Bull. Nov. Comp. Center, Comp. Science, 23 (2005), 17–44

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 represetntaion 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 $v = op(a_1, \dots a_N)$, 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 $v = op(a_1, \dots a_N)$ 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 $a_1, \dots a_N$.

The control flow is represented as the function $succ\colon Stms \to 2^{Stms}$. It is assumed that a procedure has exactly one starting node, denoted as **start**, such that $\forall s \in Stms$, $\mathbf{start} \notin succ(s)$ and one terminating node, denoted as **end**, such that $succ(\mathbf{end}) = \emptyset$.

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 version 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 ϕ-function $v_n = \phi(v_1, v_2, \ldots v_m)$ is inserted to merge them into the definition of a new variable version v_m. The ϕ-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, ϕ-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.

4. Terms

We use the following formal terms:

Program (procedure): $Prog = \langle Vars, Stms, succ\rangle$, where $Vars$ is a set of variables, $Stms$ is a set of statements, and $succ$: $Stms \to 2^{Stms}$ defines the control flow.

Statement: $s = \langle r, op, args\rangle \in Stms$, $r \in Vars$, $args = \langle a_0, \dots a_N\rangle$, $a_i \in Vars$[2], $op \in Ops$ (the nature of the Ops set is not important).

CFG is defined by a pair of functions $pred, succ$: $Stms \to 2^{Stms}$, where $succ$ is a part of $Prog$, while $pred$ is defined as $s \in pred(p) \Leftrightarrow p \in succ(s)$. Let **start** and **end** denote the unique *start node* and *end node* of the CFG.

Predicates on statements. There are four predicates that are important for the further account: $ddef, idef, iuse, duse : Stms * Vars \to \{0,1\}$, where: $ddef(\langle r, op, args\rangle, v) \stackrel{\text{def}}{\equiv} (r = v)$ denotes explicit results, $duse(\langle r, op, args\rangle, v) \stackrel{\text{def}}{\equiv} (v \in args)$ denotes explicit arguments, and $idef$, $iuse$ 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:

$$\forall\, v \in Vars, \{s | s \in Stms, ddef(s, v) \wedge idef(s, v)\} = \emptyset.$$

Execution paths. Define $Paths$ as a set of all (finite) paths over the CFG, including the cyclic ones: $Paths = \{p = \langle p_1, \dots p_N\rangle \mid \forall\ 1 \le i \le N,\ p_{i+1} \in succ(p_i)\}$. 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_λ where appropriate. Then, define $Paths[s_1, s_2]$ as a set of all finite paths starting with s_1 and ending in s_2: $Paths[s_1, s_2] = \{p | p \in Paths, p_1 = s_1, p_\lambda = s_2\}$. A path q is said to be a subpath of p, in signs $q \sqsubseteq p$, if there is an index $1 \le i \le \lambda p$ such that $i + \lambda q - 1 \le \lambda p$ and $q_j = p_{i+j-1}$ for all $1 \le j \le \lambda q$. In particular, for any path p and indices $i, j \le \lambda p$ 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 $p \oplus q = [p_1, \dots p_\lambda, q_2, \dots q_\lambda]$.

Predicates on paths. Let a be some predicate on statements, e.g. $a : Stms \to \{0,1\}$, and let $p \in Paths$. 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 $b : Stms * Stms \to \{0,1\}$:

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

[2]For simplicity, we represent constants as variables.

$\exists\ 1{\le}i{\le}\lambda(p),\ b(p_i, p_{i+1}) \Leftrightarrow b^{\exists}(p)$

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

$$\neg c^{\forall}(p) \Leftrightarrow \neg(\forall\ s \in p,\ c(s)) \qquad (\neg c)^{\forall}(p) \Leftrightarrow \neg c^{\exists}(p)$$
$$(\neg c)^{\forall}(p) \Leftrightarrow \forall\ s \in p,\ \neg c(s) \qquad (\neg c)^{\exists}(p) \Leftrightarrow \neg c^{\forall}(p).$$

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:

$$\begin{cases} A\_in(t) \ = \begin{cases} false & \text{if } t = \textbf{start}, \\ \bigwedge\limits_{p \in pred(t)} A\_out(p) \setminus A^{-}_{arc}(p,t) \vee A^{+}_{arc}(p,t) & \text{otherwise}, \end{cases} \\ A\_out(t) = A\_in(t) \setminus A^{-}_{nd}(t) \vee A^{+}_{nd}(t). \end{cases} \tag{1}$$

Attribute $A\_in(s)$ specifies the value of A before the execution of s, while $A\_out(s)$ stands for the value of A after s. Single-argument predicates $A^{-}_{nd}(t)$ and $A^{+}_{nd}(t)$ contribute to the node transition function, while two-argument predicates $A^{-}_{arc}(p,t)$ and $A^{+}_{arc}(p,t)$ 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

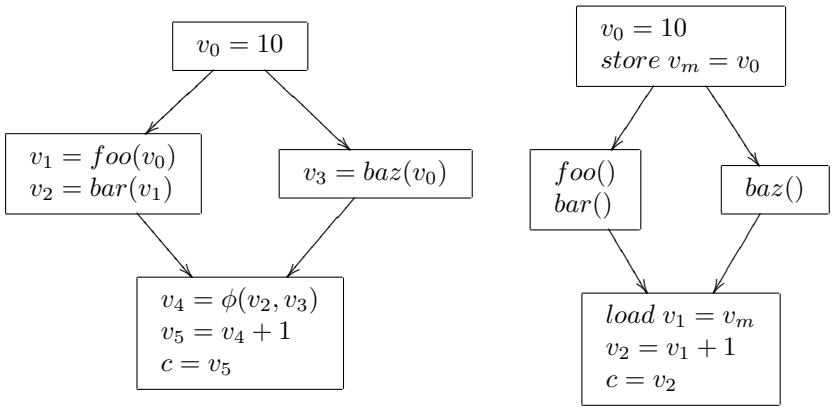
A program with implicit results and arguments can be directly translated to full SSA by applying versioning to all variables, rewriting all implicit and explicit definitions and uses, as it was suggested in [6]. Figure 2a shows the full SSA form for the source program in Figure 1. However, there are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



a) Full SSA form b) Partial SSA form

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 loose 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:

$InsertL, InsertS : Stms * Stms * Vars \Rightarrow \{0, 1\}$.

If $InsertL(s, p, v) = 1$, 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 $InsertS(s, p, v) = 1$, then an assignment $v_m = v_r$ is inserted.

We call the pair $\langle InsertL, InsertS \rangle$ 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: $P \stackrel{\text{def}}{=} \langle InsertL_P, InsertS_P \rangle$.

Further on we skip the program symbol $Prog$ and variable symbols v, v_r and v_m[3]. In other words, we assume that all argumentation is applied to an arbitrary (but fixed at this moment) program $Prog$ 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, Chapter 8] independently for addressed and non-addressed variables.

$$\begin{cases} AvailReg\_in(t) & = \begin{cases} false & \text{if } t = \textbf{start}, \\ \bigwedge\limits_{p \in pred(t)} (AvailReg\_out(p) \vee InsertL(p, t)) & \text{otherwise,} \end{cases} \\ AvailReg\_out(t) & = (AvailReg\_in(t) \vee ddef(t)) \setminus idef(t); \end{cases} \tag{2}$$

$$\begin{cases} AvailMem\_in(t) & = \begin{cases} true & \text{if } t = \textbf{start}, \\ \bigwedge\limits_{p \in pred(t)} (AvailMem\_out(p) \vee InsertS(p, t)) & \text{otherwise,} \end{cases} \\ AvailMem\_out(t) & = (AvailMem\_in(t) \vee idef(t)) \setminus ddef(t). \end{cases} \tag{3}$$

Consider the first equation. The explicit definition ($ddef$) contributes *true* to the solution, while implicit definition ($idef$) resets the solution to

[3] e.g. we replace $idef(s, v)$ by $idef(s)$.

$false$. So, $AvailReg\_in(s)$ 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]:

$$\forall s \in Stms \begin{cases} duse(s) & \leq AvailReg\_in(s), \\ iuse(s) & \leq AvailMem\_in(s); \end{cases} \tag{4}$$

$$\forall s_1 \in Stms, \forall s_2 \in succ(s_1) \begin{cases} InsertS(s_1, s_2) & \leq AvailReg\_out(s_1), \\ InsertL(s_1, s_2) & \leq AvailMem\_out(s_1). \end{cases}$$

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

For a program $Prog$, we denote a set of all correct placement functions as $CORRECT(Prog)$.

8. Optimality criteria

Obviously, there is more than one correct placement function for a given program $Prog$. 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.

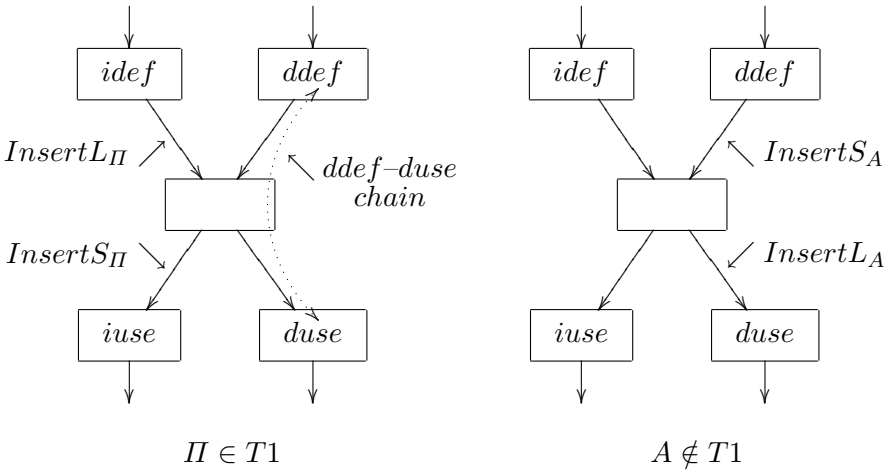


Figure 3. An example of def-use complete placement function

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_λ, 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 $T1$:

$$\begin{aligned} A \in T1 \stackrel{\text{def}}{\equiv} & \forall p \in Paths \\ & ddef(p_1) \ \wedge \ \neg(ddef \vee idef)^{\exists}(\,]p[\,) \ \wedge \ duse(p_\lambda) \\ & \Rightarrow \ \neg InsertL_A^{\exists}(\,]p[\,). \end{aligned} \tag{5}$$

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 (Π) 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 $T2$:

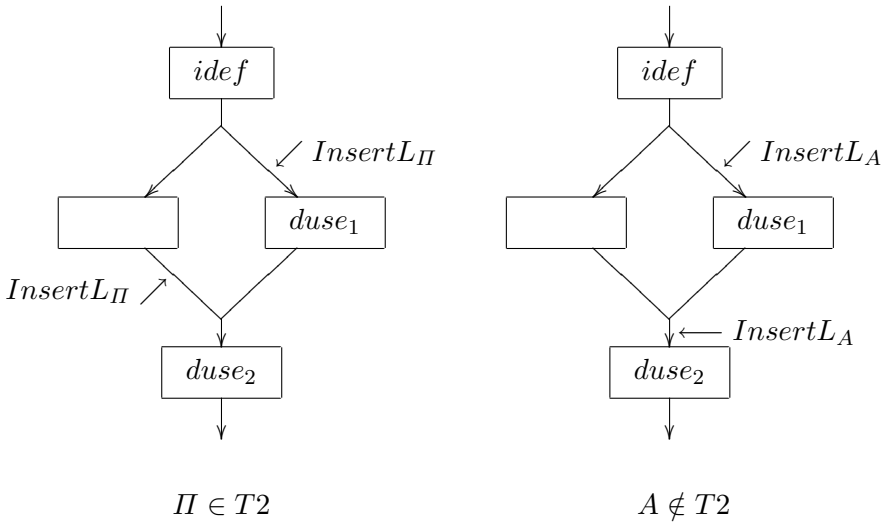


Figure 4. A placement function with redundant loads

$$A \in T2 \stackrel{\text{def}}{\equiv} A \in T1 \ \wedge \forall p \in Paths \\ InsertL_A(p_1, p_2) \ \wedge \ InsertL_A(p_{\lambda-1}, p_\lambda) \Rightarrow idef^{\exists}(\,]p[\,). \quad (6)$$

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 Π 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 $T3$:

$$A \in T3 \stackrel{\text{def}}{\equiv} A \in T2 \wedge \\ \forall s_1, s_2 \in Stms, InsertS_A(s_1, s_2) \Rightarrow \\ \forall p \in Paths(s_2, \textbf{end}), \ \exists j{<}\lambda(p), \\ \neg idef^{\exists}(\,[p_1, p_j[\,) \ \wedge \ \neg InsertS_A^{\exists}(\,[p_1, p_j]\,) \ \wedge \ iuse(p_j). \quad (7)$$

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 $\leftrightarrow$ 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 Π inserts load and store out of the loop, while the placement function A moves them into the loop. So, Π is computationally better.

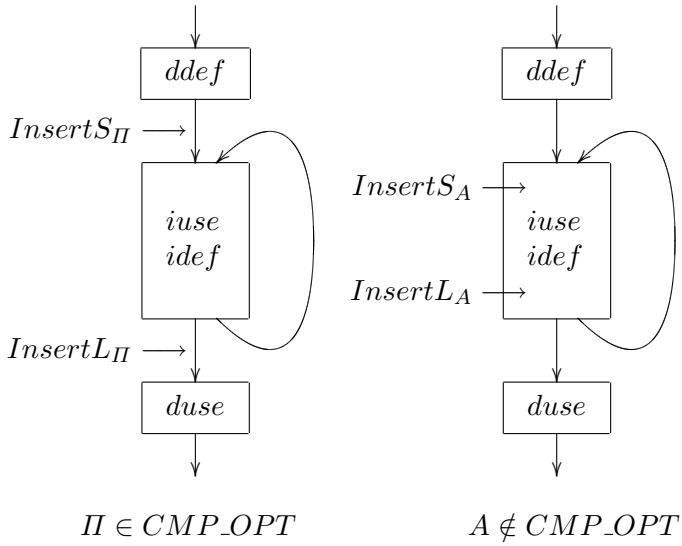


Figure 5. An example of a computationally optimal placement function

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

$$count\_loads_A(p) \stackrel{\text{def}}{\equiv} |\{\ i \mid 1{\le}i{\le}\lambda(p),\ InsertL_A(p_i, p_{i+1})\}|,$$
$$count\_stores_A(p) \stackrel{\text{def}}{\equiv} |\{\ i \mid 1{\le}i{\le}\lambda(p),\ InsertS_A(p_i, p_{i+1})\}|.$$

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

$$\begin{array}{rcll} A \le_L B & \stackrel{\text{def}}{\equiv} & \forall p \in Paths[\mathbf{start}, \mathbf{end}] & count\_loads_A(p) \le count\_loads_B(p), \\ A \le_S B & \stackrel{\text{def}}{\equiv} & \forall p \in Paths[\mathbf{start}, \mathbf{end}] & count\_stores_A(p) \le count\_stores_B(p), \\ A \le_{comp} B & \stackrel{\text{def}}{\equiv} & (A \le_L B) \wedge (A \le_S B). & \end{array}$$

The relationship $\le_{comp}$ generates a partial order on $T3$. Denote the set of minimal elements as CMP_OPT:

$$CMP\_OPT \stackrel{\text{def}}{\equiv} \{A \mid A \in T3, \forall\ B \in T3,\ A \le_{comp} B\}. \tag{8}$$

Further on we prove that CMP_OPT is not empty.

Note that all placement functions that belong to CMP_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 *load*),
- v_r is used at the path end (either explicitly or by *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 $LtRg$:

$$\begin{aligned} LtRg_A \stackrel{\text{def}}{\equiv} \{ p \mid p \in Paths, \\ (ddef(p_1) \vee InsertL_A(p_1, p_2)) \wedge \\ (duse(p_\lambda) \vee InsertS_A(p_{\lambda-1}, p_\lambda)) \wedge \\ \neg ddef^{\exists}( ]p[ ) \ \wedge \ \neg InsertL_A^{\exists}( ]p]) \\ \}. \end{aligned} \tag{9}$$

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:

$$\begin{aligned} LiveArcs_A(p \in Paths[\mathbf{start}, \mathbf{end}]) \stackrel{\text{def}}{\equiv} \{ \langle a, b \rangle \mid \exists\, q \in LtRg_A, q \sqsubseteq p, \\ \exists\, i, a = q_i, b = q_{i+1} \}. \end{aligned} \tag{10}$$

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

$$A \leq_{LT} B \stackrel{\text{def}}{\equiv} \forall\, p \in Paths[\mathbf{start}, \mathbf{end}], LiveArcs_A(p) \subseteq LiveArcs_B(p),$$

$$LT\_OPT \stackrel{\text{def}}{\equiv} \{ A \mid A \in CMP\_OPT \wedge (\forall\, B \in CMP\_OPT \quad A \leq_{LT} B) \}.$$

Further we build the placement function Π that belongs to 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 Prog can be solved by computing a placement function that belongs to 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 **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:

$$UpSafe\_in(s) \Leftrightarrow \forall p \in Paths[\mathbf{start}, s] \\ \exists i{<}\lambda p, Comp(p_i) \wedge Transp^{\forall}([p_i, p_\lambda[).$$

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

[5]Effective implementation runs many systems in parallel using the bit vector-based approach.

$$DownSafe\_out(s) \Leftrightarrow \forall p \in Paths[s, \mathbf{end}] \; \exists i{<}\lambda p, Comp(p_i) \wedge Transp^{\forall}(]p_1, p_i[).$$

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 occurence 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 $\langle a, b\rangle$ such that $a \notin Safe \wedge b \in Safe$. Finally, $Replace = Comp$.

$$\begin{cases} UpSafe\_in(t) = \begin{cases} false & \text{if } t = \mathbf{start}, \\ \bigwedge\limits_{p \in pred(t)} UpSafe\_out(p) & \text{otherwise}, \end{cases} \\ UpSafe\_out(t) = (UpSafe\_in(t) \vee Comp(t)) \wedge Transp(t); \end{cases} \tag{11}$$

$$\begin{cases} DownSafe\_in(t) = (DownSafe\_out(t) \wedge Transp(t)) \vee Comp(t), \\ DownSafe\_out(t) = \begin{cases} false & \text{if } t = \mathbf{end}, \\ \bigwedge\limits_{p \in succ(t)} DownSafe\_in(p) & \text{otherwise}; \end{cases} \end{cases} \tag{12}$$

$$\begin{aligned} Safe\_in(t) &= UpSafe\_in(t) \vee DownSafe\_in(t), \\ Safe\_out(t) &= UpSafe\_out(t) \vee DownSafe\_out(t); \end{aligned}$$

$$\begin{aligned} Insert_{BCM}(p, q) &= p \in pred(q) \wedge \\ &\quad (\neg Safe\_out(p) \vee \neg Transp(p)) \wedge Safe\_in(q), \\ Replace_{BCM}(p) &= Comp(p). \end{aligned}$$

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 $Insert_{BCM}$ and $Replace_{BCM}$ 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 $Insert_{BCM}$ and $Replace_{BCM}$ 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 $Insert_{BCM}$ and $Replace_{BCM}$ 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 LT_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 ($ddef$), implicit definition ($idef$), explicit use ($duse$) and implicit use ($iuse$). 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 *full anticipabilty* for the addressed variable, i.e., the presence of uses of the addressed variable on each outgoing path:

$$\begin{aligned} FAntMem\_out(t \in Stms) \;\Leftrightarrow\; & \forall p \in Paths[t, \mathbf{end}] \\ & \exists i{>}1,\; iuse(p_i) \;\wedge\; \neg(ddef \vee idef)^{\exists}(\,[p_1, p_i[\,). \end{aligned}$$

The next system (14) computes *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:

$$\begin{aligned} FAvlMem\_in(t \in Stms) \;\Leftrightarrow\; & \forall p \in Paths[\mathbf{start}, t] \exists i, \\ & (iuse(p_i) \vee idef(p_i)) \;\wedge\; \neg(ddef \vee idef)^{\exists}(\,]p_i, t[\,); \end{aligned}$$

$$\begin{cases} FAntMem\_in(t) = FAntMem\_out(t) \setminus ddef(t) \setminus idef(t) \vee iuse(t), \\ FAntMem\_out(t) = \begin{cases} false & \text{if } t = \mathbf{end}, \\ \bigwedge\limits_{p \in succ(t)} FAntMem\_in(p) & \text{otherwise;} \end{cases} \end{cases} \tag{13}$$

$$\begin{cases} FAvlMem\_in(t) = \begin{cases} true & \text{if } t = \textbf{start}, \\ \bigwedge\limits_{p \in pred(t)} FAvlMem\_out(p) & \text{otherwise}, \end{cases} \\ FAvlMem\_out(t) = (FAvlMem\_in(t) \vee idef(t) \vee iuse(t)) \setminus ddef(t). \end{cases} \tag{14}$$

The set of points where $FAntMem(s) \vee FAvlMem(s)$ 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.

$$SafeS\_in(t \in Stms) \stackrel{\text{def}}{\equiv} FAntMem\_in(t) \vee FAvlMem\_in(t),$$
$$SafeS\_out(t \in Stms) \stackrel{\text{def}}{\equiv} FAntMem\_out(t) \vee FAvlMem\_out(t),$$

$$InsertS_{\Pi}(p, s \in Stms) \stackrel{\text{def}}{\equiv} s \in succ(p) \wedge (\neg SafeS\_out(p) \vee ddef(p)) \wedge SafeS\_in(s)). \tag{15}$$

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

$$\forall s \in Stms\ FAvlMemOnly\_in(s) \leq FAvlMem\_in(s).$$

Let us see the difference beween the systems: $FAvlMemOnly\_in(s)$ is true only if the actual value of the variable at the point s is available in memory *only*, while $FAvlMem\_in(s)$ is true if the actual value is available in memory only, or if it is available in memory and register at the same time.

$$\begin{cases} PAntReg\_in(t) = PAntReg\_out(t) \setminus ddef(t) \setminus idef(t) \vee duse(t), \\ PAntReg\_out(t) = \begin{cases} false & \text{if } t = \textbf{end}, \\ \bigvee\limits_{p \in succ(t)} (PAntReg\_in(p) \vee InsertS_{\Pi}(t, p)) & \text{otherwise;} \end{cases} \end{cases} \tag{16}$$

$$\begin{cases} FAvlMemOnly\_in(t) = \begin{cases} true & \text{if } t = \textbf{start}, \\ \bigwedge\limits_{p \in pred(t)} FAvlMemOnly\_out(p) & \text{otherwise}, \end{cases} \\ FAvlMemOnly\_out(t) = (FAvlMemOnly\_in(t) \vee idef(t)) \setminus (ddef(t) \vee duse(t)); \end{cases} \tag{17}$$

$$InsertL_{\Pi}(p, s \in Stms) \stackrel{\text{def}}{\equiv} s \in succ(p) \wedge FAvlMemOnly\_out(p) \wedge ((PAntReg\_in(s) \wedge \neg FAvlMemOnly\_in(s)) \vee duse(s)). \tag{18}$$

From now on we denote the placement function $\langle InsertL_{\Pi}, InsertS_{\Pi} \rangle$ as Π.

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

of predicates: $\neg Transp$ is replaced by $(ddef \vee idef)$, $Comp$ is replaced by $iuse \vee idef$. However, the equations for the *load* placement are different, because one of them employs $\vee$ 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:

$$\begin{cases} A\_in(t) & = \begin{cases} false & \text{if } t = \textbf{start}, \\ \bigwedge\limits_{p \in pred(t)} A\_out(p) \setminus A^-_{arc}(p,t) \vee A^+_{arc}(p,t) & \text{otherwise,} \end{cases} \\ A\_out(t) & = A\_in(t) \setminus A^-_{nd}(t) \vee A^+_{nd}(t). \end{cases} \tag{19}$$

Theorem 4. *For a statement s the following assertions are equivalent:*

- *$A\_in(t)$ is true;*
- $\forall p \in Paths[\textbf{start}, t]\ \exists i$,
  $(A^+_{nd}(p_i) \vee A^+_{arc}(p_{i-1}, p_i)) \ \wedge\ \neg {A^-_{nd}}^{\exists}([p_i, p_\lambda[\ ) \ \wedge\ \neg {A^-_{arc}}^{\exists}([p_i, p_\lambda]\ )$.

Theorem 5. *For a statement s the following assertions are equivalent:*

- *$A\_in(t)$ is false;*
- $\exists p \in Paths$, *such that* $p_\lambda = t$,
  $((A^-_{nd}(p_1) \vee A^-_{arc}(p_1, p_2)) \ \wedge \neg {A^+_{nd}}^{\exists}([p[\ ) \ \wedge\ \neg {A^+_{arc}}^{\exists}(p)$.

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

$$\begin{cases} B\_in(t) & = B\_out(t) \setminus B^-_{nd}(t) \vee B^+_{nd}(t), \\ B\_out(t) & = \begin{cases} false & \text{if } t = \textbf{end}, \\ \bigwedge\limits_{p \in succ(t)} B\_in(p) \setminus B^-_{arc}(p,t) \vee B^+_{arc}(p,t) & \text{otherwise.} \end{cases} \end{cases} \tag{20}$$

Theorem 6. *For a statement s the following assertions are equivalent:*

- *$B\_in(t)$ is true;*
- $\forall p \in Paths[t, \textbf{end}]\ \exists i \le \lambda(p)$,
  $\neg {B^-_{nd}}^{\exists}([p_1, p_i[\ ) \ \wedge\ \neg {B^-_{arc}}^{\exists}([p_1, p_i]\ ) \ \wedge\ (B^+_{nd}(p_i) \vee B^+_{arc}(p_{i-1}, p_i))$.

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* $\Pi = \langle InsertL_\Pi, InsertS_\Pi \rangle$*, 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 $\exists s_1, s_2 \in Stms, InsertS_\Pi(s_1, s_2) \wedge \neg AvailReg\_out(s_1)$.

  1. $InsertS(s_1, s_2) \Rightarrow \neg SafeS\_out(s_1) \vee ddef(s_1) \Rightarrow \neg FAvlMem\_out(s_1) \Rightarrow \exists q \in Paths, ddef(q_1) \ \wedge \ (q_\lambda = s_1) \wedge \neg(idef \vee iuse)^\exists(q)$.
  2. $\neg AvailReg\_out(s_1) \Rightarrow \exists p \in Paths, \ idef(p_1) \ \wedge \ (p_\lambda = s_1) \ \wedge \neg(ddef \vee InsertL_\Pi)^\exists(p)$.
  3. Let p_i be the first common statement of the paths q and p. From (1) and (2) $\Rightarrow \neg FAvlMemOnly\_in(p_i)$.
  4. $InsertS_\Pi(s_1, s_2)! \Rightarrow PAntReg\_in(s_1 = p_\lambda) \xrightarrow{\text{from (2)}} PAntReg\_out(p_1)$.
  5. (2) $\Rightarrow FAvlMemOnly\_out(p_1)$.
  6. (2) $\Rightarrow PAntReg^\forall(p)$.
  7. According to (3), (5) and (6) the path $[p_1, p_i]$ contains an edge where *FAvlMemOnly* changes its value, and $PAntReg$ remains true. So, $InsertL_\Pi$ is true on the edge, which contradicts (2).

- Assume $\exists s_1, s_2 \in Stms, InsertL_\Pi(s_1, s_2) \wedge \neg AvailMem\_out(s_1)$.

  1. $InsertL_\Pi(s_1, s_2) \Rightarrow FAvlMemOnly\_out(s_1)$.
  2. Comparing the equations for *FAvlMemOnly* and *AvailMem*, we have $\forall s \in Stms, FAvlMemOnly\_out(s) \Rightarrow AvailMem\_out(s)$.
  3. $InsertL_\Pi(s_1, s_2) \Rightarrow AvailMem\_out(s_1)$, which contradicts the assumption above.

- Assume $\exists s \in Stms, duse(s) \wedge \neg AvailReg\_in(s)$.

  1. Applying the auxiliary theorem (5) to the definition of *AvailReg*, we have $\exists p \in Paths, \ idef(p_1), p_\lambda = s_1, \neg(ddef \vee InsertL_\Pi)^\exists(p)$.
  2. (1) and $duse(s) \Rightarrow PAntReg^\forall(p)$.

3. $idef(p_1) \Rightarrow FAvlMemOnly\_out(p_1)$.
4. (2), (3) and $\neg InsertL_{\Pi}^{\exists}(p) \Rightarrow FAvlMemOnly^{\forall}(p)$.
5. From (4) and definition of $InsertL_{\Pi} \Rightarrow InsertL_{\Pi}(p_{\lambda-1}, p_{\lambda})$, which contradicts the definition of the path p.

- Assume $\exists s \in Stms,\ iuse(s) \wedge \neg AvailMem\_in(s)$.

1. $iuse(s) \Rightarrow FAntMem\_in(s) \Rightarrow SafeS\_in(s)$.
2. Applying the auxiliary theorem (5) to the definition of $AvailMem$, we have $\exists p \in Paths,\ ddef(p_1) \wedge (p_{\lambda} = s) \wedge \neg(idef \vee InsertS_{\Pi})^{\exists}(p)$.
3. $\neg InsertS_{\Pi}^{\exists}(p) \xrightarrow{\text{from (1)}} SafeS\_in^{\forall}(\ ]p]\ )$.
4. $ddef(p_1) \wedge SafeS\_in(p_2) \Rightarrow InsertS_{\Pi}(p_1, p_2), which$ contradicts (2).

13. Proofs of optimality

Theorem 8. *The placement function Π belongs to $T1$.*

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

$$InsertL(p_i, p_{i+1}) \Rightarrow FAvlMemOnly\_out(p_i).$$

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

Theorem 9. *The placement function Π belongs to $T2$.*

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

Theorem 10. *The placement function Π belongs to $T3$.*

Proof. This statement follows from the structure of equations (14): $InsertS(s_1, s_2) \Rightarrow SafeS\_in(s_1) \wedge \neg SafeS\_out(s_2) \Rightarrow FAntMem\_in(s_2)$. Using the auxiliary Theorem 6 to the last statement, we come to the definition of $T3$.

Lemma 1. *Any placement function that belongs to CMP_OPT places load statements only on those edges where PAntReg is true:* $\forall P \in CMP\_OPT$, $\forall s_1, s_2 \in Stms, InsertL_P(s_1, s_2) \Rightarrow PAntReg\_in(s_2) \wedge PAntReg\_out(s_1)$.

Proof. If there are P, s_1, s_2 contradictory to the statement above, then we can built P': $InsertL_{P'} = InsertL_P \setminus \{(s_1, s_2)\}$. The correctness of P' is natural consequence of $\neg PAntReg\_out(s_1)$, because the edge (s_1, s_2) does not belong to any $idef$–$duse$ path. Obviously, P' is computationally better than P, so $P \notin CMP\_OPT$. Finally, $PAntReg\_out(s_1)$ is an obvious consequence of $PAntReg\_in(s_2)$.

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

Proof. We prove separately that Π places the minimal amount of (1) *loads* and (2) *stores*.
Case 1:
$\forall Q \in T3, \forall p \in Paths[\mathbf{start}, \mathbf{end}], count\_loads_Q(p) \geq count\_loads_\Pi(p)$.
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:
$(t \sqsubseteq p) \ \wedge \ (t_1) = \mathbf{start} \vee idef(t_1) \vee ddef(t_1)) \ \wedge \ (t_\lambda = \mathbf{end} \vee idef(t_\lambda) \vee ddef(t_\lambda)) \ \wedge \ count\_loads_Q(t) < count\_loads_\Pi(t)$.

Lemma 2. $count\_loads_\Pi(t) \leq 1$.

Proof. Let us assume the contrary:

$$\exists i, j, \ i<j \wedge InsertL_\Pi(t_i, t_{i+1}) \wedge InsertL_\Pi(t_j, t_{j+1}).$$

Then
$InsertL_\Pi(t_i, t_{i+1}) \Rightarrow \neg FAvlMemOnly\_in(t_{i+1}) \vee duse(t_{i+1}) \Rightarrow$
$\neg FAvlMemOnly\_out(t_{i+1}) \Rightarrow \neg FAvlMemOnly\_in(t_j)$
that contradicts $InsertL_\Pi(t_j, t_{j+1})$.

From Lemma 2 we have $count\_loads_Q(t) = 0$ and $count\_loads_\Pi(t) = 1$, that is, $InsertL^{\exists}_\Pi(t)$ and $\neg InsertL^{\exists}_Q(t)$. Assume the placement function Π inserts *load* between t_i and t_{i+1}. Then:

1) $\neg InsertL_Q)^{\exists}(t) \xrightarrow{Q \in CORRECT} \neg duse(t_{i+1}) \Rightarrow$
$PAntReg\_in(t_{i+1}) \wedge \neg FAvlMemOnly\_in(t_{i+1}) \wedge FAvlMemOnly_out(t_i)$;
2) from (13) $\Rightarrow \neg FAvlMemOnly\_in(t_{i+1}) \Rightarrow \exists q \in Paths, \ ddef(q_1) \wedge (q_\lambda = t_{i+1}) \wedge \neg(ddef \vee InsertL_\Pi)^{\exists}(q)$;

3) from (13) $\Rightarrow PAntReg\_in(t_{i+1}) \Rightarrow \exists s \in Paths,\ (s_1 = t_{i+1}) \wedge (duse(s_\lambda) \vee InsertS_\Pi(s_{\lambda-1}, s_\lambda)) \wedge \neg(ddef \vee InsertL_\Pi)^\exists(s)$;
4) consider a compound path $m = [t_1, \ldots, t_{i+1} = s_1, \ldots, s_\lambda]$. From (13) $\Rightarrow FAvlMemOnly\_out(t_i) \Rightarrow \neg ddef(t_1) \Rightarrow idef(t_1) \xrightarrow{\text{by definition of } s} InsertL_Q^\exists(m)$. By the initial assumption, $\neg InsertL_Q^\exists(t) \Rightarrow InsertL_Q^\exists(p)$;
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 $InsertL_Q^\exists(n)$. It means that $Q \in T1$ is false, which contradicts the initial assumption.

Case 2: $\forall Q \in T3, \forall p \in FullPaths, count\_stores_Q(p) \geq count\_stores_\Pi(p)$.
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:
$(t \sqsubseteq p) \wedge (iuse(t_1) \vee t_1 = \mathbf{start}) \wedge (iuse(t_\lambda) \vee t_\lambda = \mathbf{end}) \wedge$
$\neg iuse^\exists(\,]t_1, t_\lambda[\,) \wedge count\_stores_Q(t) < count\_stores_\Pi(t)$.

Lemma 3. $count\_stores_\Pi(t) \leq 1$.

Proof. Assume the contrary:
$\exists i, j,\ i<j \wedge InsertS_\Pi(t_i, t_{i+1}) \wedge InsertS_\Pi(t_j, t_{j+1})$.
Then
$InsertS_\Pi(t_j, q_{j+1}) \Rightarrow \neg FAntMem\_in(q_j) \vee duse(q_j) \Rightarrow \neg FAntMem\_in(q_j) \Rightarrow \neg FAntMem\_in(q_{i+1})$,
which contradicts $InsertS_\Pi(q_i, q_{i+1})$.

From Lemma 3 we have $\neg InsertS_Q^\exists(t)$ and $\exists! i,\ InsertS_\Pi(t_i, t_{i+1})$. Further on:

1) $InsertS_\Pi(t_i, t_{i+1}) \Rightarrow FAntMem\_in(t_{i+1}) \Rightarrow iuse(t_\lambda)$;
2) $InsertS_\Pi(t_i, t_{i+1}) \Rightarrow \neg SafeS\_out(t_i) \vee ddef(t_i)$. If $ddef(t_i)$ is true, then $Q \notin CORRECT$, because Q does not add *store* on the path $[t_i, t_\lambda]$. So, $SafeS\_out(t_i)$ is false;
3) $\neg SafeS\_out(t_i) \Rightarrow \neg FAvlMem\_out(t_i) \xrightarrow{\text{by Theorem 5}} \exists u \in Paths,\ ddef(u_1) \wedge (u_\lambda = t_i) \wedge \neg(ddef \vee idef)^\exists(\,[u_1, u_{\lambda-1}]\,)$;
4) On the path *ddef–iuse*, each correct placement must put a *store*, so $InsertS_Q^\exists(u \oplus [t_i, t_\lambda])$. By the initial assumption, $\neg InsertS_Q^\exists(t)$, this is why $InsertS_Q^\exists(u)$;
5) From (13) $\Rightarrow \neg SafeS\_out(t_i) \Rightarrow \neg FAntMem\_out(t_i) \Rightarrow \exists v \in Paths,\ (v_1 = t_i) \wedge (ddef(v_\lambda) \vee idef(v_\lambda) \vee succ(v_\lambda) = \emptyset) \wedge \neg iuse^\exists(v)$;
6) Now consider a composite path $u \oplus v$. On the path, the placement function Q adds a *store* that is not used below. So, $Q \notin T3$.

Theorem 12. *The placement function Π belongs to LT_OPT.*

Proof. Assume the contrary, $\exists A \in CMP\_OPT$, $\exists p \in FullPaths$, $\exists i$ such that $(\langle p_i, p_{i+1}\rangle \notin LiveArcs_A(p)) \wedge (\langle p_i, p_{i+1}\rangle \in LiveArcs_\Pi(p))$. Let us select a maximal subpath $q = [p_\alpha, p_\beta]$, such that $q \in LtRg_\Pi$ and $p_i, p_{i+1} \in q$. By the nature of the path p, we have four cases:

1) $ddef(q_1) \wedge duse(q_\lambda)$;
2) $ddef(q_1) \wedge InsertS_\Pi(q_{\lambda-1}, q_\lambda)$;
3) $InsertL_\Pi(q_1, q_2) \wedge duse(q_\lambda)$;
4) $InsertL_\Pi(q_1, q_2) \wedge InsertS_\Pi(q_{\lambda-1}, q_\lambda)$.

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

1) $ddef(q_1) \wedge duse(q_\lambda)$. It is obvious that $q \in LtRg_A$, so the initial assumption is incorrect ($\langle p_i, p_{i+1}\rangle \in LiveArcs_A(p)$);
2) $ddef(q_1) \wedge InsertS_\Pi(q_{\lambda-1}, q_\lambda)$. There are three conclusions from $InsertS_\Pi(q_{\lambda-1}, q_\lambda)$:

   (a) $\exists u \in Paths$ such that $(u_1 = q_{\lambda-1}) \wedge \neg iuse^{\exists}(\ ]u[\ ) \wedge (idef(u_\lambda) \vee ddef(u_\lambda) \vee (u_\lambda = \mathbf{end}))$,
   (b) $\exists z \sqsubseteq p,\ (z_1 = q_\lambda) \wedge iuse(z_\lambda) \wedge \neg(ddef \vee idef)^{\exists}(\ [z[\ )$,
   (c) $\exists w \in Paths,\ ddef(w_1) \wedge \neg(idef \vee iuse)^{\exists}(\ ]w]\ ) \wedge (w_\lambda = q_{\lambda-1})$.

   Since the placement function A is correct, we have $InsertS_A^{\exists}(w \oplus z)$. Since $A \in T3$, we have $\neg InsertS_A^{\exists}(w \oplus u)$. Combining the previous statements, we have $\exists j, InsertS_A(z_j, z_{j+1})$. Now consider a path $q \oplus z$. From the statements above, we conclude that the variable v_r is alive on the edge $\langle z_j, z_{j+1}\rangle$. Then $A \in T1 \Rightarrow \neg InsertL_A^{\exists}(\ q \oplus [z_1, z_j]\ )$, so v_r is alive on the whole path q, which contradicts the initial assumption.
3) $InsertL_\Pi(q_1, q_2) \wedge duse(q_\lambda)$. By the initial assumption, $\langle p_i, p_{i+1}\rangle \notin LiveArcs_A(p)$. This can happen only if $\exists i{\leq}j{\leq}\beta,\ InsertL_A(p_j, p_{j+1})$. There are two conclusions from $InsertL_\Pi(q_1, q_2)$:

   (a) $\exists k{<}\alpha,\ idef(p_k) \wedge \neg(ddef \vee duse)^{\exists}(\ ]p_k, p_\alpha]\ )$,
   (b) $\neg FAvlMemOnly\_in(q_2) \vee duse(q_2)$.

   Suppose $duse(q_2)$ is true. Consider the path $[p_k, q_2]$ that connects an implicit definition with an explicit use, so $InsertL_A^{\exists}(\ [p_k, q_2]\ )$. This is why there is a *load* added by A on the path $[p_k, q_\lambda]$. As a result, we come to $A \notin T2$, which contradicts the initial assumption.

   Suppose the inverse: $\neg duse(q_2)$. Then $\neg FAvlMemOnly\_in(q_2) \rightarrow \exists u \in Paths,\ \ (duse(u_1) \vee ddef(u_1)) \wedge (u_\lambda = q_2) \wedge \neg ddef^{\exists}(p)$.

If $ddef(u_1)$ is true, then A does not insert *load* on the path $[u_1, q_\lambda]$ (because $A \in T1$). In other words, the truth of $InsertL_A(p_j, p_{j+1})$ contradicts $A \in T1$.

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 $ddef$–$duse(u_1)$–$InsertL_A(p_j, p_{j+1})$–$duse(q_\lambda)$ that contradicts $A \in T1$. If the definition is implicit, we have a path $idef$–$InsertL_A$–$duse(u_1)$–$InsertL_A(p_j, p_{j+1})$–$duse(q_\lambda)$, which contradicts $A \in T2$;

4) $InsertL_{\Pi}(q_1, q_2) \ \wedge \ InsertS_{\Pi}(q_{\lambda-1}, q_\lambda)$ 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: $\exists u \in Paths, (u_1{=}q_2) \wedge duse(u_\lambda) \wedge \neg(idef \vee ddef)^{\exists}(\ [u_1, u_\lambda[\ )$. 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 $q_{\lambda-1}$: $\exists w \in Paths, ddef(w_1) \wedge (w_\lambda = q_{\lambda-1}) \wedge \neg(idef \vee iuse)^{\exists}(\ ]w_1, u_\lambda]\ )$. 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 $A \in T2$, we have $\neg InsertL_A^{\exists}(\ ]p_k, p_t]\ )$, so the path $[p_k, p_t]$ is a contiguous liveness interval for the placement function A.

14. Experiments

Table 1. Comparison of naive and optimal placement functions

	JRE 1.4.2		Java2Demo		FOP 0.20.5	
	naive	optimal	naive	optimal	naive	optimal
Number of stores	395087	63927	16065	3513	212210	36090
Number of loads	319454	273274	9883	8212	106109	89645
Size of generated code, KB	14973.04	14951.70	470.00	468.46	5013.32	5010.62

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.

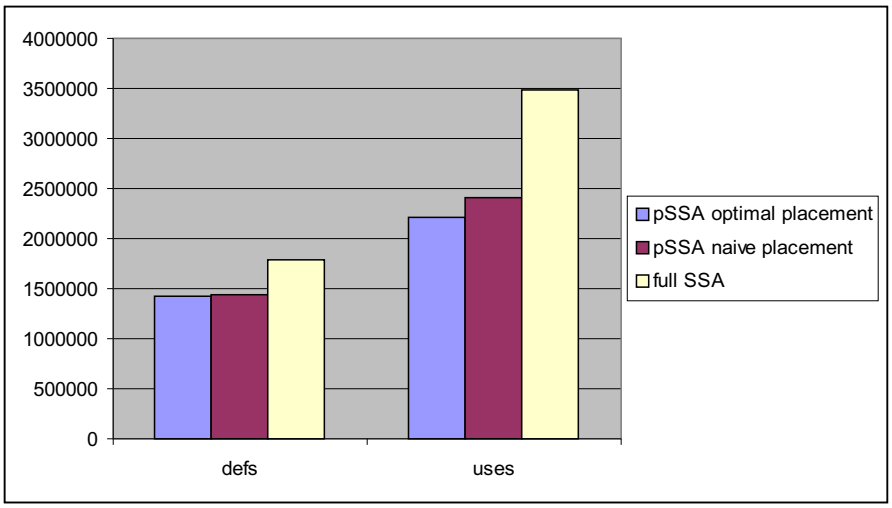


Figure 6. Size of the def-use information

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: $\langle a, b, c\rangle = foo(a, b, c)$, 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 $IsAlias$: $v_1 = \ldots; \quad *p = 1; \quad v_2 = IsAlias(p, \&v_1)$, 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 ϕ-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 (μ- and χ-functions) into the program IR that are similar to our loads and stores. Consider a small example: let a call to $foo()$ reads and wrties a variable v. Here is the HSSA form for the example: $\mu(v_i);\ foo();\ v_j = \chi(v_i)$. However, the μ- and χ-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 μ- and χ-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

[1] Alpern B. et al. The Jalapeño virtual machine // IBM Systems Journal. — 2000. — Vol. 39, N 1. — P. 211–238.

[2] Bodik R., Gupta R., Soffa M. L. Load-reuse analysis: design and evaluation // Proc. of the ACM SIGPLAN 1999 conf. on Programming language design and implementation (PLDI '99). — New York: ACM Press, 1999. — P. 64–76.

[3] Bilardi G., Pingali K. The static single assignment form and its computation. — 1999. — 37 p. — (Tech. Rep. / Cornell University).

[4] Chow F. C. et al. A new algorithm for partial redundancy elimination based on SSA form // Proc. of the SIGPLAN Conf. on Programming Language Design and Implementation. — 1997. — P. 273–286.

[5] Chow F. C. et al. Effective representation of aliases and indirect memory operations in SSA form // Proc. of the Internat. Conf. of Compier Construction. — 1996. — P. 253–267.

[6] Cytron R. et al. Efficiently computing static single assignment form and the control dependence graph // ACM Trans. on Programming Languages and Systems. — 1991. — Vol. 13, N 4. — P. 451–490.

[7] Cytron R., Gershbein R. Efficient accomodation of may-alias information in SSA form // Proc. of the SIGPLAN'93 Conf. on Programming Language Design and Implementation (PLDI-93), June 1993.

[8] Cooper K. D., Li Xu An efficient static analysis algorithm to detect redundant memory operations // Proc. of the workshop on Memory System Performance (MSP '02). — New York: ACM Press, 2002. — P. 97–107.

[9] Fitzgerald R. et al. Marmot: an optimizing compiler for Java. — Microsoft Research, 1999. — 29 p. — (Tech. Rep. / Microsoft Research; MSR-TR-99-33).

[10] Gurchenkov D. Modification of the SSAPRE algorithm for semantic code motion // Proc. of the 8th Korea—Russia Internat. Sympos. of Sci. and Technology (KORUS 2004). — 2004. — Vol. I. — P. 51–55.

[11] Grabar' A. V., Zemtsov P. A., Nalimov E. V. Optimizing code generator in SOKRAT project // Software Intellectualization and Quality. — Novosibirsk, 1994. — P. 49–67 (In Russian).

[12] Grabar' A. V., Zemtsov P. A., Nalimov E. V. On operator side effect analysis in SOKRAT optimizing code generator // Software Intellectualization and Quality. — Novosibirsk, 1994. — P. 82–89. (In Russian).

[13] Kildall G. A. A unified approach to global program optimization // Conf. Record of the ACM SIGACT/SIGPLAN Sympos. of Principles of Programming Languages, Boston, MA, October 1973. — P. 194–206.

[14] Knoop J., Rüthing O., Steffen B. Optimal code motion: Theory and practice // ACM Trans. on Programming Languages and Systems. — 1994. — Vol. 16, N 4. — P. 1117–1155.

[15] Knobe K., Sarkar V. Array ssa form and its use in parallelization // Proc. of the 25th ACM SIGPLAN-SIGACT Sympos. on Principles of Programming Languages (POPL '98). — New York: ACM Press, 1998. — P. 107–120.

[16] Lo R. et al. Register promotion by sparse partial redundancy elimination of loads and stores // SIGPLAN Not. — 1998. — Vol. 33, N 5. — P. 26–37.

[17] Lapkowski Ch., Hendren L. J. Extended SSA Numbering: Introducing SSA properties to languages with multi-level pointers // Proc. of CASCON'96, Toronto, Canada, November 1996. — P. 128–143.

[18] Shin-Ming Liu, Lo R., Chow F. Loop induction variable canonicalization in parallelizing compilers // Proc. of the Fourth Internat. Conf. on Parallel Architectures and Compilation Techniques, October 1996. — P. 228–237.

[19] Mikheev V. Design of multilingual retargetable compilers: Experience of the XDS framework evolution // Proc. of the Joint Modular Language Conf.. — Lect. Notes Comput. Sci. — 2000. — N 1897. — P. 238–249.

[20] Mikheev V. et al. Overview of Excelsior JET, a high performance alternative to Java virtual machines // Proc. of the 3rd Internat. Workshop on Software and Performance (WOSP-02). — New York: ACM Press, 2002. — P. 104–113.

[21] Morel E., Renvoise C. Global optimization by suppression of partial redundancies // Comm. ACM. — 1979. — Vol. 22, N 2. — P. 96–103.

[22] Muchnik S. S. Advanced Compiler Design and Implementation. — San Francisco: Morgan Kaufmann Publishers, 1997.

[23] Novillo D. Tree SSA — a new optimization infrastructure for GCC // Proc. of the GCC Developers Summit, Ottawa, Ontario, Canada, May 25–27 2003. — P. 181–193.

[24] Pottosin I. V. SOKRAT system: A programming environment for embedded computers. — Novosibirsk, 1992. — (Prep. / SB RAS Institute of Informatics Systems; N 11) (In Russian).

[25] Suganuma T. et al. Overview of the IBM Java Just-In-Time compiler // IBM Systems J. — 2000. — Vol. 39, N 1. — P. 175–193.

[26] Scales D. J. et al. The Swift Java compiler: Design and implementation. — Palp Alto, CA, 2000. — (Tech. Rep. / Compaq Western Research Laboratory; WRL 2000/2).

[27] Tu P., Padua D. Gated ssa-based demand-driven symbolic analysis for parallelizing compilers // Proc. of the 9th Internat. Conf. on Supercomputing (ICS'95). — New York, NY: ACM Press, 1995. — P. 414–423.

[28] VanDrunden Th., Hosking A. Anticipation-based partial redundancy elimination for static single assignment form // Software — Practice and Experience. — 2004. — Vol. 34, N 15. — P. 1413–1439.

[29] Wegman M., Zadeck K. Constant propagation with conditional branches // ACM Trans. on Programming Languages and Systems. — 1991. — Vol. 13, N 2. — P. 181–210.