Partial SSA form: compact representation for programs with indirect memory operations

D.S. Gurchenkov, P.E. Pavlov, E.M. Baskakov

Abstract. The paper presents an improvement over traditional SSA form, called partial SSA that features only partial translation of a program into the single-assignment state. Partial SSA is more compact than the traditionally used full SSA while it suits well most program optimization algorithms. The paper introduces formal quality criteria for this kind of internal representation and presents a translation algorithm (based on the code motion principles) that produces the internal representation satisfying these criteria. Proofs of correctness and optimality are also given.

1. CFG-based program representation

We use an internal representation of a compiled program (IR) in the form of a control flow graph (CFG), which is a directed graph representing a single procedure (many papers treat “program” and “procedure” as synonyms). The basic entity is a statement $v = op(a_1, \ldots, a_N)$, where $v$ is a variable name, $op$ stands for an operation symbol, $a_i$ stands for either variables or constants. A set of all program statements is denoted as $Stms$.

For a statement $s$ of the form $v = op(a_1, \ldots, a_N)$ we say that $s$ is a definition of $v$ (there may be more than one definition for a given $v$), and $s$ is the use of each of $a_1, \ldots, a_N$.

The control flow is represented as the function $succ: Stms \rightarrow 2^{Stms}$. It is assumed that a procedure has exactly one starting node, denoted as $\text{start}$, such that $\forall s \in Stms$, $\text{start} \not\in succ(s)$ and one terminating node, denoted as $\text{end}$, such that $succ(\text{end}) = \emptyset$.

More complex program representations are built on top of CFG, and the most notable one is SSA form.

2. SSA form

Static Single Assignment form (SSA form [6]) is a de-facto standard IR used nowadays by absolute majority of optimizing compilers [23, 19, 20, 9, 25, 26, 1]. This kind of program representation is popular because it represents explicitly both control and data flow in a quite compact and easy-to-maintain form. To date, most optimization algorithms have been reimplemented to operate on SSA form [29, 18, 4, 10, 28]. SSA-based implementations of optimization algorithms typically perform faster and have better computation
complexity as compared to the traditional implementations based on bit vectors. Better performance a is natural consequence of the sparse nature of SSA form [4].

In SSA form, each definition of a variable\(^1\) is given a unique version, and different versions of the same variable can be regarded as different program variables. Each use of a variable can only refer to a single reaching definition. When several definitions of a variable \(v\), e.g. \(v_1, v_2, \ldots v_m\), reach a confluence node in the control flow graph, a \(\phi\)-function \(v_n = \phi(v_1, v_2, \ldots v_m)\) is inserted to merge them into the definition of a new variable \(v_m\). The \(\phi\)-function acts as an intelligent assignment, producing as a result one of its arguments depending on what incoming edge the control flow arrived.

Translation to SSA form is described in the works of Cytron, Zadeck et. al. [6], a very good overview of alternative algorithms is given in [3]. The translation process comprises two steps: at the first step, \(\phi\)-functions like \(v = \phi(v, \ldots v)\) are added at the merge points; at the second step, variables are versioned.

3. Implicit arguments and results

In most programming languages, a statement may implicitly depend on a variable even if the variable is not a syntactical part of the statement body. A trivial example is a call to an external function, which may (but not must) modify any global variable.

There is an obvious request to find and explicate such implicit dependencies, this is why side effects analysis is an important part of modern compilers. Explication of the dependencies requires a more complicated notion of statements: instead of a simple \(a = op(b, c)\), a statement must be encoded like \(\langle a, u, v, \ldots m \rangle = op(b, c, k, \ldots l)\), where additional variables in the left and right parts denote implicit results and arguments, respectively.

Note the difference in how the compiler treats explicit and implicit arguments. If a variable \(v\) is implicitly used by a statement \(s\), the compiler has to store the actual value of \(v\) into the memory location associated with \(v\)’s symbolic name, because \(s\) accesses the value from memory. In contrast, an explicit use of \(v\) gives the compiler freedom to place the variable to a register and generate the appropriate CPU instruction for \(s\). So we can say that an implicit argument is a use of an addressed variable (location), while an explicit argument is a use of a non-addressed variable (register).

\(^1\)Each statement that alters value of a variable \(v\) is regarded as its definition.
Let start notation that was introduced in [14]:

Paths $\text{CFG}$, including the cyclic ones: define a variable both explicitly and implicitly: side-effects analysis. For simplicity, we assume that a statement cannot denote implicit results and arguments that are already computed by $\text{ddef, idef, iuse, duse}$ important for the further account: $\text{ddef, idef, iuse, duse}$

Let $\text{start}$ and $\text{end}$ denote the unique $\text{start node}$ and $\text{end node}$ of the $\text{CFG}$.

$\text{Predicates on statements.}$ There are four predicates that are important for the further account: $\text{ddef, idef, iuse, duse} : Stms \times V ars \rightarrow \{0, 1\}$, where: $\text{ddef}(s) \equiv (r = v)$ denotes explicit results, $\text{duse}(s) \equiv (v \in \text{args})$ denotes explicit arguments, and $\text{idef, iuse}$ denote implicit results and arguments that are already computed by side-effects analysis. For simplicity, we assume that a statement cannot define a variable both explicitly and implicitly:

$$\forall v \in V ars, \{s | s \in Stms, ddef(s, v) \land idef(s, v)\} = \emptyset.$$

$\text{Execution paths.}$ Define $\text{Paths}$ as a set of all finite paths over the $\text{CFG}$, including the cyclic ones: $\text{Paths} = \{p = \langle p_1, \ldots, p_N \rangle | \forall 1 \leq i \leq N, p_{i+1} \in \text{succ}(p_i)\}$. The $i$-th element of $p$ is denoted as $p_i$ and the length of $p$ is denoted as $\lambda(p)$. For simplicity, we reduce $p_{\lambda(p)}$ to $p_\lambda$ where appropriate.

Then, define $\text{Paths}[s_1, s_2]$ as a set of all finite paths starting with $s_1$ and ending in $s_2$: $\text{Paths}[s_1, s_2] = \{p | p \in \text{Paths}, p_1 = s_1, p_\lambda = s_2\}$. A path $q$ is said to be a subpath of $p$, in signs $q \subseteq p$, if there is an index $1 \leq i \leq \lambda(p)$ such that $i + \lambda q - 1 \leq \lambda p$ and $q_j = p_{i+j-1}$ for all $1 \leq j \leq \lambda q$. In particular, for any path $p$ and indices $i, j \leq \lambda p$ we denote subpath induced by a sequence of nodes from $p_i$ to $p_j$ by $[p_i, p_j]$. Moreover, if $p_i$ or $p_j$ is excluded from this subpath, we will write $[p_i, p_j]$ or $[p_i, p_j]$, respectively. Adjoined paths are concatenated in a natural way: for paths $p$ and $q$, if $p_\lambda = q_1$, then $p \oplus q = [p_1, \ldots, p_\lambda, q_2, \ldots, q_\lambda]$.

$\text{Predicates on paths.}$ Let $a$ be some predicate on statements, e.g. $a : Stms \rightarrow \{0, 1\}$, and let $p \in \text{Paths}$. We use the following compact notation that was introduced in [14]:

$$\forall s \in p, a(s) \leftrightarrow a^\forall(p)$$

$$\exists s \in p, a(s) \leftrightarrow a^\exists(p)$$

The same notation can be applied for predicates over edges, if $b : Stms * Stms \rightarrow \{0, 1\}$:

$$\forall 1 \leq i \leq \lambda(p), b(p_i, p_{i+1}) \leftrightarrow b^\forall(p)$$

$^2$For simplicity, we represent constants as variables.
\[ \exists 1 \leq i \leq \lambda(p), \ b(p_i, p_{i+1}) \iff b^2(p) \]

In order to minimize the amount of parentheses, we assume these predicates have higher priority than any other operations:

\[ \neg c^y(p) \iff \neg(\forall s \in p, \ c(s)) \quad (\neg c)^y(p) \iff \neg c^3(p) \]

\[ (\neg c)^y(p) \iff \forall s \in p, \ \neg c(s) \quad (\neg c)^3(p) \iff \neg c^3(p). \]

**Notation of data flow equations.** All data flow equations in this paper keep attributes on statements, with two distinct sets for before-statement and after-statement attributes. As an example, consider the following simple data flow equation:

\[
\begin{aligned}
    A_{\text{in}}(t) &= \begin{cases} 
    \text{false} & \text{if } t = \text{start}, \\
    \bigwedge_{p \in \text{pred}(t)} A_{\text{out}}(p) \setminus A_{\text{arc}}(p, t) \lor A_{\text{arc}}^+(p, t) & \text{otherwise,} 
    \end{cases} \\
    A_{\text{out}}(t) &= A_{\text{in}}(t) \setminus A_{\text{nd}}(t) \lor A_{\text{nd}}^+(t).
\end{aligned}
\]  

(1)

Attribute \( A_{\text{in}}(s) \) specifies the value of \( A \) before the execution of \( s \), while \( A_{\text{out}}(s) \) stands for the value of \( A \) after \( s \). Single-argument predicates \( A_{\text{nd}}(t) \) and \( A_{\text{nd}}^+(t) \) contribute to the node transition function, while two-argument predicates \( A_{\text{arc}}(p, t) \) and \( A_{\text{arc}}^+(p, t) \) form the edge transition function. These equations, as well as all that follow, operate on the two-element boolean lattice and use the distributive flow functions, so the meet over all paths solution (MOP) coincides with the maximal fixed point solution (MFP) [13]. Further on we simply say “a solution of a flow equation” assuming the MFP-solution.

5. Partial SSA form

A program with implicit results and arguments can be directly translated to full SSA by applying versioning to all variables, rewriting all implicit and explicit definitions and uses, as it was suggested in [6]. Figure 2a shows the full SSA form for the source program in Figure 1. However, there are several disadvantages inherent to this approach:

- Full representation is not compact, because the number of variable versions and corresponding IR objects that keep the def-use relation is huge, it can be roughly estimated as \( |Stms| \times |Vars| \). So, the IR becomes quadratic of the size of a program, and suppose we have a quadratic algorithm operating on this IR...

In Section 14 below we show some figures for the size of full SSA for typical Java programs.
v = 10;
if(..) {
    foo();
    bar();
    else {
        baz();
    }
    v = v + 1;
    c = v;

Figure 1. A program example

Figure 2. Examples of full and partial SSA form
• Register allocation now has to take into account all implicit uses and definitions. In some cases, it may be preferable to put a variable to a register, but if the variable is indirectly used, it must be put to memory.

• We lose one-by-one correspondence between statements and SSA variables, because each statement defines more than one variable. While this is not a problem in theory, it really complicates the optimizer code.

• Addressed (indirectly used) and non-addressed (directly used) variables are different kinds of entities, and they have different optimization rules. It is sufficient to note that assignment propagation is not applicable to addressed variables (two SSA versions of an addressed variable cannot be alive at the same time). So, separation of these kinds of entities may lead to easier design of the optimizer.

We noted that full SSA is redundant for the optimization purposes, because an explicit def-use link between implicit def and implicit use does not contribute any information to the optimizer [5]. It looks advantageous to translate into SSA only those variables whose def-use links are of any use for the optimizer. Generally, we come to the following translation process:

1. Each source program variable $v$ dissipates into a pair of variables: an addressable $v_m$ and a non-addressable $v_r$.

2. Each program statement is rewritten: each explicit assignment to a source variable $v$ is converted into an assignment to the corresponding non-addressable $v_r$. Each implicit definition is interpreted as an assignment to $v_m$. Uses are interpreted correspondingly.

3. Additional assignment statements are added to the program. These statements copy the values from $v_m$ to $v_r$ and back, thus preserving program correctness.

4. Only non-addressable variables ($v_r$) are translated to SSA form with the use of the standard algorithm [6].

The steps above effectively build an internal representation which we call partial SSA form, or pSSA. Figure 2b demonstrates pSSA for the program on Figure 1.

6. Assignment placement

Of the four steps described above, steps (1) and (2) are trivial, and step (4) is performed by the standard algorithm. So, the only nontrivial action in converting a program to pSSA is step (3) which inserts assignments that
copy the values between addressed (memory) and non-addressed (register) variables. This task can be viewed as a computation of a pair of predicates over the Cartesian product of edges and variables:

\[ \text{InsertL}, \text{InsertS} : \text{Stms} \times \text{Stms} \times \text{Vars} \Rightarrow \{0, 1\}. \]

If \( \text{InsertL}(s, p, v) = 1 \), then an assignment \( v_r = v_m \) is inserted between \( s \) and its immediate successor \( p \), where \( v_r \) and \( v_m \) are non-addressable and addressable variables corresponding to \( v \) that were created at step (1). If \( \text{InsertS}(s, p, v) = 1 \), then an assignment \( v_m = v_r \) is inserted.

We call the pair \( \langle \text{InsertL}, \text{InsertS} \rangle \) a placement function \([14]\), an assignment \( v_r = v_m \) a load statement, and an assignment \( v_m = v_r \) a store statement \([12]\). From now on, we annotate predicates related to a placement function \( P \) by adding the function name as the lower index: \( P \equiv \langle \text{InsertLP}, \text{InsertSP} \rangle \).

Further on we skip the program symbol \( \text{Prog} \) and variable symbols \( v, v_r, v_m \). In other words, we assume that all argumentation is applied to an arbitrary (but fixed at this moment) program \( \text{Prog} \) and an arbitrary (but fixed) variable \( v \). This does not impose any restrictions, because all our reasoning can be performed independently for each variable.

7. Correctness criterion

Obviously, not any placement function produces a correct pSSA form for a given program. In this section, we formulate the criterion that selects correct placement functions among the others. In fact, the criterion is simple: on a path from a definition to a use, the inserted copy statements (loads and stores) must correlate to the nature of definition and use statements, so that the use statement receives the value produced by the definition, even if the value was produced in memory and used in a register or vice versa.

Let us formulate the standard data flow problem of “available variables” \([22, \text{Chapter 8}]\) independently for addressed and non-addressed variables.

\[
\begin{align*}
\text{AvailReg}_{\text{in}}(t) &= \begin{cases} 
\text{false} & \text{if } t = \text{start}, \\
\bigwedge_{p \in \text{pred}(t)} (\text{AvailReg}_{\text{out}}(p) \lor \text{InsertL}(p, t)) & \text{otherwise},
\end{cases} \\
\text{AvailReg}_{\text{out}}(t) &= (\text{AvailReg}_{\text{in}}(t) \lor \text{ddef}(t)) \setminus \text{idef}(t); \quad (2)
\end{align*}
\]

\[
\begin{align*}
\text{AvailMem}_{\text{in}}(t) &= \begin{cases} 
\text{true} & \text{if } t = \text{start}, \\
\bigwedge_{p \in \text{pred}(t)} (\text{AvailMem}_{\text{out}}(p) \lor \text{InsertS}(p, t)) & \text{otherwise},
\end{cases} \\
\text{AvailMem}_{\text{out}}(t) &= (\text{AvailMem}_{\text{in}}(t) \lor \text{idef}(t)) \setminus \text{ddef}(t).
\end{align*}
\]

Consider the first equation. The explicit definition \( \text{ddef} \) contributes \( \text{true} \) to the solution, while implicit definition \( \text{idef} \) resets the solution to

\(^3\text{e.g. we replace idef}(s, v) \text{ by idef}(s).\)
false. So, \( \text{AvailReg}_{\text{in}}(s) \) is true iff the non-addressed variable \( v_r \) contains the actual value of \( v \) on each incoming path to \( s \). So, the system really specifies the “available variables” problem with explicit definitions treated as defs and implicit definitions treated as kills. Similarly, the second equation computes availability of the actual value in the addressed variable \( v_m \).

Using the equations above, the correctness criterion may be formulated in a very short form. Really, a pSSA form of a program is correct, if (1) before execution of a statement \( s \) that uses the variable \( v \) explicitly, \( v_r \) is available, and (2) before execution of a statement \( s \) that uses the variable implicitly, \( v_m \) is available\(^4\):

\[
\forall s \in \text{Stms} \begin{cases} 
\text{duse}(s) \leq \text{AvailReg}_{\text{in}}(s), \\
\text{iuse}(s) \leq \text{AvailMem}_{\text{in}}(s);
\end{cases}
\]

\[
\forall s_1 \in \text{Stms}, \forall s_2 \in \text{succ}(s_1) \begin{cases} 
\text{InsertS}(s_1, s_2) \leq \text{AvailReg}_{\text{out}}(s_1), \\
\text{InsertL}(s_1, s_2) \leq \text{AvailMem}_{\text{out}}(s_1).
\end{cases}
\]

**Definition 1** Correctness. A pair \( \langle \text{InsertL}, \text{InsertS} \rangle \) is correct if and only if it specifies a solution of systems (2) and (3) that satisfies (4).

For a program \( \text{Prog} \), we denote a set of all correct placement functions as \( \text{CORRECT}(\text{Prog}) \).

8. Optimality criteria

Obviously, there is more than one correct placement function for a given program \( \text{Prog} \). Moreover, there is not the “best” placement function for a program. This is because the load/store statements are abstract, i.e., they exist in the program representation until optimization is finished, and then each one is translated either into assignments or into nothing, depending on the results of register allocation. This is why these statements have no fixed execution cost, so one cannot say, for instance, that adding a load statement makes a program slower.

So, the optimality criteria are somewhat abstract, they aim at making more comfort for the optimizer while minimizing the memory usage. The basic idea is this: the best placement function makes optimization no harder as compared to the full SSA and generates IR as compact as possible. This idea is complemented by several auxiliary criteria that help to choose the best placement function in the case of ambiguity. The following optimality criteria are considered.

\(^4\)load and store statements do use addressed and non-addressed variables, correspondingly.
8.1. Completeness of def-use chains

The pSSA representation of a program must contain all def-use links between explicit definitions and explicit uses for each variable. In other word, if there is a path \( p \), and a variable \( v \) is explicitly defined by \( p_1 \), explicitly used by \( p_\lambda \), and not redefined along \( p \), there must be no load statements (like \( v_r = v_m \)) on \( p \) as well. Let us denote the set of all placement functions satisfying this criterion as \( T_1 \):

\[
A \in T_1 \overset{\text{def}}{=} \forall p \in \text{Paths} \\
ddef(p_1) \land \neg(ddef \lor idef)^\exists(p) \land duse(p_\lambda) \\
\Rightarrow \neg \text{InsertL}_A^\exists(p).
\] (5)

Examples of the def-use complete and not-complete representations of the same source program are given on Figure 3. Simply speaking, every pSSA form satisfying this criterion gives the optimization algorithms everything they need, making no difference whether full SSA or pSSA is used.

8.2. Absence of redundant load statements

Consider a program fragment in Figure 4. Both placement functions are correct, and both satisfy the T1 criterion. In order to resolve the ambiguity, we state that the first function (\( II \)) is better because it does not contain redundant load statements, despite it increases the register pressure.

Denote the set of all placement functions satisfying this criterion as \( T_2 \):
8.3. Full anticipation of store statements

As the previous one, this criterion helps to select a better placement function in the case of ambiguity. Consider the example on Figure 3. The placement function $A$ does store a value from register to memory on a path $ddef$–$duse$, where the memory is not read. We state that $\Pi$ is better, because it does not store the value into memory on a path where the value is not used.

Denote the set of all placement functions satisfying this criterion as $T3$:

$$A \in T3 \equiv A \in T2 \land \forall p \in \text{Paths}$$

$$\text{Insert}_{A}(s_1, s_2) \land \text{Insert}_{\Pi}(s_2, \text{end}) \Rightarrow \exists j \prec \lambda(p), 
\neg \text{idf}^2([p_1, p_j]) \land \neg \text{Insert}_{A}^2([p_1, p_j]) \land \text{iuse}(p_j).$$

8.4. Computational optimality

Assuming that a program is already translated to pSSA, consider an “ideal” register allocation scheme, i.e., each non-addressed variable is assigned to a register. In this case, each load/store statement performs a register ↔ memory copy, so it has a fixed execution cost. So, we can say that a placement function $A$ is better than $B$ if it places the lesser number of load/store
statements on each execution path. Consider an example in Figure 5. The placement function $\Pi$ inserts load and store out of the loop, while the placement function $A$ moves them into the loop. So, $\Pi$ is computationally better.

\[
\begin{align*}
\text{\textbf{Figure 5.}} \quad &\Pi \in \text{CMP\textunderscore OPT} & A \notin \text{CMP\textunderscore OPT}
\end{align*}
\]

Let us count all load/store statements that are added to a path by a placement function. Let $p \in \text{Paths}$, $A \in T3$:

\[
\begin{align*}
count\text{\_loads}_A(p) & \equiv \{|i | 1 \leq i \leq \lambda(p), \text{InsertL}_A(p_i, p_{i+1})\}, \\
count\text{\_stores}_A(p) & \equiv \{|i | 1 \leq i \leq \lambda(p), \text{InsertS}_A(p_i, p_{i+1})\}.
\end{align*}
\]

Then we can define a partial order on the set $T3$:

\[
\begin{align*}
A \leq_L B & \equiv \forall p \in \text{Paths}[\text{start}, \text{end}] \quad \text{count\_loads}_A(p) \leq \text{count\_loads}_B(p), \\
A \leq_S B & \equiv \forall p \in \text{Paths}[\text{start}, \text{end}] \quad \text{count\_stores}_A(p) \leq \text{count\_stores}_B(p), \\
A \leq_{\text{comp}} B & \equiv (A \leq_L B) \land (A \leq_S B).
\end{align*}
\]

The relationship $\leq_{\text{comp}}$ generates a partial order on $T3$. Denote the set of minimal elements as $\text{CMP\textunderscore OPT}$:

\[
\text{CMP\textunderscore OPT} \equiv \{A | A \in T3, \forall B \in T3, \ A \leq_{\text{comp}} B\}. \quad (8)
\]

Further on we prove that CMP\textunderscore OPT is not empty.

Note that all placement functions that belong to CMP\textunderscore OPT generate the optimal code not only for “ideal” register allocation, but for any “sensible” register allocation that spills all addressed variables to memory.
8.5. Lifetime minimization

Among two placement function that are both computationally optimal, one function is better than the other if it generates shorter lifetimes for non-addressed variables. This criterion is an important one, because it minimizes the register pressure and helps back-end to generate a better code.

A non-addressable variable \( v_r \) is “alive” at an edge \( \langle s_1, s_2 \rangle \) if there is a path (also called lifetime range) that contains the edge and has the following properties:

- \( v_r \) is defined at the path beginning (either explicitly or by \textit{load}),
- \( v_r \) is used at the path end (either explicitly or by \textit{store}),
- \( v_r \) is not redefined along the path.

Let us denote the set of all lifetime ranges generated by a placement function \( A \) as \( \text{LtRg} \):

\[
\text{LtRg}_A \equiv \{ p \mid p \in \text{Paths}, \quad (\text{ddf}(p_1) \lor \text{InsertL}_A(p_1, p_2)) \land (\text{duse}(p_\lambda) \lor \text{InsertS}_A(p_{\lambda-1}, p_\lambda)) \land \neg\text{ddf}^2([ ]) \land \neg\text{InsertL}_A^2( [ ]) \}. \tag{9}
\]

For a full path \( p \) we denote a set of all edges that (1) belong to the path and (2) the variable is alive at them:

\[
\text{LiveArcs}_A(p \in \text{Paths}[\text{start}, \text{end}]) \equiv \{ (a, b) \mid \exists q \in \text{LtRg}_A, q \subseteq p, \quad \exists i, a = q_i, b = q_{i+1} \}. \tag{10}
\]

Now we can define the best placement function as the function that minimizes the lifetime of non-addressable variables:

\[
A \preceq_{LT} B \equiv \forall p \in \text{Paths}[\text{start}, \text{end}], \text{LiveArcs}_A(p) \subseteq \text{LiveArcs}_B(p),
\]

\[
\text{LT}_{OPT} \equiv \{ A \mid A \in \text{CMP}_{OPT} \land (\forall B \in \text{CMP}_{OPT} \quad A \preceq_{LT} B) \}.
\]

Further we build the placement function \( II \) that belongs to \( \text{LT}_{OPT} \), thus proving that the set is not empty.

**Definition 2** Optimality. The problem of creation of an optimal pSSA form for a program \( \text{Prog} \) can be solved by computing a placement function that belongs to \( \text{LT}_{OPT} \).
9. Code motion

To build the optimal placement function, we use a program optimization method generally known as code motion [14, 21]. The method is based on solving a system of data flow equations and using the solution to optimally move program statements over the CFG. In this section, we briefly describe the basics of busy code motion as it was formulated in [14].

**Input Data.** Optimization is applied to one syntactically identified expression $E$ at a time, that is, data flow equations are built and solved for one expression, then the program is modified and the process repeats for the next expression\(^5\). Syntactically identified expression here means a set of statements whose right parts are identical, e.g. $a \ op \ b$, where $a, b$ are some variables or constants, \(op\) is an operation symbol.

Two predicates $Comp$ and $Transp$ are defined over $Stms$. By definition, $s \in Comp$ iff its right part coincides with the active expression ($E$), that is, if $s$ computes $E$. A statement belongs to $Transp$ if no one argument of the expression is altered by execution of the statement, that is, if the statement is transparent for the expression.

In order to keep things simple, we assume that all statements belonging to $Comp$ have the same left part, that is, each statement $c = a \ op \ b$ is converted into a pair $t_E = a \ op \ b; \ c = t_E$, where $t_E$ is a new variable.

The optimization purpose is removal of as many redundant expressions as possible. In other words, the optimized program will have as few computations of $E$ on each path as possible. The task is complicated by the request for safety: if a path contained no computation of $E$ before the optimization, it must still be free from $E$ after it.

The optimization result is computed as a pair of predicates: $Insert$ contains all edges where a new statement $t_E = a \ op \ b$ must be inserted, and $Replace$ contains statements that become fully redundant and may be removed from the program.

The busy code motion (BCM) solution is computed by solving a pair of flow equations, namely $UpSafe$ and $DownSafe$. $UpSafe\_in(s)$ holds iff there is an occurrence of the expression $E$ on each path from $\text{start}$ to $s$. Simply put, $UpSafe$ holds right before the statement $s$ if the value of $E$ is computed before execution of $s$ on any execution path that includes $s$:

\[
UpSafe\_in(s) \iff \forall p \in Paths[\text{start}, s] \\exists i < \lambda p, Comp(p_i) \land Transp^\forall([p_i, p_x]).
\]

Similarly, $DownSafe\_out(s)$ holds iff there is a computation of $E$ on each path from $s$ to $\text{end}$:

\[\text{Effective implementation runs many systems in parallel using the bit vector-based approach.}\]
DownSafe_out(s) ⇔ ∀p ∈ Paths[s, end]

∃i < λp, Comp(p_i) ∧ Transp \(\uparrow\) [p_1, p_i].

Then, the Safe predicate is computed as a sum of UpSafe and DownSafe. A statement \(s\) is safe if each path going through it contains an occurrence of \(E\) (either before or after \(s\)) with the same values of arguments as they would have at \(s\). Then, the Insert predicate is computed as the upper bound of Safe: it contains all edges \((a, b)\) such that \(a \not\in Safe \land b \in Safe\). Finally, Replace = Comp.

\[
\begin{align*}
\text{UpSafe}_\text{in}(t) &= \begin{cases} 
false & \text{if } t = \text{start}, \\
\land_{p \in \text{pred}(t)} \text{UpSafe}_\text{out}(p) & \text{otherwise}, 
\end{cases} \\
\text{UpSafe}_\text{out}(t) &= (\text{UpSafe}_\text{in}(t) \lor \text{Comp}(t)) \land \text{Transp}(t);
\end{align*}
\]

\[
\begin{align*}
\text{DownSafe}_\text{in}(t) &= (\text{DownSafe}_\text{out}(t) \land \text{Transp}(t)) \lor \text{Comp}(t), \\
\text{DownSafe}_\text{out}(t) &= \begin{cases} 
false & \text{if } t = \text{end}, \\
\land_{p \in \text{succ}(t)} \text{DownSafe}_\text{in}(p) & \text{otherwise}; 
\end{cases}
\end{align*}
\]

\[
\begin{align*}
\text{Safe}_\text{in}(t) &= \text{UpSafe}_\text{in}(t) \lor \text{DownSafe}_\text{in}(t), \\
\text{Safe}_\text{out}(t) &= \text{UpSafe}_\text{out}(t) \lor \text{DownSafe}_\text{out}(t);
\end{align*}
\]

\[
\begin{align*}
\text{Insert}_{BCM}(p, q) &= p \in \text{pred}(q) \land \\
&\quad (\neg \text{Safe}_\text{out}(p) \lor \neg \text{Transp}(p)) \land \text{Safe}_\text{in}(q), \\
\text{Replace}_{BCM}(p) &= \text{Comp}(p).
\end{align*}
\]

**Theoretical results.** Code motion has a well-developed theoretical background. The following theorems, proven in [14], are important for us:

**Theorem 1.** Program transformation implied by the pair \(\text{Insert}_{BCM}\) and \(\text{Replace}_{BCM}\) is safe, it does not add a computation of \(E\) on a path where such a computation did not exist previously.

**Theorem 2.** Program transformation implied by the pair \(\text{Insert}_{BCM}\) and \(\text{Replace}_{BCM}\) is computationally optimal, it minimizes the number of computations of \(E\) on each execution path (as compared to any other safe transformation).

**Theorem 3.** Program transformation implied by the pair \(\text{Insert}_{BCM}\) and \(\text{Replace}_{BCM}\) maximizes the lifetime of the auxiliary variable \(t_E\) as compared to any other computationally optimal code motion transformation.
10. The solution

As we formulated above (Definition 2), to find the optimal pSSA form for a program, it is sufficient to compute a placement function that belongs to \( \text{LT}_{\text{OPT}} \). We compute the function by solving a system of flow equations that are similar to the equations used by code motion.

In fact, code motion computes the optimal placement for a set of statements that compute identical values, with an accent on computational and lifetime optimality, that is very close to our criteria. It moves the computation statements within the bounds imposed by a pair of predicates, one of them (Transp) essentially encodes where arguments of the expression are defined, and the other one (Comp) encodes the uses of the expression. In our case, there are four predicates instead of two, because we have the explicit definition (\( d\text{def} \)), implicit definition (\( i\text{def} \)), explicit use (\( d\text{use} \)) and implicit use (\( i\text{use} \)). Also, a placement function must comply with our correctness and (desirably) optimality criteria, which are more complex than the restrictions imposed on the code motion transformation.

However, below we show the flow equations for optimal placement function, and these equations are derived from the equations employed by the code motion. Two equations define anticipability, i.e. whether a value is used below on some (for partial anticipability) or on all (for full anticipability) execution paths. Other two equations compute full availability, i.e., whether a value is computed in advance on all execution paths.

System (13) computes \( \text{full anticipability} \) for the addressed variable, i.e., the presence of uses of the addressed variable on each outgoing path:

\[
\begin{align*}
F\text{AntMem}_\text{out}(t & \in \text{Stms}) \iff \forall p \in \text{Paths}[t, \text{end}] \\
& \exists i > 1, \ i\text{use}(p_i) \land \neg (d\text{def} \lor i\text{def})^3[ ]p_i, t[ ].
\end{align*}
\]

The next system (14) computes \( \text{full availability} \) for the addressed variable, i.e., whether every path coming to a statement has either an explicit definition or a copy from register to memory:

\[
\begin{align*}
F\text{AvlMem}_\text{in}(t & \in \text{Stms}) \iff \forall p \in \text{Paths}[\text{start}, t] \exists i, \\
& (i\text{use}(p_i) \lor i\text{def}(p_i)) \land \neg (d\text{def} \lor i\text{def})^3[ ]p_i, t[ ];
\end{align*}
\]

\[
\begin{align*}
F\text{AvlMem}_\text{in}(t) &= F\text{AvlMem}_\text{out}(t) \setminus d\text{def}(t) \setminus i\text{def}(t) \lor i\text{use}(t), \\
F\text{AvlMem}_\text{out}(t) &= \begin{cases} \\
false & \text{if } t = \text{end}, \\
\bigwedge_{p \in \text{succ}(t)} F\text{AvlMem}_\text{in}(p) & \text{otherwise;}
\end{cases}
\end{align*}
\]

(13)
\[
F_{\text{AvlMem} \text{in}(t)}(t) = \begin{cases} 
\text{true} & \text{if } t = \text{start}, \\
\bigwedge_{p \in \text{pred}(t)} F_{\text{AvlMem} \text{out}(p)} & \text{otherwise}, 
\end{cases}
\]

\[
F_{\text{AvlMem} \text{out}(t)}(t) = (F_{\text{AvlMem} \text{in}(t)}(t) \lor \text{idef}(t) \lor \text{iuse}(t)) \setminus \text{ddef}(t).
\]

The set of points where \( F_{\text{AntMem}}(s) \lor F_{\text{AvlMem}}(s) \) is true defines the set of “safe” store placements. Selecting a placement that puts stores at the upper bound of the safe set, we obtain the optimal solution.

\[
S_{\text{safe}}(t \in \text{Stms}) \text{ def } \equiv F_{\text{AvlMem} \text{in}(t)}(t) \lor F_{\text{AvlMem} \text{out}(t)}(t),
\]

\[
\text{SafeS}_{\text{in}}(t \in \text{Stms}) \text{ def } \equiv F_{\text{AvlMem} \text{in}(t)}(t),
\]

\[
\text{SafeS}_{\text{out}}(t \in \text{Stms}) \text{ def } \equiv F_{\text{AvlMem} \text{out}(t)}(t),
\]

\[
\text{InsertS}_{\Pi}(p, s \in \text{Stms}) \text{ def } \equiv s \in \text{succ}(p) \land (\neg \text{SafeS}_{\text{out}}(p) \lor \text{ddef}(p)) \land \text{SafeS}_{\text{in}}(s).
\]

System (16) computes partial anticipability of non-addressed variable, i.e. whether there is at least one use of \( v_r \) downwards.

System (17) is similar to (14), but it misses one item, so

\[
\forall s \in \text{Stms} F_{\text{AvlMemOnly} \text{in}(s)}(s) \leq F_{\text{AvlMem} \text{in}(s)}.
\]

Let us see the difference between the systems: \( F_{\text{AvlMemOnly} \text{in}(s)}(s) \) is true only if the actual value of the variable at the point \( s \) is available in memory only, while \( F_{\text{AvlMem} \text{in}(s)}(s) \) is true if the actual value is available in memory only, or if it is available in memory and register at the same time.

\[
\begin{align*}
\text{PAntReg}_{\text{in}}(t) &= P_{\text{AntReg} \text{out}(t)} \setminus \text{ddef}(t) \setminus \text{idef}(t) \lor \text{duse}(t), \\
\text{PAntReg}_{\text{out}}(t) &= \begin{cases} 
\text{false} & \text{if } t = \text{end}, \\
\bigvee_{p \in \text{succ}(t)} (\text{PAntReg}_{\text{in}}(p) \lor \text{InsertS}_{\Pi}(t, p)) & \text{otherwise};
\end{cases} \\
F_{\text{AvlMemOnly} \text{in}(t)}(t) &= \begin{cases} 
\text{true} & \text{if } t = \text{start}, \\
\bigwedge_{p \in \text{pred}(t)} F_{\text{AvlMemOnly} \text{out}(p)} & \text{otherwise},
\end{cases}
\end{align*}
\]

\[
\text{InsertL}_{\Pi}(p, s \in \text{Stms}) \text{ def } \equiv s \in \text{succ}(p) \land F_{\text{AvlMemOnly} \text{out}(p)} \land ((\text{PAntReg}_{\text{in}}(s) \land \neg F_{\text{AvlMemOnly} \text{in}(s)}) \lor \text{duse}(s)).
\]

From now on we denote the placement function \( \langle \text{InsertL}_{\Pi}, \text{InsertS}_{\Pi} \rangle \) as \( \Pi \).

It is easy to observe that flow equations computing store placements are just the same as in the busy code motion, the only difference is substitution
Partial SSA form

of predicates: \( \neg Transp \) is replaced by \((ddef \lor idef)\), \(Comp\) is replaced by \(iuse \lor idef\). However, the equations for the load placement are different, because one of them employs \(\lor\) as a merge function, so it computes “partial” but not “full” anticipability. Finally, the order of equation systems is important, systems (16) and (17) depend on the previous ones.

11. Auxiliary theorems

The following theorems help us to switch from the language of data flow equations to the standard notation of “for each path there is a statement with the given properties” and back.

Let us consider an abstract system of data flow equations:

\[
\begin{align*}
A_{in}(t) &= \begin{cases} 
false & \text{if } t = \text{start,} \\
\bigwedge_{p \in \text{pred}(t)} A_{out}(p) \setminus A_{arc}^{-}(p,t) \lor A_{arc}^{+}(p,t) & \text{otherwise,}
\end{cases} \\
A_{out}(t) &= A_{in}(t) \setminus A_{nd}^{-}(t) \lor A_{nd}^{+}(t).
\end{align*}
\]

Theorem 4. For a statement \(s\) the following assertions are equivalent:

- \(A_{in}(t)\) is true;
- \(\forall p \in \text{Paths}[\text{start},t] \exists i, (A_{arc}^{+}(p_{i-1},p_{i}) \lor A_{arc}^{-}(p_{i},p_{\lambda})) \land \neg A_{nd}^{-}(p_{i},p_{\lambda}) \land \neg A_{arc}^{-}(p_{i},p_{\lambda}).

Theorem 5. For a statement \(s\) the following assertions are equivalent:

- \(A_{in}(t)\) is false;
- \(\exists p \in \text{Paths}, \text{ such that } p_{\lambda} = t, (A_{arc}^{-}(p_{i}) \lor A_{arc}^{+}(p_{1},p_{2})) \land \neg A_{nd}^{+}(p_{i}) \land \neg A_{arc}^{+}(p_{i}).

Also we need a version of theorem (4) formulated for a backward system. Consider an abstract backward system of flow equations:

\[
\begin{align*}
B_{in}(t) &= B_{out}(t) \setminus B_{nd}(t) \lor B_{nd}^{+}(t), \\
B_{out}(t) &= \begin{cases} 
false & \text{if } t = \text{end,} \\
\bigwedge_{p \in \text{succ}(t)} B_{in}(p) \setminus B_{arc}(p,t) \lor B_{arc}^{+}(p,t) & \text{otherwise.}
\end{cases}
\end{align*}
\]

Theorem 6. For a statement \(s\) the following assertions are equivalent:

- \(B_{in}(t)\) is true;
- \(\forall p \in \text{Paths}[t,\text{end}] \exists i \leq \lambda(p), \neg B_{nd}^{-}(p_{i}) \land \neg B_{arc}^{-}(p_{i}) \land (B_{nd}^{+}(p_{i}) \lor B_{arc}^{+}(p_{i-1},p_{i})).\)
We omit proofs of the theorems here. An interested reader can either consult [14], or prove them himself by applying the standard algorithm for computing the meet-over-all-paths solution (MOP) [13].

12. Proof of correctness

Theorem 7. The placement function $\Pi = (\text{InsertL}_\Pi, \text{InsertS}_\Pi)$, which is the solution of (18, 15), satisfies the correctness criterion stated in Definition (1).

Proof. Assume the contrary. System (4) contains 4 inequalities. Let us consider a negation of each of them:

- Assume $\exists s_1, s_2 \in \text{Stms}, \text{InsertS}_\Pi(s_1, s_2) \land \neg \text{AvailReg\_out}(s_1)$.

  1. $\text{InsertS}(s_1, s_2) \Rightarrow \neg \text{SafeS\_out}(s_1) \lor \text{ddef}(s_1) \Rightarrow \neg \text{FAvlMem\_out}(s_1) \Rightarrow \exists q \in \text{Paths}, \text{ddef}(q_1) \land (q_\lambda = s_1) \land \neg (\text{idef} \lor \text{iuse})^\exists(q)$.

  2. $\neg \text{AvailReg\_out}(s_1) \Rightarrow \exists p \in \text{Paths}, \text{idef}(p_1) \land (p_\lambda = s_1) \land \neg (\text{ddef} \lor \text{InsertL}_\Pi)^\exists(p)$.

  3. Let $p_i$ be the first common statement of the paths $q$ and $p$. From (1) and (2) $\Rightarrow \neg \text{FAvlMemOnly\_in}(p_i)$.

  4. $\text{InsertS}_\Pi(s_1, s_2) \Rightarrow \text{PAntReg\_in}(p_1) \Rightarrow \text{PAntReg\_out}(p_1)$.

  5. (2) $\Rightarrow \text{FAvlMemOnly\_out}(p_1)$.

  6. (2) $\Rightarrow \text{PAntReg\_out}(p)$.

  7. According to (3), (5) and (6) the path $[p_1, p_i]$ contains an edge where $\text{FAvlMemOnly}$ changes its value, and $\text{PAntReg}$ remains true. So, $\text{InsertL}_\Pi$ is true on the edge, which contradicts (2).

- Assume $\exists s_1, s_2 \in \text{Stms}, \text{InsertL}_\Pi(s_1, s_2) \land \neg \text{AvailMem\_out}(s_1)$.

  1. $\text{InsertL}_\Pi(s_1, s_2) \Rightarrow \text{FAvlMemOnly\_out}(s_1)$.

  2. Comparing the equations for $\text{FAvlMemOnly}$ and $\text{AvailMem}$, we have $\forall s \in \text{Stms}, \text{FAvlMemOnly\_out}(s) \Rightarrow \text{AvailMem\_out}(s)$.

  3. $\text{InsertL}_\Pi(s_1, s_2) \Rightarrow \text{AvailMem\_out}(s_1)$, which contradicts the assumption above.

- Assume $\exists s \in \text{Stms}, \text{duse}(s) \land \neg \text{AvailReg\_in}(s)$.

  1. Applying the auxiliary theorem (5) to the definition of $\text{AvailReg}$, we have $\exists p \in \text{Paths}, \text{idef}(p_1), p_\lambda = s_1, \neg (\text{ddef} \lor \text{InsertL}_\Pi)^\exists(p)$.

  2. (1) and $\text{duse}(s) \Rightarrow \text{PAntReg\_out}(p)$. 

The proof is completed.
3. \( \text{idef}(p_1) \Rightarrow \text{FAvlMemOnly} \text{ out}(p_1) \).

4. (2), (3) and \( \neg\text{InsertL}_P(p) \Rightarrow \text{FAvlMemOnly}^\forall(p) \).

5. From (4) and definition of \( \text{InsertL}_P \Rightarrow \text{InsertL}_P(p_{\lambda-1}, p_\lambda) \), which contradicts the definition of the path \( p \).

- Assume \( \exists s \in \text{Stms}, \text{iuse}(s) \land \neg\text{AvailMem in}(s) \).

1. \( \text{iuse}(s) \Rightarrow \text{FAntMem in}(s) \Rightarrow \text{SafeS in}(s) \).

2. Applying the auxiliary theorem (5) to the definition of \( \text{AvailMem} \), we have \( \exists p \in \text{Paths}, \text{ddef}(p_1) \land (p_\lambda = s) \land \neg(\text{idef} \lor \text{InsertS}_P)^3(p) \).

3. \( \neg\text{InsertL}_P^2(p) \xrightarrow{\text{from (1)}} \text{SafeS in}^\forall([p]) \).

4. \( \text{ddef}(p_1) \land \text{SafeS in}(p_2) \Rightarrow \text{InsertS}_P(p_1, p_2) \), which contradicts (2).

13. Proofs of optimality

**Theorem 8.** The placement function \( \Pi \) belongs to \( T_1 \).

**Proof.** Assume the contrary: \( \exists p \in \text{Paths}, \text{ddef}(p_1), \text{duse}(p_\lambda), \neg\text{idef}^3(p) \) and \( \exists p_i, p_{i+1} \) such that \( \text{InsertL}_P(p_i, p_{i+1}) \) is true. Then

\[
\text{InsertL}(p_i, p_{i+1}) \Rightarrow \text{FAvlMemOnly} \text{ out}(p_i).
\]

Applying the auxiliary Theorem 4 to the last statement, we have \( \forall q \in \text{Paths[ start, p_i]} \exists q_j, \text{idef}(q_j), \neg\text{ddef}^3([q_j, q_\lambda]) \). Applying the last statement to the path \( p: \exists k<i \) such that \( \text{idef}(p_k) \) is true, which contradicts the definition of \( p \).

**Theorem 9.** The placement function \( \Pi \) belongs to \( T_2 \).

**Proof.** Assume the contrary: \( \exists p \in \text{Paths} \) such that \( \text{InsertL}_P(p_1, p_2) \land \text{InsertL}_P(p_{\lambda-1}, p_\lambda) \land \neg\text{idef}^3([p]) \). Using the first item of the conjunction, we have \( \text{InsertL}(p_1, p_2) \Rightarrow \neg\text{FAvlMemOnly in}(p_1) \lor \text{duse}(p_2) \Rightarrow \neg\text{FAvlMemOnly in}(p_2) \). The third item of the conjunction means that the predicate \( \text{FAvlMemOnly} \) cannot change its value on the path \( [p_2, p_{\lambda-1}] \), so \( \text{FAvlMemOnly out}(p_{\lambda-1}) \) is false. This contradicts the second item of the conjunction.

**Theorem 10.** The placement function \( \Pi \) belongs to \( T_3 \).

**Proof.** This statement follows from the structure of equations (14): \( \text{InsertS}(s_1, s_2) \Rightarrow \text{SafeS in}(s_1) \land \neg\text{SafeS out}(s_2) \Rightarrow \text{FAntMem in}(s_2) \). Using the auxiliary Theorem 6 to the last statement, we come to the definition of \( T_3 \).
Lemma 1. Any placement function that belongs to CMP_OPT places load
statements only on those edges where PAntReg is true: ∀P ∈ CMP_OPT,
∀s₁, s₂ ∈ Stms, InsertLₚ(s₁, s₂) ⇒ PAntReg_in(s₂) ∧ PAntReg_out(s₁).

Proof. If there are P, s₁, s₂ contradictory to the statement above, then
we can built P': InsertLₚ' = InsertLₚ \ {s₁, s₂}. The correctness of P' is
natural consequence of ¬PAntReg_out(s₁), because the edge (s₁, s₂) does not
belong to any idef–duse path. Obviously, P' is computationally better than
P, so P /∈ CMP_OPT. Finally, PAntReg_out(s₁) is an obvious consequence
of PAntReg_in(s₂).

Theorem 11. The placement function II belongs to CMP_OPT (8).

Proof. We prove separately that II places the minimal amount of (1) loads
and (2) stores.
Case 1: ∀Q ∈ T₃, ∀p ∈ Paths[start, end], count_loadsQ(p) ≥ count_loadsII(p).
Let us assume the contrary, that there are a placement function Q and a
path p such that the inequality above is not true. Select a minimal subpath
of p which is bounded by definitions and does not satisfy the inequality
above:
(t ⊆ p) ∧ (t₁) = start ∨ idef(t₁) ∨ ddef(t₁) ∧ (t₂ = end ∨ idef(t₂) ∨
ddef(t₂)) ∧ count_loadsQ(t) < count_loadsII(t).

Lemma 2. count_loadsII(t) ≤ 1.

Proof. Let us assume the contrary:

∃i, j, i < j ∧ InsertLII(tᵢ, tᵢ₊₁) ∧ InsertLII(tⱼ, tⱼ₊₁).

Then InsertLII(tᵢ, tᵢ₊₁) ⇒ ¬FAvlMemOnly_in(tᵢ₊₁) ∨ duse(tᵢ₊₁) ⇒
¬FAvlMemOnly_out(tᵢ₊₁) ⇒ ¬FAvlMemOnly_in(tⱼ)
that contradicts InsertLII(tⱼ, tⱼ₊₁).

From Lemma 2 we have count_loadsQ(t) = 0 and count_loadsII(t) = 1,
that is, InsertLII(t) and ¬InsertLII₂(t). Assume the placement function II
inserts load between tᵢ and tᵢ₊₁. Then:

1) ¬InsertLQ₂(t) ≃(t) Q ∈ CORRECT duse(tᵢ₊₁) ⇒
PAntReg_in(tᵢ₊₁) ∧ ¬FAvlMemOnly_in(tᵢ₊₁) ∧ FAvlMemOnly_out(tᵢ);
2) from (13) ⇒ ¬FAvlMemOnly_in(tᵢ₊₁) ⇒ ∃q ∈ Paths, ddef(q₁) ∧ (q₁ =
tᵢ₊₁) ∧ ¬(ddef ∨ InsertLII₂(q)).
3) from (13) \( P_{\text{AntReg.in}}(t_{i+1}) \Rightarrow \exists s \in \text{Paths}, \ (s_1 = t_{i+1}) \wedge (\text{duse}(s_\lambda) \vee \text{Insert}_{\text{II}}(s_{\lambda-1}, s_\lambda)) \wedge \neg (\text{ddef} \vee \text{Insert}_{\text{II}}^3(s)); \)

4) consider a compound path \( m = [t_1, \ldots, t_{i+1} = s_1, \ldots, s_\lambda]. \) From (13) \( \Rightarrow \text{FAvlMemOnly.out}(t_1) \Rightarrow \neg \text{ddef}(t_1) \Rightarrow \text{idef}(t_1) \) by definition of \( s \)
\( \Rightarrow \text{Insert}_{\text{II}}^3(m). \) By the initial assumption, \( \neg \text{Insert}_{\text{II}}^3(q)(t) \Rightarrow \text{Insert}_{\text{II}}^3(p); \)

5) consider a compound path \( n = [q_1, \ldots, q_{i+1} = s_1, \ldots, s_\lambda]. \) The path connects an explicit definition to an explicit use, and \( \text{Insert}_{\text{II}}^3(n). \) It means that \( Q \in T1 \) is false, which contradicts the initial assumption.

**Case 2:** \( \forall Q \in T3, \forall p \in \text{FullPaths}, \text{count_stores}_Q(p) \geq \text{count_stores}_{\text{II}}(p). \)

Let us assume the contrary, that there is a placement function \( Q \) and a path \( p \) such that the inequality above is not true. Select a minimal subpath of \( p \) which is bounded by implicit uses and does not satisfy the inequality above:
\( (t \subseteq p) \wedge (\text{iuse}(t_1) \vee t_1 = \text{start}) \wedge (\text{iuse}(t_\lambda) \vee t_\lambda = \text{end}) \wedge \neg \text{iuse}^3(t_1, t_\lambda[\]) \wedge \text{count_stores}_Q(t) < \text{count_stores}_{\text{II}}(t). \)

**Lemma 3.** \( \text{count_stores}_{\text{II}}(t) \leq 1. \)

**Proof.** Assume the contrary:
\( \exists i, j, i < j \wedge \text{Insert}_{\text{II}}(t_i, t_{i+1}) \wedge \text{Insert}_{\text{II}}(t_j, t_{j+1}). \)
Then
\( \text{Insert}_{\text{II}}(t_i, q_{j+1}) \Rightarrow \neg \text{FAntMem.in}(q_j) \vee \text{duse}(q_j) \Rightarrow \neg \text{FAntMem.in}(q_j) \Rightarrow \neg \text{FAntMem.in}(q_{i+1}), \)
which contradicts \( \text{Insert}_{\text{II}}(q_i, q_{i+1}). \)

From Lemma 3 we have \( \neg \text{Insert}_{\text{II}}^3(t) \) and \( \exists i, \text{Insert}_{\text{II}}(t_i, t_{i+1}). \) Further on:

1) \( \text{Insert}_{\text{II}}(t_i, t_{i+1}) \Rightarrow \text{FAntMem.in}(t_{i+1}) \Rightarrow \text{iuse}(t_\lambda); \)

2) \( \text{Insert}_{\text{II}}(t_i, t_{i+1}) \Rightarrow \neg \text{SafeS.out}(t_i) \vee \text{ddef}(t_i). \) If \( \text{ddef}(t_i) \) is true, then \( Q \not\in \text{CORRECT}, \) because \( Q \) does not add \text{store} on the path \([t_i, t_\lambda]. \) So, \( \text{SafeS.out}(t_i) \) is false;

3) \( \neg \text{SafeS.out}(t_i) \Rightarrow \neg \text{FAvlMem.out}(t_i) \) by Theorem 5 \( \Rightarrow \exists u \in \text{Paths}, \text{ddef}(u_1) \wedge (u_\lambda = t_i) \wedge \neg (\text{ddef} \vee \text{idef})^3([u_1, u_{\lambda-1}]); \)

4) On the path \( \text{ddef} \wedge \text{iuse}, \) each correct placement must put a \text{store}, so
\( \text{Insert}_{\text{III}}^3(u \oplus [t_i, t_\lambda]). \) By the initial assumption, \( \neg \text{Insert}_{\text{III}}^3(t), \) this is why
\( \text{Insert}_{\text{II}}^3(u); \)

5) From (13) \( \Rightarrow \neg \text{SafeS.out}(t_i) \Rightarrow \neg \text{FAntMem.out}(t_i) \Rightarrow \exists v \in \text{Paths}, \) \( (v_1 = t_i) \wedge (\text{ddef}(v_\lambda) \vee \text{idef}(v_\lambda) \vee \text{suc}(v_\lambda) = \emptyset) \wedge \neg \text{iuse}^3(v); \)

6) Now consider a composite path \( u \oplus v. \) On the path, the placement function \( Q \) adds a \text{store} that is not used below. So, \( Q \not\in T3. \)
Theorem 12. The placement function $\Pi$ belongs to $LT_{OPT}$.

Proof. Assume the contrary, $\exists A \in CMP_{OPT}$, $\exists p \in \text{FullPaths}$, $\exists i$ such that $(\langle p_i, p_{i+1} \rangle \not\in \text{LiveArcs}_A(p)) \land (\langle p_i, p_{i+1} \rangle \in \text{LiveArcs}_H(p))$. Let us select a maximal subpath $q = [p_a, p_b]$, such that $q \in \text{LtR}_{\Pi}$ and $p_i, p_{i+1} \in q$. By the nature of the path $p$, we have four cases:

1) $ddef(q_1) \land duse(q_\lambda);
2) ddef(q_1) \land \text{Insert}_S(q_{\lambda-1}, q_\lambda);
3) \text{Insert}_L(q_1, q_2) \land duse(q_\lambda);
4) \text{Insert}_L(q_1, q_2) \land \text{Insert}_S(q_{\lambda-1}, q_\lambda).

Now we consider each case and prove that such $A$ and $p$ do not exist:

1) $ddef(q_1) \land duse(q_\lambda)$. It is obvious that $q \in \text{LtR}_A$, so the initial assumption is incorrect $(\langle p_i, p_{i+1} \rangle \not\in \text{LiveArcs}_A(p))$;
2) $ddef(q_1) \land \text{Insert}_S(q_{\lambda-1}, q_\lambda)$. There are three conclusions from $\text{Insert}_S(q_{\lambda-1}, q_\lambda)$:

(a) $\exists u \in \text{Paths}$ such that $(u_1 = q_{\lambda-1}) \land \neg iuse^3(\lfloor u \rfloor) \land (iuse(f(u_\lambda)) \lor ddef(u_\lambda) \lor (u_\lambda = \text{end}))$;
(b) $\exists z \subseteq p$, $(z_1 = q_\lambda) \land iuse(z_\lambda) \land \neg (ddef \lor iuse^3(\lfloor z \rfloor))$;
(c) $\exists w \in \text{Paths}$, $ddef(w_1) \land \neg (ddef \lor iuse^3(\lfloor w \rfloor)) \land (w_\lambda = q_{\lambda-1})$.

Since the placement function $A$ is correct, we have $\text{Insert}_S^3_A(w \oplus z)$. Since $A \in T3$, we have $\neg \text{Insert}_S^3_A(w \oplus u)$. Combining the previous statements, we have $\exists j, \text{Insert}_S_A(z_j, z_{j+1})$. Now consider a path $q \oplus z$. From the statements above, we conclude that the variable $v_r$ is alive on the edge $(z_j, z_{j+1})$. Then $A \in T1 \Rightarrow \neg \text{Insert}_S^3_A(q \oplus [z_1, z_j])$, so $v_r$ is alive on the whole path $q$, which contradicts the initial assumption.

3) $\text{Insert}_L(q_1, q_2) \land duse(q_\lambda)$. By the initial assumption, $\langle p_i, p_{i+1} \rangle \not\in \text{LiveArcs}_A(p)$. This can happen only if $\exists i \leq j \leq \beta$, $\text{Insert}_L_A(p_j, p_{j+1})$. There are two conclusions from $\text{Insert}_L(q_1, q_2)$:

(a) $\exists k < \alpha$, $iuse(p_k) \land \neg (ddef \lor duse)^3(\lfloor p_k, p_\alpha \rfloor)$;
(b) $\neg F_{AvlMemOnlyIn}(q_2) \lor duse(q_2)$.

Suppose $duse(q_2)$ is true. Consider the path $[p_k, q_2]$ that connects an implicit definition with an explicit use, so $\text{Insert}^3_A([p_k, q_2])$. This is why there is a load added by $A$ on the path $[p_k, q_\lambda]$. As a result, we come to $A \not\in T2$, which contradicts the initial assumption.

Suppose the inverse: $\neg duse(q_2)$. Then $\neg F_{AvlMemOnlyIn}(q_2) \Rightarrow \exists u \in \text{Paths}$, $(duse(u_1) \lor ddef(u_\lambda)) \land (u_\lambda = q_2) \land \neg ddef^3(p)$. 


If $ddef(u_1)$ is true, then $A$ does not insert load on the path $[u_1, q_\lambda]$ (because $A \in T_1$). In other words, the truth of $InsertL_A(p_j, p_{j+1})$ contradicts $A \in T_1$.

If $duse(u_1)$ is true, then (assuming that the source program is correct) there is at least one definition that reaches $u_1$. If the definition is explicit, we have a path $ddef–duse(u_1)–InsertL_A–duse(q_\lambda)$ that contradicts $A \in T_1$. If the definition is implicit, we have a path $idef–InsertL_A–duse(u_1)–InsertL_A–duse(q_\lambda)$, which contradicts $A \in T_2$;

4) $InsertL_{\Pi}(q_1, q_2) \land InsertS_{\Pi}(q_{\lambda-1}, q_{\lambda})$ This case can be proven as a combination of the previous ones:

Because $InsertL_{\Pi}(q_1, q_2)$ holds, there is a $duse$ reachable downwards from $q_2$: $\exists u \in Paths, (u_1=q_2) \land duse(u_\lambda) \land \neg(idef \lor ddef) \exists (|u_1, u_\lambda|)$. Applying the already proved case (3) to the path $u$, we have $\exists k<\alpha$, $InsertL_A(p_k, p_{k+1})$.

Because $InsertS_{\Pi}(q_{\lambda-1}, q_{\lambda})$ holds, there is a $ddef$ that reaches $q_{\lambda-1}$: $\exists w \in Paths, ddef(w_1) \land (w_\lambda = q_{\lambda-1}) \land \neg(idef \lor iuse) \exists (|w_1, u_{\lambda}|)$. Applying the already proved case (2) to the path $w$, we have $\exists t>\beta$, $InsertS_A(p_t, p_{t+1})$.

Now consider the composite path $[p_k, p_t]$. Because $A \in T_2$, we have $\neg InsertL_A^\geq( |p_k, p_t| )$, so the path $[p_k, p_t]$ is a contiguous liveness interval for the placement function $A$.

14. Experiments

<table>
<thead>
<tr>
<th>JRE 1.4.2</th>
<th>Java2Demo</th>
<th>FOP 0.20.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>naive</td>
<td>optimal</td>
<td>naive</td>
</tr>
<tr>
<td>Number of stores</td>
<td>395087</td>
<td>63927</td>
</tr>
<tr>
<td>Number of loads</td>
<td>319454</td>
<td>273274</td>
</tr>
<tr>
<td>Size of generated code, KB</td>
<td>14973.04</td>
<td>14951.70</td>
</tr>
</tbody>
</table>

The most important advantage of the partial SSA form over the traditionally used full SSA is reduction of the IR size, that affects not only the compiler memory consumption, but also speeds up the compilation process. This section presents our measurements made using the Excelsior JET JVM [20].
The data were collected by running the JVM’s static compiler on a set of test applications. The original version uses partial SSA with the “naive” placement function described in [22]. A modified version uses the algorithm described in this paper. Test applications are:

- **JRE1.4.2.07**: Java 2 Platform API classes;
- **Java2Demo**: The Java2Demo sample that comes with the JDK;
- **FOP0.20.5**: freely distributed XML-to-PDF converter.

Table (1) presents the number of loads and stores inserted by each algorithm, and the size of the generated object code. As can be seen, the optimized placement function inserts five times less stores, and reduction of loads count is also noticeable.

![Figure 6. Size of the def-use information](image)

In Figure 6 one can see the size of def-use information (computed as the number of definitions plus the number of uses of all variables) measured on the FOP application for all three kinds of IR we have discussed.

15. Related work

**Derivations of SSA.** The original paper [6] introduces SSA for the programs with scalar variables and explicit definitions/uses only. However, the same paper suggests that implicit definitions/uses may be modelled by extra arguments and results: $(a, b, c) = \text{foo}(a, b, c)$, which are versioned as well. This is exactly what we mean by the full SSA. The paper does not consider address-taking operations.
There are many papers that somehow extend the SSA form, making it more suitable for sophisticated analysis and optimization, with strong accent on dealing with concurrent programs [27, 15]. We restrict the overview by presenting only those papers that extend SSA to cover implicit operations.

One of the later papers by Rod Cytron et. al. [7] models implicit definitions by the function \( \text{IsAlias} \): 
\[
v_1 = \ldots; \quad *p = 1; \quad v_2 = \text{IsAlias}(p, &v_1),
\]
while implicit uses are not considered. The authors build SSA form incrementally, at each iteration it does contain only a part of program semantics, so this kind of IR does not allow using the traditional SSA-based optimization (they have to be adjusted for the IR).

Another extension of SSA [17] incorporates implicit definitions and uses while removing all \( \phi \)-functions, that is, using only variable versions. This way the authors easily explicate implicit dependencies, and they also make versions of what is called “indirect variables”, which are versions of heap locations that are pointed by other variables. One can say that the proposed solution is not an internal representation, but rather a set of attributes over variables and statements that keeps the results of the side effects analysis.

The most notable SSA extension towards indirect operations is presented in the paper of Fred Chow [5]. He adds artificial statements (\( \mu \)- and \( \chi \)-functions) into the program IR that are similar to our loads and stores. Consider a small example: let a call to \( \text{foo}() \) reads and writes a variable \( v \). Here is the HSSA form for the example: 
\[
\mu(v_i); \quad \text{foo}(); \quad v_j = \chi(v_i).
\]
However, the \( \mu \)- and \( \chi \)-functions are placed right around each call (or other indirect memory access), so they are quite numerous and generate a huge number of SSA versions. In our approach, conversely, loads and stores are placed optimally, so the IR size is kept minimal. Besides, the optimal placement of loads and stores simplifies our register allocator. A flavour of the HSSA form that features indirect variables and \( \mu \)- and \( \chi \)-functions is used by the GCC compiler [23].

The idea of separating addressable and non-addressable variables and translating only non-addressable ones to SSA was originally introduced by Pavel Zemtsov in [12], the same paper was the first to mention the insertion of load and store statements as a part of SSA construction. Partial SSA form was implemented in the Sokrat project [24, 11] and then reused in the XDS framework [19] and then in Excelsior JET JVM [20].

**Loads and stores.** The concept of load and store statement that transfers values between registers and memory is not new. Load/store are widely used in the register allocation and code generation algorithms. The standard approach for placing loads and stores is described in the Muchnik bible [22, Section 3.6].

The papers of Lo, Chow et. al [16] and of Cooper and Xu [8] aims at optimization of initially ineffective load/store placement with the help of global value numbering and SSA-based code motion (SSAPRE [4]). We
believe that our approach achieves generally the same results, however we place loads and stores optimally from the very beginning, incorporating the algorithm into SSA construction, reducing the memory consumption and compilation time.

There are works aimed at removal of redundant pointer dereferences that also use the code motion approach, such as the paper of Bodik et al. [2]. They also use the terms load and store, but with different semantics, because the memory address is not fixed so the load statement serves rather as a heap access operator.

16. Conclusion

The paper presents a new internal representation called partial SSA form that is an SSA flavor with some (but not all) variables obeying the single assignment property. The proposed IR is more compact and simplifies the design of certain compiler components, while it does not hinder code optimizations.

We formulated several criteria that help us to select the “best” partial SSA representation for a given program, attempting to minimize the IR size and lifetime ranges of non-addressed variables (virtual registers) while keeping all those def-use chains that are necessary for further optimization.

We presented the algorithm for translation of an arbitrary program to the partial SSA form and formally proved that the resulting IR satisfies the optimality criteria.

Experimental results showing reduction of IR size and some improvement of the code quality are also given.

References


