Parameterized Construction of Program Representations for Sparse Dataflow Analyses

André Tavares[1], Benoit Boissinot[2], Fernando Pereira[1], and Fabrice Rastello[3]

[1] UFMG
[2] Ens Lyon
[3] Inria

Abstract. Data-flow analyses usually associate information with control flow regions. Informally, if these regions are too small, like a point between two consecutive statements, we call the analysis dense. On the other hand, if these regions include many such points, then we call it sparse. This paper presents a systematic method to build program representations that support sparse analyses. To pave the way to this framework we clarify the bibliography about well-known intermediate program representations. We show that our approach, up to parameter choice, subsumes many of these representations, such as the SSA, SSI and e-SSA forms. In particular, our algorithms are faster, simpler and more frugal than the previous techniques used to construct SSI - Static Single Information - form programs. We produce intermediate representations isomorphic to Choi *et al.*'s Sparse Evaluation Graphs (SEG) for the family of data-flow problems that can be partitioned per variables. However, contrary to SEGs, we can handle - sparsely - problems that are not in this family. We have tested our ideas in the LLVM compiler, comparing different program representations in terms of size and construction time.

1 Introduction

Many data-flow analyses bind information to pairs formed by a variable and a program point [4, 21, 24, 26, 30, 32, 36–39]. As an example, for each program point p, and each integer variable v live at p, Stephenson *et al.*'s [36] bit-width analysis finds the size, in bits, of v at p. Although well studied in the literature, this approach might produce redundant information. For instance, a given variable v may be mapped to the same bit-width along many consecutive program points. Therefore, a natural way to reduce redundancies is to make these analyses *sparser*, increasing the granularity of the program regions that they manipulate.

There exists different attempts to implement data-flow analyses sparsely. The Static Single Assignment (SSA) form [13], for instance, allows us to implement several analyses and optimizations, such as reaching definitions and constant propagation, sparsely. Since its conception, the SSA format has been generalized into many different program representations, such as the *Extended-SSA* form [4], the *Static Single Information* (SSI) form [1], and the *Static Single Use* (SSU) form [18, 23, 30]. Each of these representations extends the reach of the SSA form to sparser data-flow analyses; however, there is not a format that subsumes all

A. Cohen (Ed.): CC 2014, LNCS 8409, pp. 18–39, 2014.
© Springer-Verlag Berlin Heidelberg 2014

the others. In other words, each of these three program representations fit specific types of data-flow problems. Another attempt to model data-flow analyses sparsely is due to Choi *et al.*'s *Sparse Evaluation Graph* (SEG) [9]. This data-structure supports several different analyses sparsely, as long as the abstract state of a variable does not interfere with the abstract state of other variables in the same program. This family of analyses is known as *Partitioned Variable Problems* in the literature [41].

In this paper, we propose a framework that includes all these previous approaches. Given a data-flow problem defined by (i) a set of control flow nodes, that produce information, and (ii) a direction in which information flows: forward, backward or both ways, we build a program representation that allows to solve the problem sparsely using def-use chains. The program representations that we generate ensure a key *single information property*: the data-flow facts associated with a variable are invariant along the entire live range of this variable.

We have implemented our framework in the LLVM compiler [22], and have used it to provide intermediate representations to well-known compiler optimizations: Wegman *et al.*'s [39] conditional constant propagation, and Bodik *et al.*'s [4] algorithm for array bounds check elimination. We compare these representations with the SSI form as defined by Singer. The intermediate program representations that we build increase the size of the original program by less than 5% - one order of magnitude less than Singer's SSI form. Furthermore, the time to build these program representations is less than 2% of the time taken by the standard suite of optimizations used in the LLVM compiler. Finally, our intermediate representations have already been used in the implementation of different static analyses, already publicly available [7, 15, 32, 34].

2 Static Single Information

Our objective is to generate program representations that bestow the *Static Single Information property* (Definition 6) onto a given data-flow problem. In order to introduce this notion, we will need a number of concepts, which we define in this chapter. We start with the concept of a *Data-Flow System*, which Definition 1 recalls from the literature. We consider a *program point* a point between two consecutive instructions. If p is a program point, then $preds(p)$ (resp. $succs(p)$) is the set of all the program points that are predecessors (resp. successors) of p. A *transfer function* determines how information flows among these program points. Information are elements of a *lattice*. We find a solution to a data-flow problem by continuously solving the set of transfer functions associated with each program region until a fix point is reached. Some program points are *meet nodes*, because they combine information coming from two or more regions. The result of combining different elements of a lattice is given by a *meet* operator, which we denote by $\wedge$.

Definition 1 (Data-Flow System). *A data-flow system E_{dense} is an equation system that associates, with each program point p, an element of a lattice $\mathcal{L}$, given by the equation $x^p = \bigwedge_{s \in preds(p)} F^{s,p}(x^s)$, where: x^p denotes the abstract state*

associated with program point p; preds(p) is the set of control flow predecessors of p; $F^{s,p}$ *is the transfer function from program point s to program point p. The analysis can alternatively be written as a constraint system that binds to each program point p and each* $s \in preds(p)$ *the equation* $x^p = x^p \wedge F^{s,p}(x^s)$ *or, equivalently, the inequation* $x^p \sqsubseteq F^{s,p}(x^s)$*.*

The program representations that we generate lets us solve a class of data-flow problems that we call *Partitioned Lattice per Variable* (PLV), and that we introduce in Definition 2. Constant propagation is an example of a PLV problem. If we denote by $\mathcal{C}$ the lattice of constants, the overall lattice can be written as $\mathcal{L} = \mathcal{C}^n$, where n is the number of variables. In other words, this data-flow problem ranges on a product lattice that contains a term for each variable in the target program.

Definition 2 (Partitioned Lattice per Variable Problem (PLV)). *Let* $\mathcal{V} = \{v_1, \ldots, v_n\}$ *be the set of program variables. The Maximum Fixed Point problem on a data-flow system is a* Partitioned Lattice per Variable Problem *if, and only if,* $\mathcal{L}$ *can be decomposed into the product of* $\mathcal{L}_{v_1} \times \cdots \times \mathcal{L}_{v_n}$ *where each* $\mathcal{L}_{v_i}$ *is the lattice associated with program variable* v_i*. In other words* x^s *can be writen as* $([v_1]^s, \ldots, [v_n]^s)$ *where* $[v]^s$ *denotes the abstract state associated with variable v and program point s.* $F^{s,p}$ *can thus be decomposed into the product of* $F^{s,p}_{v_1} \times \cdots \times F^{s,p}_{v_n}$ *and the constraint system decomposed into the inequalities* $[v_i]^p \sqsubseteq F^{s,p}_{v_i}([v_1]^s, \ldots, [v_n]^s)$*.*

The transfer functions that we describe in Definition 3 have no influence on the solution of a data-flow system. The goal of a sparse data-flow analysis is to shortcut these functions. We accomplish this task by grouping contiguous program points bound to these functions into larger regions.

Definition 3 (Trivial/Constant/Undefined Transfer functions). *Let* $\mathcal{L}_{v_1} \times \mathcal{L}_{v_2} \times \cdots \times \mathcal{L}_{v_n}$ *be the decomposition per variable of lattice* $\mathcal{L}$*, where* $\mathcal{L}_{v_i}$ *is the lattice associated with variable* v_i*. Let* F_{v_i} *be a transfer function from* $\mathcal{L}$ *to* $\mathcal{L}_{v_i}$*.*

- F_{v_i} *is* trivial *if* $\forall x = ([v_1], \ldots, [v_n]) \in \mathcal{L}$, $F_{v_i}(x) = [v_i]$
- F_{v_i} *is* constant with value $C \in \mathcal{L}_{v_i}$ *if* $\forall x \in \mathcal{L}$, $F_{v_i}(x) = C$
- F_{v_i} *is* undefined *if* F_{v_i} *is constant with value* $\top$*, e.g.,* $F_{v_i}(x) = \top$*, where* $\top \wedge y = y \wedge \top = y$*.*

A sparse data-flow analysis propagates information from the control flow node where this information is created directly to the control flow node where this information is needed. Therefore, the notion of *dependence*, which we state in Definition 4, plays a fundamental role in our framework. Intuitively, we say that a variable v depends on a variable v_j if the information associated with v might change in case the information associated with v_j does.

Definition 4 (Dependence). *We say that* F_v depends on variable v_j *if:*

$$\exists x = ([v_1], \ldots, [v_n]) \neq ([v_1]', \ldots, [v_n]') = x' \text{ in } \mathcal{L}$$
$$\text{such that } [\forall k \neq j,\ [v_k] = [v_k]' \wedge F_v(x) \neq F_v(x')]$$

In a *backward* data-flow analysis, the information that comes from the predecessors of a node n is combined to produce the information that reaches the successors of n. A *forward* analysis propagates information in the opposite direction. We call meet nodes those places where information coming from multiple sources are combined. Definition 5 states this concept more formally.

Definition 5 (Meet Nodes). *Consider a forward (resp. backward) monotone PLV problem, where* (Y_v^p) *is the maximum fixed point solution of variable* v *at program point* p*. We say that a program point* p *is a meet node for variable* v *if, and only if,* p *has* $n \geq 2$ *predecessors (resp. successors),* $s_1, \ldots, s_n$*, and there exists* $s_i \neq s_j$*, such that* $Y_v^{s_i} \neq Y_v^{s_j}$*.*

Our goal is to build program representations in which the information associated with a variable is invariant along the entire live range of this variable. A variable v is *alive* at a program point p if there is a path from p to an instruction that uses v, and v is not re-defined along the way. The live range of v, which we denote by *live(v)*, is the collection of program points where v is alive.

Definition 6 (Static Single Information property). *Consider a forward (resp. backward) monotone PLV problem* E_{dense} *stated as in Definition 1. A program representation fulfills the Static Single Information property if, and only if, it meets the following properties for each variable* v*:*

- **[SPLIT-DEF]:** *for each two consecutive program points* s *and* p *such that* $p \in live(v)$*, and* $F_v^{s,p}$ *is non-trivial nor undefined, there should be an instruction between* s *and* p *that contains a definition (resp. last use) of* v*;*
- **[SPLIT-MEET]:** *each meet node* p *with* n *predecessors* $\{s_1, \ldots, s_n\}$ *(resp. successors) should have a definition (resp. use) of* v *at* p*, and* n *uses (resp. definitions) of* v*, one at each* s_i*. We shall implement these defs/uses with* ϕ/σ*-functions, as we explain in Section 2.1.*
- **[INFO]:** *each program point* $p \notin live(v)$ *should be bound to undefined transfer functions, e.g.,* $F_v^{s,p} = \lambda x.\top$ *for each* $s \in preds(p)$ *(resp.* $s \in succs(p)$*).*
- **[LINK]:** *for each two consecutive program points* s *and* p *for which* $F_v^{s,p}$ *depends on some* $[u]^s$*, there should be an instruction between* s *and* p *that contains a (potentially pseudo) use (resp. def) of* u*.*
- **[VERSION]:** *for each variable* v*, live*(v) *is a connected component of the CFG.*

2.1 Special Instructions Used to Split Live Ranges

We group control flow nodes in three kinds: interior nodes, forks and joins. At each place we use a different notation to denote live range splitting.

Interior nodes are control flow nodes that have a unique predecessor and a unique successor. At these control flow nodes we perform live range splitting via copies. If the control flow node already contains another instruction, then this copy *must* be done *in parallel* with the existing instruction. The notation,

$$inst \;\parallel\; v_1 = v_1' \;\parallel\; \ldots \;\parallel\; v_m = v_m'$$

denotes m copies $v_i = v'_i$ performed in parallel with instruction *inst*. This means that all the uses of *inst* plus all v'_i are read simultaneously, then *inst* is computed, then all definitions of *inst* plus all v_i are written simultaneously.

In forward analyses, the information produced at different definitions of a variable may reach the same meet node. To avoid that these definitions reach the same use of v, we merge them at the earliest control flow node where they meet; hence, ensuring [SPLIT-MEET]. We do this merging via special instructions called ϕ-functions, which were introduced by Cytron *et al.* to build SSA-form programs [13]. The assignment

$$v_1 = \phi(l^1 : v_1^1, \ldots, l^q : v_1^q) \ \| \ \ldots \ \| \ v_m = \phi(l^1 : v_m^1, \ldots, l^q : v_m^q)$$

contains m ϕ-functions to be performed in parallel. The ϕ symbol works as a multiplexer. It will assign to each v_i the value in v_i^j, where j is determined by l^j, the basic block last visited before reaching the ϕ-function. The above statement encapsulates m parallel copies: all the variables $v_1^j, \ldots, v_m^j$ are simultaneously copied into the variables $v_1, \ldots, v_m$. Note that our notion of control flow nodes differs from the usual notion of nodes of the CFG. A join node actually corresponds to the entry point of a CFG node: to this end we denote as $\text{In}(l)$ the point right before l. As an example in Figure 1(d), l_7 is considered to be an interior node, and the ϕ-function defining v_6 has been inserted at the join node $\text{In}(l_7)$.

In backward analyses the information that emerges from different uses of a variable may reach the same meet node. To ensure Property [SPLIT-MEET], the use that reaches the definition of a variable must be unique, in the same way that in a SSA-form program the definition that reaches a use is unique. We ensure this property via special instructions that Ananian has called σ-functions [1]. The σ-functions are the simetric of ϕ-functions, performing a parallel assignment depending on the execution path taken. The assignment

$$(l^1 : v_1^1, \ldots, l^q : v_1^q) = \sigma(v_1) \ \| \ \ldots \ \| \ (l^1 : v_m^1, \ldots, l^q : v_m^q) = \sigma(v_m)$$

represents m σ-functions that assign to each variable v_i^j the value in v_i if control flows into block l^j. These assignments happen in parallel, i.e., the m σ-functions encapsulate m parallel copies. Also, notice that variables live in different branch targets are given different names by the σ-function that ends that basic block. Similarly to join nodes, a fork node is the exit point of a CFG node: $\text{Out}(l)$ denotes the point right after CFG node l. As an example in Figure 1(d), l_2 is considered to be an interior node, and the σ-function using v_1 has been inserted at the fork node $\text{Out}(l_2)$.

2.2 Examples of PLV Problems

Many data-flow analyses can be classified as PLV problems. In this section we present some meaningful examples. Along each example we show the program representation that lets us solve it sparsely.

Class Inference: Some dynamically typed languages, such as Python, Java-Scrip, Ruby or Lua, represent objects as hash tables containing methods and

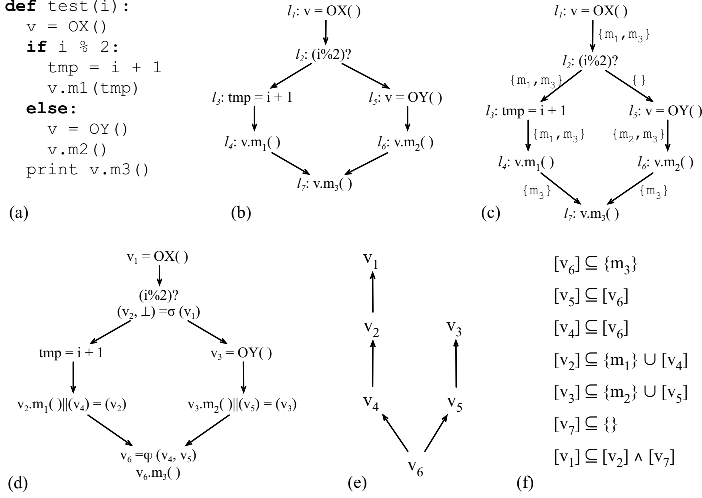

Fig. 1. Class inference as an example of backward data-flow analysis that takes information from the uses of variables

fields. In this world, it is possible to speedup execution by replacing these hash tables with actual object oriented virtual tables. A class inference engine tries to assign a virtual table to a variable v based on the ways that v is used. The Python program in Figure 1(a) illustrates this optimization. Our objective is to infer the correct suite of methods for each object bound to variable v. Figure 1(b) shows the control flow graph of the program, and Figure 1(c) shows the results of a dense implementation of this analysis. In a dense analysis, each program instruction is associated with a transfer function; however, some of these functions, such as that in label l_3, are trivial. We produce, for this example, the representation given in Figure 1(d). Because type inference is a backward analysis that extracts information from use sites, we split live ranges at these control flow nodes, and rely on σ-functions to merge them back. The use-def chains that we derive from the program representation, seen in Figure 1(e), lead naturally to a constraint system, which we show in Figure 1(f). A solution to this constraint system gives us a solution to our data-flow problem.

Constant Propagation: Figure 2 illustrates constant propagation, e.g., which variables in the program of Figure 2(a) can be replaced by constants? The CFG of this program is given in Figure 2(b). Constant propagation has a very simple lattice $\mathcal{L}$, which we show in Figure 2(c). Constant propagation is a PLV problem, as we have discussed before. In constant propagation, information is produced at the program points where variables are defined. Thus, in order to meet Definition 6, we must guarantee that each program point is reachable by a single definition of a variable. Figure 2(d) shows the intermediate representation that we create for the program in Figure 2(b). In this case, our intermediate representation is equivalent to the SSA form. The def-use chains implicit in our program

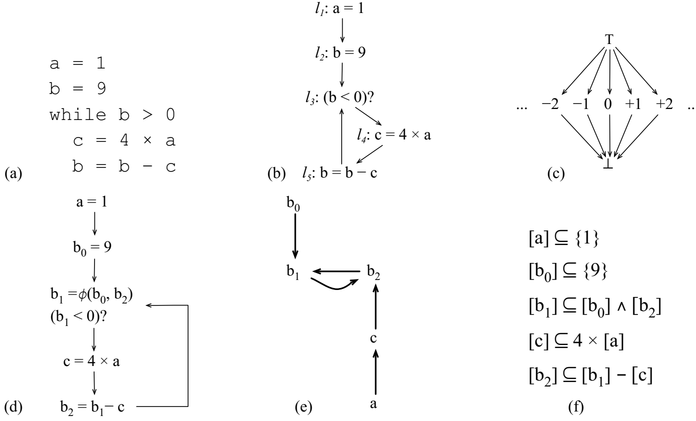

Fig. 2. Constant propagation as an example of forward data-flow analysis that takes information from the definitions of variables

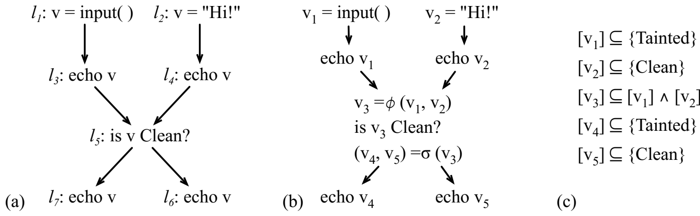

Fig. 3. Taint analysis is a forward data-flow analysis that takes information from the definitions of variables and conditional tests on these variables

representation lead to the constraint system shown in Figure 2(f). We can use the def-use chains seen in Figure 2(e) to guide a worklist-based constraint solver, as Nielson *et al.* [27, Ch.6] describe.

Taint Analysis: The objective of taint analysis [32, 33] is to find program vulnerabilities. In this case, a harmful attack is possible when input data reaches sensitive program sites without going through special functions called sanitizers. Figure 3 illustrates this type of analysis. We have used ϕ and σ-functions to split the live ranges of the variables in Figure 3(a) producing the program in Figure 3(b). Let us assume that *echo* is a sensitive function, because it is used to generate web pages. For instance, if the data passed to *echo* is a JavaScript program, then we could have an instance of cross-site scripting attack. Thus, the statement *echo* v_1 may be a source of vulnerabilities, as it outputs data that comes directly from the program input. On the other hand, we know that *echo* v_2 is always safe, for variable v_2 is initialized with a constant value. The

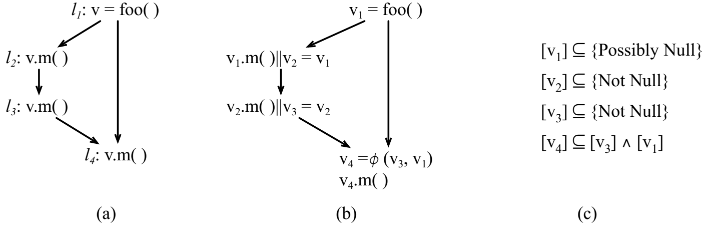

Fig. 4. Null pointer analysis as an example of forward data-flow analysis that takes information from the definitions and uses of variables

call *echo* v_5 is always safe, because variable v_5 has been sanitized; however, the call *echo* v_4 might be tainted, as variable v_4 results from a failed attempt to sanitize v. The def-use chains that we derive from the program representation lead naturally to a constraint system, which we show in Figure 3(c). The intermediate representation that we create in this case is equivalent to the *Extended Single Static Assignment* (e-SSA) form [4]. It also suits the ABCD algorithm for array bounds-checking elimination [4], Su and Wagner's range analysis [37] and Gawlitza *et al.*'s range analysis [17].

Null Pointer Analysis: The objective of null pointer analysis is to determine which references may hold null values. Nanda and Sinha have used a variant of this analysis to find which method dereferences may throw exceptions, and which may not [26]. This analysis allows compilers to remove redundant null-exception tests and helps developers to find null pointer dereferences. Figure 4 illustrates this analysis. Because information is produced at use sites, we split live ranges after each variable is used, as we show in Figure 4(b). For instance, we know that the call $v_2.m()$ cannot result in a null pointer dereference exception, otherwise an exception would have been thrown during the invocation $v_1.m()$. On the other hand, in Figure 4(c) we notice that the state of v_4 is the meet of the state of v_3, definitely not-null, and the state of v_1, possibly null, and we must conservatively assume that v_4 may be null.

3 Building the Intermediate Program Representation

A *live range splitting strategy* $\mathcal{P}_v = I_\uparrow \cup I_\downarrow$ over a variable v consists of two sets of control flow nodes (see Section 2.1 for a definition of control flow nodes). We let $I_\downarrow$ denote a set of control flow nodes that produce information for a forward analysis. Similarly, we let $I_\uparrow$ denote a set of control flow nodes that are interesting for a backward analysis. The live-range of v must be split at least at every control flow node in $\mathcal{P}_v$. Going back to the examples from Section 2.2, we have the live range splitting strategies enumerated below. Further examples are given in Figure 5.

Client	Splitting strategy $\mathcal{P}$
Alias analysis, reaching definitions cond. constant propagation [39]	$Defs_{\downarrow}$
Partial Redundancy Elimination [1, 35]	$Defs_{\downarrow} \bigcup LastUses_{\uparrow}$
ABCD [4], taint analysis [32], range analysis [17, 37]	$Defs_{\downarrow} \bigcup \mathrm{Out}(Conds)_{\downarrow}$
Stephenson's bitwidth analysis [36]	$Defs_{\downarrow} \bigcup \mathrm{Out}(Conds)_{\downarrow} \bigcup Uses_{\uparrow}$
Mahlke's bitwidth analysis [24]	$Defs_{\downarrow} \bigcup Uses_{\uparrow}$
An's type inference [19], class inference [8]	$Uses_{\uparrow}$
Hochstadt's type inference [38]	$Uses_{\uparrow} \bigcup \mathrm{Out}(Conds)_{\uparrow}$
Null-pointer analysis [26]	$Defs_{\downarrow} \bigcup Uses_{\downarrow}$

Fig. 5. Live range splitting strategies for different data-flow analyses. We use *Defs* (*Uses*) to denote the set of instructions that define (use) the variable; *Conds* to denote the set of instructions that apply a conditional test on a variable; Out(*Conds*) the exits of the corresponding basic blocks; *LastUses* to denote the set of instructions where a variable is used, and after which it is no longer live.

- **Class inference** is a backward analysis that takes information from the uses of variables. Thus, for each variable, the live-range splitting strategy contains the set of control flow nodes where that variable is used. For instance, in Figure 1(b), we have that $\mathcal{P}_v = \{l_4, l_6, l_7\}_{\uparrow}$.
- **Constant propagation** is a forward analysis that takes information from definition sites. Thus, for each variable v, the live-range splitting strategy is characterized by the set of points where v is defined. For instance, in Figure 2(b), we have that $\mathcal{P}_b = \{l_2, l_5\}_{\downarrow}$.
- **Taint analysis** is a forward analysis that takes information from control flow nodes where variables are defined, and conditional tests that use these variables. For instance, in Figure 3(a), we have that $\mathcal{P}_v = \{l_1, l_2, \mathrm{Out}(l_5)\}_{\downarrow}$.
- Nanda *et al.*'s **null pointer analysis** [26] is a forward flow problem that takes information from definitions and uses. For instance, in Figure 4(a), we have that $\mathcal{P}_v = \{l_1, l_2, l_3, l_4\}_{\downarrow}$.

```
1 function SSIfy(var v, Splitting_Strategy P_v)
2     split(v, P_v)
3     rename(v)
4     clean(v)
```

Fig. 6. Split the live ranges of v to convert it to SSI form

The algorithm SSIfy in Figure 6 implements a live range splitting strategy in three steps: split, rename and clean, which we describe in the rest of this section.

Splitting Live Ranges through the Creation of New Definitions of Variables: To implement $\mathcal{P}_v$, we must split the live ranges of v at each control flow node listed by $\mathcal{P}_v$. However, these control flow nodes are not the only ones where splitting might be necessary. As we have pointed out in Section 2.1, we might have, for the same original variable, many different sources of information reaching a common meet point. For instance, in Figure 3(b), there exist two definitions of variable v: v_1 and v_2, that reach the use of v at l_5. Information that flows forward from l_3 and l_4 collide at l_5, the meet point of the if-then-else. Hence the live-range of v has to be split at the entry of l_5, e.g., at $\text{In}(l_5)$, leading to a new definition v_3. In general, the set of control flow nodes where information collide can be easily characterized by join sets [13]. The join set of a group of nodes P contains the CFG nodes that can be reached by two or more nodes of P through disjoint paths. Join sets can be over-approximated by the notion of iterated dominance frontier [40], a core concept in SSA construction algorithms, which, for the sake of completeness, we recall below:

- **Dominance**: a CFG node n dominates a node n' if every program path from the entry node of the CFG to n' goes across n. If $n \neq n'$, then we say that n *strictly* dominates n'.
- **Dominance Frontier** (DF): a node n' is in the dominance frontier of a node n if n dominates a predecessor of n', but does not strictly dominate n'.
- **Iterated dominance frontier** (DF^+): the iterated dominance frontier of a node n is the limit of the sequence:

$$\begin{aligned} DF_1 &= DF(n) \\ DF_{i+1} &= DF_i \cup \{DF(z) \mid z \in DF_i\} \end{aligned}$$

Similarly, split sets created by the backward propagation of information can be over-approximated by the notion of *iterated post-dominance frontier* (pDF^+), which is the DF^+ [2] of the CFG where orientation of edges have been reverted. If $e = (u, v)$ is an edge in the control flow graph, then we define the dominance frontier of e, i.e., $DF(e)$, as the dominance frontier of a fictitious node n placed at the middle of e. In other words, $DF(e)$ is $DF(n)$, assuming that (u, n) and (n, v) would exist. Given this notion, we also define $DF^+(e)$, $pDF(e)$ and $pDF^+(e)$.

Figure 7 shows the algorithm that creates new definitions of variables. This algorithm has three phases. First, in lines 3-9 we create new definitions to split the live ranges of variables due to *backward* collisions of information. These new definitions are created at the iterated post-dominance frontier of control flow nodes that originate information. Notice that if the control flow node is a join (entry of a CFG node), information actually originate from each incoming edges (line 6). In lines 10-16 we perform the inverse operation: we create new definitions of variables due to the forward collision of information. Finally, in lines 17-23 we actually insert the new definitions of v. These new definitions might be created by σ functions (due exclusively to the splitting in lines 3-9); by ϕ-functions

```
1  function split(var v, Splitting_Strategy P_v = I_↓ ∪ I_↑)
2       "compute the set of split points"
3      S_↑ = ∅
4      foreach i ∈ I_↑:
5           if i.is_join:
6                foreach e ∈ incoming_edges(i):
7                     S_↑ = S_↑ ⋃ Out(pDF^+(e))
8           else:
9                S_↑ = S_↑ ⋃ Out(pDF^+(i))
10     S_↓ = ∅
11     foreach i ∈ S_↑ ⋃ Defs(v) ⋃ I_↓:
12          if i.is_fork:
13               foreach e ∈ outgoing_edges(i)
14                    S_↓ = S_↓ ⋃ In(DF^+(e))
15          else:
16               S_↓ = S_↓ ⋃ In(DF^+(i))
17     S = P_v ⋃ S_↑ ⋃ S_↓
18      "Split live range of v by inserting φ, σ, and copies"
19     foreach i ∈ S:
20          if i does not already contain any definition of v:
21               if i.is_join: insert "v = φ(v, ..., v)" at i
22               elseif i.is_fork: insert "(v, ..., v) = σ(v)" at i
23               else: insert a copy "v = v" at i
```

Fig. 7. Live range splitting. We use $\text{In}(l)$ to denote a control flow node at the entry of l, and $\text{Out}(l)$ to denote a control flow node at the exit of l. We let $\text{In}(S) = \{l \in S \mid l \in S\}$. $\text{Out}(S)$ is defined in a similar way.

(due exclusively to the splitting in lines 10-16); or by parallel copies. Contrary to Singer's algorithm, originally designed to produce SSI form programs, we do not iterate between the insertion of ϕ and σ functions.

The Algorithm split preserves the SSA property, even for data-flow analyses that do not require it. As we see in line 11, the loop that splits meet nodes forwardly include, by default, all the definition sites of a variable. We chose to implement it in this way for practical reasons: the SSA property gives us access to a fast liveness check [5], which is useful in actual compiler implementations. This algorithm inserts ϕ and σ functions conservatively. Consequently, we may have these special instructions at control flow nodes that are not true meet nodes. In other words, we may have a ϕ-function $v = \phi(v_1, v_2)$, in which the abstract states of v_1 and v_2 are the same in a final solution of the data-flow problem.

Variable Renaming: The algorithm in Figure 8 builds def-use and use-def chains for a program after live range splitting. This algorithm is similar to the standard algorithm used to rename variables during the SSA construction [2, Algorithm 19.7]. To rename a variable v we traverse the program's dominance tree, from top to bottom, stacking each new definition of v that we find. The definition currently on the top of the stack is used to replace all the uses of v that

```
1  function rename(var v)
2       "Compute use-def & def-use chains"
3       "We consider here that stack.peek() = undef if stack.isempty(),
4         and that Def(undef) = entry"
5      stack = ∅
6      foreach CFG node n in dominance order:
7           foreach m that is a predecessor of n:
8                if exists d_m of the form "l^m : v = ..." in a σ-function in Out(m):
9                     stack.set_def(d_m)
10               if exits u_m of the form "··· = l^m : v" in a φ-function in In(n):
11                    stack.set_use(u_m)
12          if exists a φ-function d in In(n) that defines v:
13               stack.set_def(d)
14          foreach instruction u in n that uses v:
15               stack.set_use(u)
16          if exists an instruction d in n that defines v:
17               stack.set_def(d)
18          foreach σ-function u in Out(n) that uses v:
19               stack.set_use(u)

21 function stack.set_use(instruction inst):
22     while Def(stack.peek()) does not dominate inst: stack.pop()
23     v_i = stack.peek()
24     replace the uses of v by v_i in inst
25     if v_i ≠ undef: set Uses(v_i) = Uses(v_i) ⋃ inst

27 function stack.set_def(instruction inst):
28     let v_i be a fresh version of v
29     replace the defs of v by v_i in inst
30     set Def(v_i) = inst
31     stack.push(v_i)
```

Fig. 8. Versioning

we find during the traversal. If the stack is empty, this means that the variable is not defined at that point. The renaming process replaces the uses of undefined variables by undef (line 3). We have two methods, *stack*.set_use and *stack*.set_def to build the chain relations between the variables. Notice that sometimes we must rename a single use inside a ϕ-function, as in lines 10-11 of the algorithm. For simplicity we consider this single use as a simple assignment when calling *stack*.set_use, as one can see in line 11. Similarly, if we must rename a single definition inside a σ-function, then we treat it as a simple assignment, like we do in lines 8-9 of the algorithm.

Dead and Undefined Code Elimination: The algorithm in Figure 9 eliminates ϕ-functions that define variables not actually used in the code, σ-functions that use variables not actually defined in the code, and parallel copies that ei-

```
1  function clean(var v)
2      let web = {v_i | v_i is a version of v}
3      let defined = ∅
4      let active = { inst | inst is actual instruction and web ∩ inst.defs ≠ ∅}
5      while exists inst in active s.t. web ∩ inst.defs \ defined ≠ ∅:
6          foreach v_i ∈ web ∩ inst.defs\defined:
7              active = active ∪ Uses(v_i)
8              defined = defined ∪ {v_i}
9      let used = ∅
10     let active = {inst |inst is actual instruction and web ∩ inst.uses ≠ ∅}
11     while exists inst ∈ active s.t. inst.uses\used ≠ ∅:
12         foreach v_i ∈ web ∩ inst.uses\used:
13             active = active ∪ Def(v_i)
14             used = used ∪ {v_i}
15     let live = defined ∩ used
16     foreach non actual inst ∈ Def(web):
17         foreach v_i operand of inst s.t. v_i ∉ live:
18                     replace v_i by undef
19         if inst.defs = {undef} or inst.uses = {undef}
20             eliminate inst from the program
```

Fig. 9. Dead and undefined code elimination. Original instructions not inserted by split are called *actual* instruction. We let *inst*.defs denote the set of variables defined by *inst*, and *inst*.uses denote the set of variables used by *inst*.

ther define or use variables that do not reach any actual instruction. "Actual" instructions are those instructions that already existed in the program before we transformed it with split. In line 3 we let "*web*" be the set of versions of v, so as to restrict the cleaning process to variable v, as we see in lines 4-6 and lines 10-12. The set "*active*" is initialized to actual instructions in line 4. Then, during the loop in lines 5-8 we add to active ϕ-functions, σ-functions, and copies that can reach actual definitions through use-def chains. The corresponding version of v is then marked as *defined* (line 8). The next loop, in lines 11-14 performs a similar process to add to the active set the instructions that can reach actual uses through def-use chains. The corresponding version of v is then marked as *used* (line 14). Each non live variable (see line 15), i.e. either undefined or dead (non used) is replaced by undef in all ϕ, σ, or copy functions where it appears. This is done in lines 15-18. Finally useless ϕ, σ, or copy functions are removed in lines 19-20. As a historical curiosity, Cytron *et al.*'s procedure to build SSA form produced what is called *the minimal representation* [13]. Some of the ϕ-functions in the minimal representation define variables that are never used. Briggs *et al.* [6] remove these variables; hence, producing what compiler writers normally call *pruned SSA-form.* We close this section stating that the SSIfy algorithm preserves the semantics of the modified program [1]:

[1] The theorems in this paper are proved in the companion report, available on-line.

Theorem 1 (Semantics). SSIfy *maintains the following property: if a value n written into variable v at control flow node i' is read at a control flow node i in the original program, then the same value assigned to a version of variable v at control flow node i' is read at a control flow node i after transformation.*

The Propagation Engine: Def-use chains can be used to solve, sparsely, a PLV problem about any program that fulfills the SSI property. However, in order to be able to rely on these def-use chains, we need to derive a sparse constraint system from the original - dense - system. This sparse system is constructed according to Definition 7. Theorem 2 states that such a system exists for any program, and can be obtained directly from the Algorithm SSIfy. The algorithm in Figure 10 provides worklist based solvers for backward and forward sparse data-flow systems built as in Definition 7.

Definition 7 (SSI constrained system). *Let E_{dense} be a constraint system extracted from a program that meets the SSI properties. Hence, for each pair (variable v, program point p) we have equations $[v]^p = [v]^p \wedge F_v^{s,p}([v_1]^s, \ldots, [v_n]^s)$. We define a system of sparse equations E_{sparse}^{ssi} as follows:*

- *Let $\{a, \ldots, b\}$ be the variables used (resp. defined) at control flow node i, where variable v is defined (resp. used). Let s and p be the program points around i. The LINK property ensures that $F_v^{s,p}$ depends only on some $[a]^s \ldots [b]^s$. Thus, there exists a function G_v^i defined as the projection of $F_v^{s,p}$ on $\mathcal{L}_a \times \cdots \times \mathcal{L}_b$, such that $G_v^i([a]^s, \ldots, [b]^s) = F_v^{s,p}([v_1]^s, \ldots, [v_n]^s)$.*
- *The sparse constrained system associates with each variable v, and each definition (resp. use) point s of v, the corresponding constraint $[v] \sqsubseteq G_v^s([a], \ldots, [b])$ where $a, \ldots, b$ are used (resp. defined) at i.*

Theorem 2 (Correctness of SSIfy). *The execution of SSIfy(v, $\mathcal{P}_v$), for every variable v in the target program, creates a new program representation such that:*

1. *there exists a system of equations E_{dense}^{ssi}, isomorphic to E_{dense} for which the new program representation fulfills the SSI property.*
2. *if E_{dense} is monotone then E_{dense}^{ssi} is also monotone.*

4 Our Approach vs Other Sparse Evaluation Frameworks

There have been previous efforts to provide theoretical and practical frameworks in which data-flow analyses could be performed sparsely. In order to clarify some details of our contribution, this section compares it with three previous approaches: Choi's Sparse Evaluation Graphs, Ananian's Static Single Information form and Oh's Sparse Abstract Interpretation Framework.

Sparse Evaluation Graphs: Choi's *Sparse Evaluation Graphs* [9] are one of the earliest data-structures designed to support sparse analyses. The nodes of

```
1  function forward_propagate(transfer_functions 𝒢)
2      worklist = ∅
3      foreach variable v: [v] = ⊤
4      foreach instruction i: worklist += i
5      while worklist ≠ ∅:
6          let i ∈ worklist
7          worklist −= i
8          foreach v ∈ i.defs:
9              [v]_new = [v] ∧ G_v^i([i.uses])
10             if [v] ≠ [v]_new:
11                 worklist += Uses(v)
12                 [v] = [v]_new
```

Fig. 10. Forward propagation engine under SSI. For backward propagation, we replace i.defs by i.uses and i.uses / Uses(v) by i.defs / Def(v).

this graph represent program regions where information produced by the data-flow analysis might change. Choi *et al.*'s ideas have been further expanded, for example, by Johnson *et al.*'s *Quick Propagation Graphs* [21], or Ramalingan's *Compact Evaluation Graphs* [31]. Nowadays we have efficient algorithms that build such data-structures [20, 29]. These graphs improve many data-flow analyses in terms of runtime and memory consumption. However, they are more limited than our approach, because they can only handle sparsely problems that Zadeck has classified as *Partitioned Variable*. In these problems, a program variable can be analyzed independently from the others. Reaching definitions and liveness analysis are examples of PVPs, as this kind of information can be computed for one program variable independently from the others. For these problems we can build intermediate program representations isomorphic to SEGs, as we state in Theorem 3. However, many data-flow problems, in particular the PLV analyses that we mentioned in Section 2.2, do not fit into this category. Nevertheless, we can handle them sparsely. The sparse evaluation graphs can still support PLV problems, but, in this case, a new SEG vertex would be created for every control flow node where new information is produced, and we would have a dense analysis.

Theorem 3 (Equivalence SSI/SEG). *Given a forward Sparse Evaluation Graph (SEG) that represents a variable v in a program representation Prog with CFG G, there exits a live range splitting strategy that once applied on v builds a program representation that is isomorphic to SEG.*

Static Single Information Form and Similar Program Representations: Scott Ananian has introduced in the late nineties the *Static Single Information* (SSI) form, a program representation that supports both forward and backward analyses [1]. This representation was later revisited by Jeremy Singer [35]. The σ-functions that we use in this paper is a notation borrowed from Ananian's work, and the algorithms that we discuss in Section 3 improve on Singer's ideas.

Contrary to Singer's algorithm we do not iterate between the insertion of phi and sigma functions. Consequently, as we will show in Section 5, we insert less phi and sigma functions. Nonetheless, as we show in Theorem 2, our method is enough to ensure the SSI properties for any combination of unidirectional problems. In addition to the SSI form, we can emulate several other different representations, by changing our parameterizations. Notice that for SSI we have $\{Defs_{\downarrow} \cup LastUses_{\uparrow}\}$. For Bodik's e-SSA [4] we have $Defs_{\downarrow} \bigcup \text{Out}(Conds)_{\downarrow}$. Finally, for SSU [18, 23, 30] we have $Uses_{\uparrow}$.

The SSI constrained system might have several inequations for the same left-hand-side, due to the way we insert phi and sigma functions. Definition 6, as opposed to the original SSI definition [1, 35], does not ensure the SSA or the SSU properties. These guarantees are not necessary to every sparse analysis. It is a common assumption in the compiler's literature that "data-flow analysis (...) can be made simpler when each variable has only one definition", as stated in Chapter 19 of Appel's textbook [2]. A naive interpretation of the above statement could lead one to conclude that data-flow analyses become simpler as soon as the program representation enforces a single source of information per live-range: SSA for forward propagation, SSU for backward, and the *original* SSI for bi-directional analyses. This premature conclusion is contradicted by the example of dead-code elimination, a backward data-flow analysis that the SSA form simplifies. Indeed, the SSA form fulfills our definition of the SSI property for dead-code elimination. Nevertheless, the corresponding constraint system may have several inequations, with the same left-hand-side, i.e., one for each use of a given variable v. Even though we may have several sources of information, we can still solve this backward analysis using the algorithm in Figure 10. To see this fact, we can replace G_v^i in Figure 10 by "*i is a useful instruction or one of its definitions is marked as useful*" and one obtains the classical algorithm for dead-code elimination.

Sparse Abstract Interpretation Framework: Recently, Oh *et al.* [28] have designed and tested a framework that sparsifies flow analyses modelled via abstract interpretation. They have used this framework to implement standard analyses on the interval [11] and on the octogon lattices [25], and have processed large code bodies. We believe that our approach leads to a sparser implementation. We base this assumption on the fact that Oh *et al.*'s approach relies on standard def-use chains to propagate information, whereas in our case, the merging nodes combine information before passing it ahead. As an example, lets consider the code `if () then a=•; else a=•; endif if () then •=a; else •=a; endif` under a forward analysis that generates information at definitions and requires it at uses. We let the symbol • denote unimportant values. In this scenario, Oh et al.'s framework creates four dependence links between the two control flow nodes where information is produced and the two control flow nodes where it is consumed. Our method, on the other hand, converts the program to SSA form; hence, creating two names for variable a. We avoid the

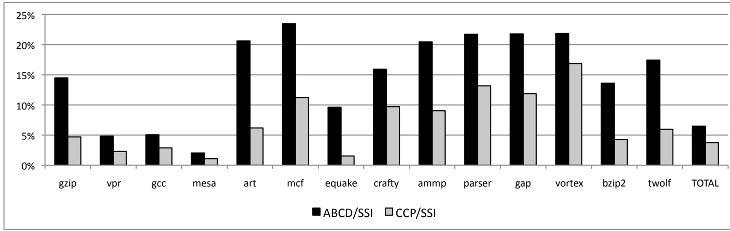

Fig. 11. Comparison of the time taken to produce the different representations. 100% is the time to use the SSI live range splitting strategy. The shorter the bar, the faster the live range splitting strategy. The SSI conversion took 1315.2s in total, the ABCD conversion took 85.2s, and the CCP conversion took 49.4s.

extra links because a ϕ-function merges the data that comes from these names before propagating it to the use sites.

5 Experimental Results

This section describes an empirical evaluation of the size and runtime efficiency of our algorithms. Our experiments were conducted on a dual core `Intel Pentium D` of 2.80GHz of clock, 1GB of memory, running `Linux Gentoo`, version 2.6.27. Our framework runs in LLVM 2.5 [22], and it passes all the tests that LLVM does. The LLVM test suite consists of over 1.3 million lines of C code. In this paper we show results for SPEC CPU 2000. To compare different live range splitting strategies we generate the program representations below. Figure 5 explains the sets *Defs, Uses* and *Conds.*

1. *SSI*: Ananian's Static Single Information form [1] is our baseline. We build the SSI program representation via Singer's iterative algorithm.
2. *ABCD*: $(\{Defs, Conds\}_\downarrow)$. This live range splitting strategy generalizes the ABCD algorithm for array bounds checking elimination [4]. An example of this live range splitting strategy is given in Figure 3.
3. *CCP*: $(\{Defs, Conds_{eq}\}_\downarrow)$. This splitting strategy, which supports Wegman *et al.*'s [39] conditional constant propagation, is a subset of the previous strategy. Differently of the ABCD client, this client requires that only variables used in equality tests, e.g., `==`, undergo live range splitting. That is, $Conds_{eq}(v)$ denotes the conditional tests that check if v equals a given value.

Runtime: The chart in Figure 11 compares the execution time of the three live range splitting strategies. We show only the time to perform live range splitting. The time to execute the optimization itself, removing array bound checks or performing constant propagation, is not shown. The bars are normalized to the running time of the SSI live range splitting strategy. On the average, the ABCD client runs in 6.8% and the CCP client runs in 4.1% of the time of SSI.

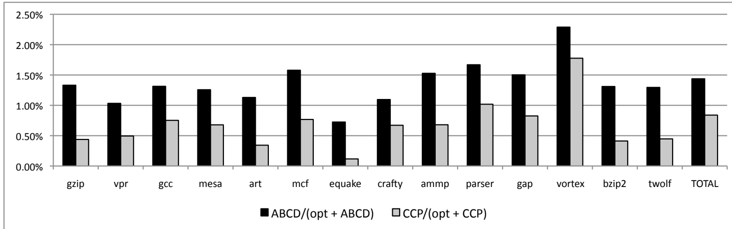

Fig. 12. Execution time of two different live range splitting strategies compared to the total time taken by machine independent LLVM optimizations (`opt` -O1). 100% is the time taken by `opt`. The shorter the bar, the faster the conversion.

These two forward analyses tend to run faster in benchmarks with sparse control flow graphs, which present fewer conditional branches, and therefore fewer opportunities to restrict the ranges of variables.

In order to put the time reported in Figure 11 in perspective, Figure 12 compares the running time of our live range splitting algorithms with the time to run the other standard optimizations in our baseline compiler[2]. In our setting, LLVM -O1 runs 67 passes, among analysis and optimizations, which include partial redundancy elimination, constant propagation, dead code elimination, global value numbering and invariant code motion. We believe that this list of passes is a meaningful representative of the optimizations that are likely to be found in an industrial strength compiler. The bars are normalized to the optimizer's time, which consists of the time taken by machine independent optimizations plus the time taken by one of the live range splitting clients, e.g, ABCD or CCP. The ABCD client takes 1.48% of the optimizer's time, and the CCP client takes 0.9%. To emphasize the speed of these passes, we notice that the bars do not include the time to do machine dependent optimizations such as register allocation.

Space: Figure 13 outlines how much each live range splitting strategy increases program size. We show results only to the ABCD and CCP clients, to keep the chart easy to read. The SSI conversion increases program size in 17.6% on average. This is an absolute value, i.e., we sum up every ϕ and σ function inserted, and divide it by the number of bytecode instructions in the original program. This compiler already uses the SSA-form by default, and we do not count as new instructions the ϕ-functions originally used in the program. The ABCD client increases program size by 2.75%, and the CCP client increases program size by 1.84%.

An interesting question that deserves attention is "What is the benefit of using a sparse data-flow analysis in practice?" We have not implemented dense versions of the ABCD or the CCP clients. However, previous works have shown that sparse analyses tend to outperform equivalent dense versions in terms of

[2] To check the list of LLVM's target independent optimizations try
`llvm-as < /dev/null | opt -std-compile-opts -disable-output -debug-pass=Arguments.`

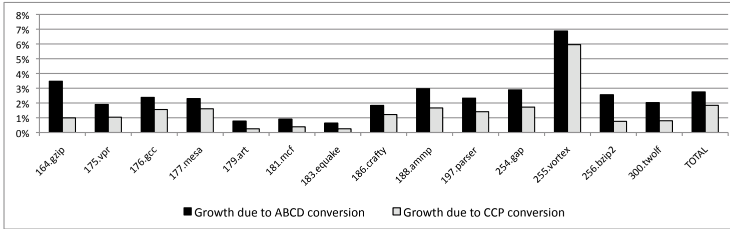

Fig. 13. Growth in program size due to the insertion of new ϕ and σ functions to perform live range splitting

time and space efficiency [9, 31]. In particular, the e-SSA format used by the ABCD and the CCP optimizations is the same program representation adopted by the tainted flow framework of Rimsa *et al.* [32, 33], which has been shown to be faster than a dense implementation of the analysis, even taking the time to perform live range splitting into consideration.

6 Conclusion

This paper has presented a systematic way to build program representations that suit sparse data-flow analyses. We build different program representations by splitting the live ranges of variables. The way in which we split live ranges depends on two factors: (i) which control flow nodes produce new information, e.g., uses, definitions, tests, etc; and (ii), how this information propagates along the variable live range: forwardly or backwardly. We have used an implementation of our framework in LLVM to convert programs to the Static Single Information form [1], and to provide intermediate representations to the ABCD array bounds-check elimination algorithm [4] and to Wegman *et al.*'s Conditional Constant Propagation algorithm [39]. Our framework has been used by Couto *et al.* [15] and by Rodrigues *et al.* [34] in different implementations of range analyses. We have also used our live range splitting algorithm, implemented in the `phc` PHP compiler [3], to provide the Extended Static Single Assignment form necessary to solve the tainted flow problem [32, 33].

Extending our Approach. For the sake of simplicity, in this paper we have restricted our discussion to: non relational analysis (PLV), intermediate-representation based appoach, and scalar variables without aliasing.

(1) *non relation analysis.* In this paper we have focused on PLV problems, i.e. solved by analyses that associate some information with each variable individually. For instance, we bind i to a range $0 \leq i <$ `MAX_N`, but we do not relate i and j, as in $0 \leq i < j$. A relational analysis that provides a all-to-all relation between all variables of the program is dense by nature, as any control flow node both produces and consumes information for the analysis. Nevertheless, our framework is compatible with the notion of *packing.* Each pack is a set of

variable groups selected to be related together. This approach is usually adopted in practical relational analyses, such as those used in Astrée [12, 25].

(2) *IR based approach.* Our framework constructs an intermediate representation (IR) that preserves the semantic of the program. Like the SSA form, this IR has to be updated, and prior to final code generation, destructed. Our own experience as compiler developers let us believe that manipulating an IR such as SSA has many engineering advantages over building, and afterward dropping, a separate sparse evaluation graph (SEG) for each analysis. Testimony of this observation is the fact that the SSA form is used in virtually every modern compiler. Although this opinion is admittedly arguable, we would like to point out that updating and destructing our SSI form is equivalent to the update and destruction of SSA form. More importantly, there is no fundamental limitation in using our technique to build a separate SEG without modifying the IR. This SEG will inherit the sparse properties as his corresponding SSI flavor, with the benefit of avoiding the quadratic complexity of direct def-use chains ($|\text{Defs}(v)| \times |\text{Uses}(v)|$ for a variable v) thanks to the use of ϕ and σ nodes. Note that this quadratic complexity becomes critical when dealing with code with aliasing or predication [28, pp.234].

(3) *analysis of scalar variables without aliasing or predication.* The most successful flavor of SSA form is the minimal and pruned representation restricted to scalar variables. The SSI form that we describe in this paper is akin to this flavor. Nevertheless, there exists several extensions to deal with code with predication (e.g. ψ-SSA form [14]) and aliasing (e.g. Hashed SSA [10] or Array SSA [16]). Such extensions can be applied without limitations to our SSI form allowing a wider range of analyses involving object aliasing and predication.

Acknowledgments. We thank the CC referees for very helpful comments on this paper, and we thank Laure Gonnord for enlightening discussions about the abstract interpretation framework. This project has been made possible by the cooperation FAPEMIG-INRIA, grant 11/2009.

References

1. Ananian, S.: The static single information form. Master's thesis. MIT (September 1999)
2. Appel, A.W., Palsberg, J.: Modern Compiler Implementation in Java, 2nd edn. Cambridge University Press (2002)
3. Biggar, P., de Vries, E., Gregg, D.: A practical solution for scripting language compilers. In: SAC, pp. 1916–1923. ACM (2009)
4. Bodik, R., Gupta, R., Sarkar, V.: ABCD: Eliminating array bounds checks on demand. In: PLDI, pp. 321–333. ACM (2000)
5. Boissinot, B., Hack, S., Grund, D., de Dinechin, B.D., Rastello, F.: Fast liveness checking for SSA-form programs. In: CGO, pp. 35–44. IEEE (2008)
6. Briggs, P., Cooper, K.D., Torczon, L.: Improvements to graph coloring register allocation. TOPLAS 16(3), 428–455 (1994)

7. Campos, V.H.S., Rodrigues, R.E., de Assis Costa, I.R., Pereira, F.M.Q.: Speed and precision in range analysis. In: de Carvalho Junior, F.H., Barbosa, L.S. (eds.) SBLP 2012. LNCS, vol. 7554, pp. 42–56. Springer, Heidelberg (2012)
8. Chambers, C., Ungar, D.: Customization: Optimizing compiler technology for self, a dynamically-typed object-oriented programming language. SIGPLAN Not. 24(7), 146–160 (1989)
9. Choi, J.-D., Cytron, R., Ferrante, J.: Automatic construction of sparse data flow evaluation graphs. In: POPL, pp. 55–66. ACM (1991)
10. Chow, F., Chan, S., Liu, S.-M., Lo, R., Streich, M.: Effective representation of aliases and indirect memory operations in SSA form. In: Gyimóthy, T. (ed.) CC 1996. LNCS, vol. 1060, pp. 253–267. Springer, Heidelberg (1996)
11. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, pp. 238–252. ACM (1977)
12. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Rival, X.: Why does astrée scale up? Form. Methods Syst. Des. 35(3), 229–264 (2009)
13. Cytron, R., Ferrante, J., Rosen, B.K., Wegman, M.N., Zadeck, F.K.: Efficiently computing static single assignment form and the control dependence graph. TOPLAS 13(4), 451–490 (1991)
14. de Ferrière, F.: Improvements to the ψ-SSA representation. In: SCOPES, pp. 111–121. ACM (2007)
15. Teixeira, D.C., Pereira, F.M.Q.: The design and implementation of a non-iterative range analysis algorithm on a production compiler. In: SBLP, pp. 45–59. SBC (2011)
16. Fink, S.J., Knobe, K., Sarkar, V.: Unified analysis of array and object references in strongly typed languages. In: SAS 2000. LNCS, vol. 1824, pp. 155–174. Springer, Heidelberg (2000)
17. Gawlitza, T., Leroux, J., Reineke, J., Seidl, H., Sutre, G., Wilhelm, R.: Polynomial precise interval analysis revisited. Efficient Algorithms 1, 422–437 (2009)
18. George, L., Matthias, B.: Taming the IXP network processor. In: PLDI, pp. 26–37. ACM (2003)
19. An, J.H., Chaudhuri, A., Foster, J.S., Hicks, M.: Dynamic inference of static types for ruby. In: POPL, pp. 459–472. ACM (2011)
20. Johnson, R., Pearson, D., Pingali, K.: The program tree structure. In: PLDI, pp. 171–185. ACM (1994)
21. Johnson, R., Pingali, K.: Dependence-based program analysis. In: PLDI, pp. 78–89. ACM (1993)
22. Lattner, C., Adve, V.S.: LLVM: A compilation framework for lifelong program analysis & transformation. In: CGO, pp. 75–88. IEEE (2004)
23. Lo, R., Chow, F., Kennedy, R., Liu, S.-M., Tu, P.: Register promotion by sparse partial redundancy elimination of loads and stores. In: PLDI, pp. 26–37. ACM (1998)
24. Mahlke, S., Ravindran, R., Schlansker, M., Schreiber, R., Sherwood, T.: Bitwidth cognizant architecture synthesis of custom hardware accelerators. TCAD 20(11), 1355–1371 (2001)
25. Miné, A.: The octagon abstract domain. Higher Order Symbol. Comput. 19, 31–100 (2006)
26. Nanda, M.G., Sinha, S.: Accurate interprocedural null-dereference analysis for java. In: ICSE, pp. 133–143 (2009)
27. Nielson, F., Nielson, H.R., Hankin, C.: Principles of program analysis. Springer (2005)

28. Oh, H., Heo, K., Lee, W., Lee, W., Yi, K.: Design and implementation of sparse global analyses for c-like languages. In: PLDI, pp. 229–238. ACM (2012)
29. Pingali, K., Bilardi, G.: Optimal control dependence computation and the roman chariots problem. In: TOPLAS, pp. 462–491. ACM (1997)
30. Plevyak, J.B.: Optimization of Object-Oriented and Concurrent Programs. PhD thesis, University of Illinois at Urbana-Champaign (1996)
31. Ramalingam, G.: On sparse evaluation representations. Theoretical Computer Science 277(1-2), 119–147 (2002)
32. Rimsa, A., d'Amorim, M., Quintão Pereira, F.M.: Tainted flow analysis on e-SSA-form programs. In: Knoop, J. (ed.) CC 2011. LNCS, vol. 6601, pp. 124–143. Springer, Heidelberg (2011)
33. Rimsa, A.A., D'Amorim, M., Pereira, F.M.Q., Bigonha, R.: Efficient static checker for tainted variable attacks. Science of Computer Programming 80, 91–105 (2014)
34. Rodrigues, R.E., Campos, V.H.S., Pereira, F.M.Q.: A fast and low overhead technique to secure programs against integer overflows. In: CGO, pp. 1–11. ACM (2013)
35. Singer, J.: Static Program Analysis Based on Virtual Register Renaming. PhD thesis, University of Cambridge (2006)
36. Stephenson, M., Babb, J., Amarasinghe, S.: Bitwidth analysis with application to silicon compilation. In: PLDI, pp. 108–120. ACM (2000)
37. Su, Z., Wagner, D.: A class of polynomially solvable range constraints for interval analysis without widenings. Theoretical Computeter Science 345(1), 122–138 (2005)
38. Tobin-Hochstadt, S., Felleisen, M.: The design and implementation of typed scheme. In: POPL, pp. 395–406 (2008)
39. Wegman, M.N., Zadeck, F.K.: Constant propagation with conditional branches. TOPLAS 13(2) (1991)
40. Weiss, M.: The transitive closure of control dependence: The iterated join. TOPLAS 1(2), 178–190 (1992)
41. Zadeck, F.K.: Incremental Data Flow Analysis in a Structured Program Editor. PhD thesis, Rice University (1984)