figure 5b shows the constructed SSA form for the program shown in Figure 5a. Figure 5c shows the corresponding minimal SSA form—as constructed by Cytron et al.'s algorithm. Since there is only one definition for the variable x, the ϕ functions constructed by our algorithm are superfluous.

```
x ← ...

if (...)
    goto second_loop_entry

while (...) {
    ...
second_loop_entry:
    ...
}

use(x)
```

(a) Program with irreducible control flow



(b) Intermediate representation with our SSA construction algorithm



(c) Intermediate representation in minimal SSA form

Fig. 5. Our SSA construction algorithm can produce extraneous ϕ functions in presence of irreducible control flow. We remove these ϕ functions afterwards.

In general, a non-empty set P of ϕ functions is *redundant* iff the ϕ functions just reference each other or one other value v: $\exists v \in V\ \forall v_i \in P\colon v_i\colon \phi(x_1, \ldots, x_n)\ x_i \in P \cup \{v\}$. In particular, when P contains only a single ϕ function, this

[2] This is only possible because the `then` part contains no labels.

```
proc removeRedundantPhis(phiFunctions):
    sccs ← computePhiSCCs(inducedSubgraph(phiFunctions))
    for scc in topologicalSort(sccs):
        processSCC(scc)

proc processSCC(scc):
    if len(scc) = 1: return # we already handled trivial φ functions

    inner ← set()
    outerOps ← set()
    for phi in scc:
        isInner ← True
        for operand in phi.getOperands():
            if operand not in scc:
                outerOps.add(operand)
                isInner ← False
        if isInner:
            inner.add(phi)

    if len(outerOps) = 1:
        replaceSCCByValue(scc, outerOps.pop())
    else if len(outerOps) > 1:
        removeRedundantPhis(inner)
```

Algorithm 5. Remove superfluous ϕ functions in case of irreducible data flow

definition degenerates to the definition of a trivial ϕ function given in Section 2.2. We show that each set of redundant ϕ functions P contains a strongly connected component (SCC) that is also redundant. This implies a definition of minimality that is independent of the source program and more strict than the definition by Cytron et al. [10].

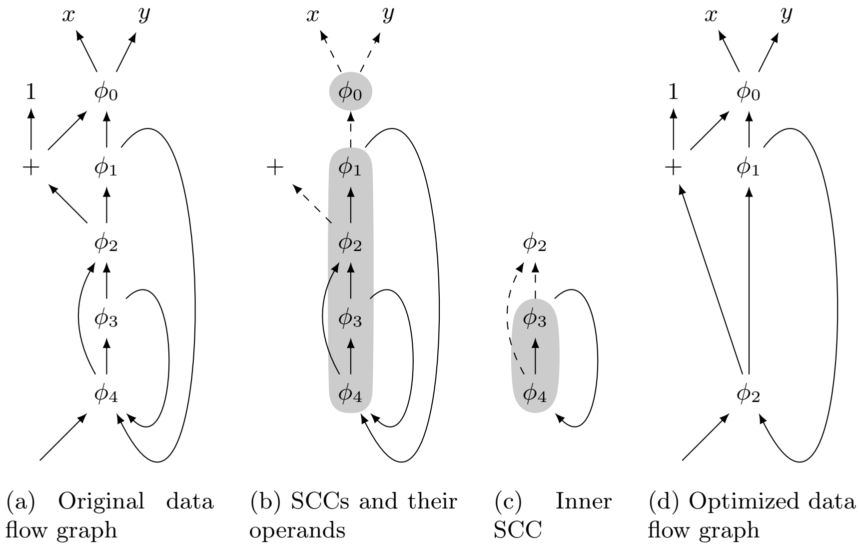
Lemma 1. *Let P be a redundant set of ϕ functions with respect to v. Then there is a strongly connected component $S \subseteq P$ that is also redundant.*

Proof. Let P' be the condensation of P, i.e., each SCC in P is contracted into a single node. The resulting P' is acyclic [11]. Since P' is non-empty, it has a leaf s'. Let S be the SCC, which corresponds to s'. Since s' is a leaf, the ϕ functions in S only refer to v or other ϕ functions in S. Hence, S is a redundant SCC. □

Algorithm 5 exploits Lemma 1 to remove superfluous ϕ functions. The function removeRedundantPhis takes a set of ϕ functions and computes the SCCs of their induced subgraph. Figure 6b shows the resulting SCCs for the data flow graph in Figure 6a. Each dashed edge targets a value that does not belong to the SCC of its source. We process the SCCs in topological order to ensure that used values outside of the current SCC are already contracted. In our example, this means we process the SCC containing only ϕ_0 first. Since ϕ_0 is the only ϕ function within its SCC, we already handled it during removal of trivial ϕ functions. Thus, we skip this SCC.

For the next SCC containing ϕ_1–ϕ_4, we construct two sets: The set inner contains all ϕ functions having operands solely within the SCC. The set $\mathsf{outerOps}$ contains all ϕ function operands that do not belong to the SCC. For our example, $\mathsf{inner} = \{\phi_3, \phi_4\}$ and $\mathsf{outerOps} = \{\phi_0, +\}$.

If the $\mathsf{outerOps}$ set is empty, the corresponding basic blocks must be unreachable and we skip the SCC. In case that the $\mathsf{outerOps}$ set contains exactly one value, all ϕ functions within the SCC can only get this value. Thus, we replace the SCC by the value. If the $\mathsf{outerOps}$ set contains multiple values, all ϕ functions that have an operand outside the SCC are necessary. We collected the remaining ϕ functions in the set inner. Since our SCC can contain multiple inner SCCs, we recursively perform the procedure with the inner ϕ functions. Figure 6c shows the inner SCC for our example. In the recursive step, we replace this SCC by ϕ_2. Figure 6d shows the resulting data flow graph.



(a) Original data flow graph

(b) SCCs and their operands

(c) Inner SCC

(d) Optimized data flow graph

Fig. 6. Algorithm 5 detects the inner SCC spanned by ϕ_3 and ϕ_4. This SCC represents the same value. Thus, it gets replaced by ϕ_2.

Performing on-the-fly optimizations (Section 3.1) can also lead to irreducible data flow. For example, let us reconsider Figure 2 described in Section 2.2. Assume that both assignments to x are copies from some other variable y. If we now perform copy propagation, we obtain two ϕ functions v_2: $\phi(\mathsf{v}_0, \mathsf{v}_3)$ and v_3: $\phi(\mathsf{v}_0, \mathsf{v}_2)$ that form a superfluous SCC. Note that this SCC also will appear after performing copy propagation on a program constructed with Cytron's algorithm. Thus, Algorithm 5 is also applicable to other SSA construction algorithms. Finally, Algorithm 5 can be seen as a generalization of the local simplifications by Aycock and Horspool (see Section 7).

3.3 Reducing the Number of Temporary ϕ Functions

The presented algorithm is structurally simple and easy to implement. Though, it might produce many temporary ϕ functions, which get removed right away during IR construction. In the following, we describe two extensions to the algorithm, which aim at reducing or even eliminating these temporary ϕ functions.

Marker Algorithm. Many control flow joins do not need a ϕ function for a variable. So instead of placing a ϕ function before recursing, we just mark the block as visited. If we reach this block again during recursion, we will place a ϕ function there to break the cycle. After collecting the definitions from all predecessors, we remove the marker and place a ϕ function (or reuse the one placed by recursion) if we found different definitions. Using this technique, no temporary ϕ functions are placed in acyclic data-flow regions. Temporary ϕ functions are only generated in data-flow loops and might be determined as unnecessary later on.

SCC Algorithm. While recursing we use Tarjan's algorithm to detect data-flow cycles, i.e., SCCs [17]. If only a unique value enters the cycle, no ϕ functions will be necessary. Otherwise, we place a ϕ function into every basic block, which has a predecessor from outside the cycle. In order to add operands to these ϕ functions, we apply the recursive look-up again as this may require placement of further ϕ functions. This mirrors the algorithm for removing redundant cycles of ϕ functions described in Section 3.2. In case of recursing over sealed blocks, the algorithm only places necessary ϕ functions. The next section gives a formal definition of a necessary ϕ function and shows that an algorithm that only places necessary ϕ functions produces minimal SSA form.

4 Properties of Our Algorithm

Because most optimizations treat ϕ functions as uninterpreted functions, it is beneficial to place as few ϕ functions as possible. In the rest of this section, we show that our algorithm does not place dead ϕ functions and constructs minimal (according to Cytron et al.'s definition) SSA form for programs with reducible control flow.

Pruned SSA Form. A program is said to be in pruned SSA form [7] if each ϕ function (transitively) has at least one non-ϕ user. We only create ϕ functions on demand when a user asks for it: Either a variable being read or another ϕ function needing an argument. So our construction naturally produces a program in pruned SSA form.

Minimal SSA Form. Minimal SSA form requires that ϕ functions for a variable v only occur in basic blocks where different definitions of v meet for the first time. Cytron et al.'s formal definition is based on the following two terms:

Definition 1 (Path Convergence). *Two non-null paths* $X_0 \to^+ X_J$ *and* $Y_0 \to^+ Y_K$ *are said to* converge *at a block* Z *iff the following conditions hold:*

$$X_0 \neq Y_0; \tag{1}$$

$$X_J = Z = Y_K; \tag{2}$$

$$(X_j = Y_k) \Rightarrow (j = J \vee k = K). \tag{3}$$

Definition 2 (Necessary ϕ Function). *A ϕ function for variable v is necessary in block Z iff two non-null paths $X \rightarrow^+ Z$ and $Y \rightarrow^+ Z$ converge at a block Z, such that the blocks X and Y contain assignments to v.*

A program with only necessary ϕ functions is in *minimal SSA form*. The following is a proof that our algorithm presented in Section 2 with the simplification rule for ϕ functions produces minimal SSA form for reducible programs.

We say a block A *dominates* a block B if every path from the entry block to B passes through A. We say A *strictly dominates* B if A dominates B and $A \neq B$. Each block C except the entry block has a unique immediate dominator $idom(C)$, i.e., a strict dominator of C, which does not dominate any other strict dominator of C. The dominance relation can be represented as a tree whose nodes are the basic blocks with a connection between immediately dominating blocks.

Definition 3 (Reducible Flow Graph, Hecht and Ullmann [12]). *A (control) flow graph G is reducible iff for each cycle C of G there is a node of C, which dominates all other nodes in C.*

We now assume that our construction algorithm is finished and has produced a program with a reducible CFG. We observe that the simplification rule tryRemoveTrivialPhi of Algorithm 3 was applied at least once to each ϕ function with its current arguments. This is because we apply the rule each time a ϕ function's parameters are set for the first time. In the case that a simplification of another operation leads to a change of parameters, the rule is applied again. Furthermore, our construction algorithm fulfills the following property:

Definition 4 (SSA Property). *In an SSA-form program a path from a definition of an SSA value for variable v to its use cannot contain another definition or ϕ function for v. The use of the operands of ϕ function happens in the respective predecessor blocks not in the ϕ's block itself.*

The SSA property ensures that only the "most recent" SSA value of a variable v is used. Furthermore, it forbids multiple ϕ functions for one variable in the same basic block.

Lemma 2. *Let p be a ϕ function in a block P. Furthermore, let q in a block Q and r in a block R be two operands of p, such that p, q and r are pairwise distinct. Then at least one of Q and R does not dominate P.*

Proof. Assume that Q and R dominate P, i.e., every path from the start block to P contains Q and R. Since immediate dominance forms a tree, Q dominates R or R dominates Q. Without loss of generality, let Q dominate R. Furthermore, let S be the corresponding predecessor block of P where p is using q. Then there is a path from the start block crossing Q then R and S. This violates the SSA property. □

Lemma 3. *If a ϕ function p in a block P for a variable v is unnecessary, but non-trivial, then it has an operand q in a block Q, such that q is an unnecessary ϕ function and Q does not dominate P.*

Proof. The node p must have at least two different operands r and s, which are not p itself. Otherwise, p is trivial. They can either be:

- The result of a direct assignment to v.
- The result of a necessary ϕ function r'. This however means that r' was reachable by at least two different direct assignments to v. So there is a path from a direct assignment of v to p.
- Another unnecessary ϕ function.

Assume neither r in a block R nor s in a block S is an unnecessary ϕ function. Then a path from an assignment to v in a block V_r crosses R and a path from an assignment to v in a block V_s crosses S. They converge at P or earlier. Convergence at P is not possible because p is unnecessary. An earlier convergence would imply a necessary ϕ function at this point, which violates the SSA property.

So r or s must be an unnecessary ϕ function. Without loss of generality, let this be r.

If R does not dominate P, then r is the sought-after q. So let R dominate P. Due to Lemma 2, S does not dominate P. Employing the SSA property, $r \neq p$ yields $R \neq P$. Thus, R strictly dominates P. This implies that R dominates all predecessors of P, which contain the uses of p, especially the predecessor S' that contains the use of s. Due to the SSA property, there is a path from S to S' that does not contain R. Employing R dominates S' this yields R dominates S.

Now assume that s is necessary. Let X contain the most recent definition of v on a path from the start block to R. By Definition 2 there are two definitions of v that render s necessary. Since R dominates S, the SSA property yields that one of these definitions is contained in a block Y on a path $R \rightarrow^+ S$. Thus, there are paths $X \rightarrow^+ P$ and $Y \rightarrow^+ P$ rendering p necessary. Since this is a contradiction, s is unnecessary and the sought-after q. □

Theorem 1. *A program in SSA form with a reducible CFG G without any trivial ϕ functions is in minimal SSA form.*

Proof. Assume G is not in minimal SSA form and contains no trivial ϕ functions. We choose an unnecessary ϕ function p. Due to Lemma 3, p has an operand q, which is unnecessary and does not dominate p. By induction q has an unnecessary ϕ function as operand as well and so on. Since the program only has a finite number of operations, there must be a cycle when following the q chain. A cycle in the ϕ functions implies a cycle in G. As G is reducible, the control flow cycle contains one entry block, which dominates all other blocks in the cycle. Without loss of generality, let q be in the entry block, which means it dominates p. Therefore, our assumption is wrong and G is either in minimal SSA form or there exist trivial ϕ functions. □

Because our construction algorithm will remove all trivial ϕ functions, the resulting IR must be in minimal SSA form for reducible CFGs.

4.1 Time Complexity

We use the following parameters to provide a precise worst-case complexity for our construction algorithm:

- B denotes the number of basic blocks.
- E denotes the number of CFG edges.
- P denotes the program size.
- V denotes the number of variables in the program.

We start our analysis with the simple SSA construction algorithm presented in Section 2.3. In the worst case, SSA construction needs to insert $\Theta(B)$ ϕ functions with $\Theta(E)$ operands for each variable. In combination with the fact the construction of SSA form *within* all basic block is in $\Theta(P)$, this leads to a lower bound of $\Omega(P + (B + E) \cdot V)$.

We show that our algorithm matches this lower bound, leading to a worst-case complexity of $\Theta(P + (B + E) \cdot V)$. Our algorithm requires $\Theta(P)$ to fill all basic blocks. Due to our variable mapping, we place at most $\mathcal{O}(B \cdot V)$ ϕ functions. Furthermore, we perform at most $\mathcal{O}(E \cdot V)$ recursive requests at block predecessors. Altogether, this leads to a worst-case complexity of $\Theta(P + (B + E) \cdot V)$.

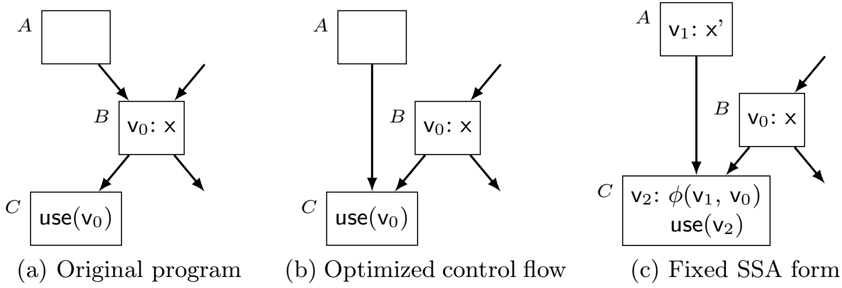
Next, we consider the on-the-fly optimization of ϕ functions. Once we optimized a ϕ function, we check whether we can optimize the ϕ functions that use the former one. Since our algorithm constructs at most $B \cdot V$ ϕ functions, this leads to $\mathcal{O}(B^2 \cdot V^2)$ checks. One check needs to compare at most $\mathcal{O}(B)$ operands of the ϕ function. However, using the caching technique described in Section 3.1, the number of checks performed for each ϕ functions amortizes the time for checking the corresponding ϕ function. Thus, the on-the-fly optimization of ϕ functions can be performed in $\mathcal{O}(B^2 \cdot V^2)$.

To obtain minimal SSA form, we need to contract SCCs that pass the same value. Since we consider only ϕ functions and their operands, the size of the SCCs is in $\mathcal{O}((B + E) \cdot V)$. Hence, computing the SCCs for the data flow graph is in $\mathcal{O}(P + (B + E) \cdot V)$. Computing the sets `inner` and `outer` consider each ϕ function and its operands exactly once. Thus, it is also in $\mathcal{O}((B + E) \cdot V)$. The same argument applies for the contraction of a SCC in case there is only one outer operand. In the other case, we iterate the process with a subgraph that is induced by a proper subset of the nodes in our SCC. Thus, we need at most $B \cdot V$ iterations. In total, this leads to a time complexity in $\mathcal{O}(P + B \cdot (B + E) \cdot V^2)$ for the contraction of SCCs.

5 Other Applications of the Algorithm

5.1 SSA Reconstruction

Some transformations like live range splitting, rematerialization or jump threading introduce additional definitions for an SSA value. Because this violates the SSA property, SSA has to be *reconstructed*. For the latter transformation, we run



(a) Original program (b) Optimized control flow (c) Fixed SSA form

Fig. 7. We assume that A always jumps via B to C. Adjusting A's jump requires SSA reconstruction.

```
int f(int x) {
  int a;
  if (x = 0) {
    a ← 23;
  } else {
    a ← 42;
  }

  return a;
}
```

(a) Source program

```
int f(int x) {

  if (x = 0) {

  } else {

  }
  v0: φ(23, 42)
  return v0
}
```

(b) SSA form

```
f(x : int, ret : int → ⊥) → ⊥ {
  let then := () → ⊥
        next(23)
      else := () → ⊥
        next(42)
      next := (a : int) → ⊥
        ret(a)
  in
    branch(x = 0, then, else)
}
```

(c) CPS version

Fig. 8. An imperative program in SSA form and converted to a functional CPS program

through an example in order to demonstrate how our algorithm can be leveraged for SSA reconstruction.

Consider an analysis determined that the basic block B in Figure 7a always branches to C when entered from A. Thus, we let A directly jump to C (Figure 7b). However, definition v_0 does not dominate its use anymore. We can fix this issue by first inserting a copy v_1 of v_0 into A. Then, we invoke writeVariable(V, A, x') and writeVariable(V, B, x) while V is just some handle to refer to the set of definitions, which represent the "same variable". Next, a call to readVariableRecursive(V, C) adds a necessary ϕ function and yields v_2 as new definition, which we can use to update v_0's original use (Figure 7c).

In particular, for jump threading, it is desirable to not depend on dominance calculations—as opposed to Cytron et al.'s algorithm: Usually, several iterations of jump threading are performed until no further improvements are possible. Since jump threading alters control flow in a non-trivial way, each iteration would require a re-computation of the dominance tree.

Note that SSA reconstruction always runs on complete CFGs. Hence, sealing and issues with non-sealed basic blocks do not arise in this setting.

5.2 CPS Construction

CPS is a functional programming technique that captures control flow in *continuations*. Continuations are functions, which never return. Instead, each continuation invokes another continuation in tail position.

SSA form can be considered as a restricted form of CPS [13,3]. Our algorithm is also suitable to directly convert an imperative program into a functional CPS program without the need for a third program representation. Instead of ϕ functions, we have to place parameters in local functions. Instead of adding operands to ϕ functions, we add arguments to the predecessors' calls. Like when constructing SSA form, on-the-fly optimizations, as described in Section 3.1, can be exploited to shrink the program. Figure 8 demonstrates CPS construction.

In a CPS program, we cannot simply remove a ϕ function. Rather, we would have to eliminate a parameter, fix its function's type and adjust all users of this function. As this set of transformations is expensive, it is worthwhile to not introduce unnecessary parameters in the first place and therefore use the extensions described in Section 3.3.

6 Evaluation

6.1 Comparison to Cytron et al.'s Algorithm

We implemented the algorithm presented in this paper in LLVM 3.1 [2] to compare it against an existing, highly-tuned implementation of Cytron et al.'s algorithm. Table 1 shows the number of constructed instructions for both algorithms. Since LLVM first models accesses to local variables with load and stores instructions, we also denoted the instructions immediately before SSA construction.

Table 1. Comparison of instruction counts of LLVM's normal implementation and our algorithm. *#mem* are alloca, load and store instructions. *Insn ratio* is the quotient between *#insn* of *before SSA construction* and *marker*.

Bench-	Before SSA Constr.			After SSA Constr.			Marker			Insn
mark	#insn	#mem	#phi	#insn	#mem	#phi	#insn	#mem	#phi	ratio
gzip	12,038	5,480	82	9,187	2,117	594	9,179	2,117	594	76%
vpr	40,701	21,226	129	27,155	6,608	1,201	27,092	6,608	1,201	67%
gcc	516,537	206,295	2,230	395,652	74,736	12,904	393,554	74,683	12,910	76%
mcf	3,988	2,173	14	2,613	658	154	2,613	658	154	66%
crafty	44,891	18,804	116	36,050	8,613	1,466	36,007	8,613	1,466	80%
parser	30,237	14,542	100	20,485	3,647	1,243	20,467	3,647	1,243	68%
perlbmk	185,576	86,762	1,764	140,489	37,599	5,840	140,331	37,517	5,857	76%
gap	201,185	86,157	4,074	149,755	29,476	9,325	149,676	29,475	9,326	74%
vortex	126,097	65,245	990	88,257	25,656	2,739	88,220	25,661	2,737	70%
bzip2	8,605	4,170	9	6,012	1,227	359	5,993	1,227	359	70%
twolf	76,078	38,320	246	58,737	18,376	2,849	58,733	18,377	2,849	77%
Sum	1,245,933	549,174	9,754	934,392	208,713	38,674	931,865	208,583	38,696	75%

In total, SSA construction reduces the number of instructions by 25%, which demonstrates the significant overhead of the temporary non-SSA IR.

Comparing the number of constructed instructions, we see small differences between the results of LLVM's and our SSA-construction algorithm. One reason for the different number of ϕ functions is the removal of redundant SCCs: 3 (out of 11) non-trivial SCCs do not originate from irreducible control flow and are not removed by Cytron et al.'s algorithm. The remaining difference in the number of ϕ functions and memory instructions stems from minor differences in handling unreachable code. In most benchmarks, our algorithm triggers LLVM's constant folding more often, and thus further reduces the overall instruction count. Exploiting more on-the-fly optimizations like common subexpression elimination as described in Section 3.1 would shrink the overall instruction count even further.

Table 2. Executed instructions for Cytron et al.'s algorithm and the Marker algorithm

Benchmark	Cytron et al.	Marker	$\frac{\text{Marker}}{\text{Cytron et al.}}$
164.gzip	969,233,677	967,798,047	99.85%
175.vpr	3,039,801,575	3,025,286,080	99.52%
176.gcc	25,935,984,569	26,009,545,723	100.28%
181.mcf	722,918,540	722,507,455	99.94%
186.crafty	3,653,881,430	3,632,605,590	99.42%
197.parser	2,084,205,254	2,068,075,482	99.23%
253.perlbmk	12,246,953,644	12,062,833,383	98.50%
254.gap	8,358,757,289	8,339,871,545	99.77%
255.vortex	7,841,416,740	7,845,699,772	100.05%
256.bzip2	569,176,687	564,577,209	99.19%
300.twolf	6,424,027,368	6,408,289,297	99.76%
Sum	71,846,356,773	71,647,089,583	99.72%

For the runtime comparison of both benchmarks, we count the number of executed x86 instructions. Table 2 shows the counts collected by the valgrind instrumentation tool. While the results vary for each benchmark, the marker algorithm needs slightly (0.28%) fewer instructions in total. All measurements were performed on a Core i7-2600 CPU with 3.4 GHz, by compiling the C-programs of the SPEC CINT2000 benchmark suite.

6.2 Effect of On-the-Fly Optimization

We also evaluated the effects of performing on-the-fly optimizations (as described in Section 3.1) on the speed and quality of SSA construction. Our libFirm [1] compiler library has always featured a variant of the construction algorithms described in this paper.

There are many optimizations interweaved with the SSA construction. The results are shown in Table 3. Enabling on-the-fly optimizations during construction results in an increased construction time of 0.84 s, but the resulting graph

Table 3. Effect of on-the-fly optimizations on construction time and IR size

Benchmark	No On-the-fly Optimizations			On-the-fly Optimizations		
	Time	IR Time	Instructions	Time	IR Time	Instructions
164.gzip	1.38 s	0.03 s	10,520	1.34 s	0.05 s	9,255
175.vpr	3.80 s	0.08 s	28,506	3.81 s	0.12 s	26,227
176.gcc	59.80 s	0.61 s	408,798	59.16 s	0.91 s	349,964
181.mcf	0.57 s	0.02 s	2,631	0.60 s	0.03 s	2,418
186.crafty	7.50 s	0.13 s	42,604	7.32 s	0.18 s	37,384
197.parser	5.54 s	0.06 s	19,900	5.55 s	0.09 s	18,344
253.perlbmk	25.10 s	0.29 s	143,039	24.79 s	0.41 s	129,337
254.gap	18.06 s	0.25 s	152,983	17.87 s	0.34 s	132,955
255.vortex	17.66 s	0.35 s	98,694	17.54 s	0.45 s	92,416
256.bzip2	1.03 s	0.01 s	6,623	1.02 s	0.02 s	5,665
300.twolf	7.24 s	0.18 s	60,445	7.20 s	0.27 s	55,346
Sum	147.67 s	2.01 s	974,743	146.18 s	2.86 s	859,311

has only 88.2% the number of nodes. This speeds up later optimizations resulting in an 1.49 s faster overall compilation.

6.3 Conclusion

The algorithm has been shown to be as fast as the Cytron et al.'s algorithm in practice. However, if the algorithm is combined with on-the-fly optimizations, the overall compilation time is reduced. This makes the algorithm an interesting candidate for just-in-time compilers.

7 Related Work

SSA form was invented by Rosen, Wegman, and Zadeck [15] and became popular after Cytron et al. [10] presented an efficient algorithm for constructing it. This algorithm can be found in all textbooks presenting SSA form and is used by the majority of compilers. For each variable, the iterated dominance frontiers of all blocks containing a definition is computed. Then, a rewrite phase creates new variable numbers, inserts ϕ functions and patches users. The details necessary for this paper were already discussed in Section 1.

Choi et al. [7] present an extension to the previous algorithm that constructs minimal and pruned SSA form. It computes liveness information for each variable v and inserts a ϕ function for v only if v is live at the corresponding basic block. This technique can also be applied to other SSA construction algorithms to ensure pruned SSA form, but comes along with the costs of computing liveness information.

Briggs et al. [6] present semi-pruned SSA form, which omits the costly liveness analysis. However, they only can prevent the creation of dead ϕ functions for variables that are local to a basic block.

Sreedhar and Gao [16] present a data structure, called DJ graph, that enhances the dominance tree with edges from the CFG. Compared to computing iterated dominance frontiers for each basic block, this data structure is only linear in the program size and allows to compute the blocks where ϕ functions need to be placed in linear time per variable. This gives an SSA construction algorithm with cubic worst-case complexity in the size of the source program.

There are also a range of construction algorithms, which aim for simplicity instead. Brandis and Mössenböck [5] present a simple SSA construction algorithm that directly works on the AST like our algorithm. However, their algorithm is restricted to structured control flow (no gotos) and does not construct pruned SSA form. Click and Paleczny [8] describe a graph-based SSA intermediate representation used in the Java HotSpot server compiler [14] and an algorithm to construct this IR from the AST. Their algorithm is in the spirit as the one of Brandis and Mössenböck and thus does construct neither pruned nor minimal SSA form. Aycock and Horspool present an SSA construction algorithm that is designed for simplicity [4]. They place a ϕ function for each variable at each basic block. Afterwards they employ the following rules to remove ϕ functions:

1. Remove ϕ functions of the form $v_i = \phi(v_i, \ldots, v_i)$.
2. Substitute ϕ functions of the form $v_i = \phi(v_{i_1}, \ldots, v_{i_n})$ with $i_1, \ldots, i_n \in \{i, j\}$ by v_j.

This results in minimal SSA form for reducible programs. The obvious drawback of this approach is the overhead of inserting ϕ functions at each basic block. This also includes basic blocks that are prior to every basic block that contains a real definition of the corresponding variable.

8 Conclusions

In this paper, we presented a novel, simple, and efficient algorithm for SSA construction. In comparison to existing algorithms it has several advantages: It does not require other analyses and transformations to produce minimal (on reducible CFGs) and pruned SSA form. It can be directly constructed from the source language without passing through a non-SSA CFG. It is well suited to perform several standard optimizations (constant folding, value numbering, etc.) already during SSA construction. This reduces the footprint of the constructed program, which is important in scenarios where compilation time is of the essence. After IR construction, a post pass ensures minimal SSA form for arbitrary control flow. Our algorithm is also useful for SSA reconstruction where, up to now, standard SSA construction algorithms where not directly applicable. Finally, we proved that our algorithm always constructs pruned and minimal SSA form.

In terms of performance, a non-optimized implementation of our algorithm is slightly faster than the highly-optimized implementation of Cytron et al.'s algorithm in the LLVM compiler, measured on the SPEC CINT2000 benchmark suite. We expect that after fine-tuning our implementation, we can improve the performance even more.

Acknowledgments. We thank the anonymous reviewers for their helpful comments. This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89).

References

1. libFirm – The FIRM intermediate representation library, `http://libfirm.org`
2. The LLVM compiler infrastructure project, `http://llvm.org`
3. Appel, A.W.: SSA is functional programming. SIGPLAN Notices 33(4), 17–20 (1998)
4. Aycock, J., Horspool, N.: Simple Generation of Static Single-Assignment Form. In: Watt, A. (ed.) CC 2000. LNCS, vol. 1781, pp. 110–125. Springer, Heidelberg (2000)
5. Brandis, M.M., Mössenböck, H.: Single-pass generation of static single-assignment form for structured languages. ACM Trans. Program. Lang. Syst. 16(6), 1684–1698 (1994)
6. Briggs, P., Cooper, K.D., Harvey, T.J., Simpson, L.T.: Practical improvements to the construction and destruction of static single assignment form. Softw. Pract. Exper. 28(8), 859–881 (1998)
7. Choi, J.D., Cytron, R., Ferrante, J.: Automatic construction of sparse data flow evaluation graphs. In: Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1991, pp. 55–66. ACM, New York (1991)
8. Click, C., Paleczny, M.: A simple graph-based intermediate representation. In: Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations, IR 1995, pp. 35–49. ACM, New York (1995)
9. Cocke, J.: Programming languages and their compilers: Preliminary notes. Courant Institute of Mathematical Sciences, New York University (1969)
10. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. TOPLAS 13(4), 451–490 (1991)
11. Eswaran, K.P., Tarjan, R.E.: Augmentation problems. SIAM J. Comput. 5(4), 653–665 (1976)
12. Hecht, M.S., Ullman, J.D.: Characterizations of reducible flow graphs. J. ACM 21(3), 367–375 (1974)
13. Kelsey, R.A.: A correspondence between continuation passing style and static single assignment form. SIGPLAN Not. 30, 13–22 (1995)
14. Paleczny, M., Vick, C., Click, C.: The Java HotSpot™server compiler. In: Symposium on Java™ Virtual Machine Research and Technology Symposium, JVM 2001, pp. 1–12. USENIX Association, Berkeley (2001)
15. Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Global value numbers and redundant computations. In: Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1988, pp. 12–27. ACM Press, New York (1988)
16. Sreedhar, V.C., Gao, G.R.: A linear time algorithm for placing ϕ-nodes. In: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1995, pp. 62–73. ACM, New York (1995)
17. Tarjan, R.E.: Depth-first search and linear graph algorithms. SIAM Journal Computing 1(2), 146–160 (1972)