Parameterized Construction of Program Representations for Sparse Dataflow Analyses

André L. C. Tavares[1], Mariza A. S. Bigonha[1], Roberto S. Bigonha[1], Benoit Boissinot[2], Fernando M. Q. Pereira[1] and Fabrice Rastello[2]

[1] UFMG – 6627 Antônio Carlos Av, 31.270-010, Belo Horizonte, Brazil
{andrelct,mariza,bigonha,fernando}@dcc.ufmg.br
[2] ENS Lyon – 46 allée d'Italie, 69364 Lyon, France
{fabrice.rastello,benoit.boissinot}@ens-lyon.fr

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 by 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 [1, 6, 12, 21, 50, 27, 31, 33, 39, 41, 43, 44, 47, 49, 51]. As an example, for each program point p, and each integer variable v live at p, Stephenson *et al.*'s [47] bit-width analysis finds the size, in bits, of v at p. Although well studied in the literature, this approach has some drawbacks; in particular, it suffers from an excess of 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. This observation is not new, and there have been, along the history of optimizing compilers, different attempts to implement data-flow analyses sparsely.

The Static Single Assignment (SSA) form [20], 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 [6], the *Static Single Information* (SSI) form [2, 7, 45], and the *Static Single Use* (SSU) form [39, 24, 29]. Each of these representations extends the reach of the SSA form to more sparse data-flow analyses; however, there is not a format that subsumes all 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) [14]. 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 the other variables in the same program. This family of analyses is known as *Partitioned Variable Problems* in the literature [53].

In this paper, we propose a framework that is general enough to include all these previous approaches. Given a data-flow problem, which is defined by (i) a set of control flow points, 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. To build these program representations, we use an algorithm that is as powerful as the method that Singer has used to convert a program to SSI form [45]. However, our algorithm is faster: as we show in Section 3, for all unidirectional and all non-truly bidirectional data-flow analysis we can avoid iterating the live range splitting process that builds intermediate representations.

We have implemented our framework in the LLVM compiler [28], and have used it to provide intermediate representations to well-known compiler optimizations: Wegman *et al.*'s [51] conditional constant propagation, and Bodik *et al.*'s [6] 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 [11, 22, 41, 42].

2 Static Single Information

Our objective is to generate program representations that bestow the *Static Single Information* (Definition 6) property 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* any instruction in the source code, in addition to control flow branches. Consecutive program points are related by predicates *preds* and *succs*, such that *preds*(i) is the set of all the predecessors of point i, and if $s \in \mathit{preds}(i)$, then $i \in \mathit{succs}(s)$. A *transfer function* determines how information flows among these 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 considered *meet nodes*, because they combine information that comes 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 i, an element of a lattice $\mathcal{L}$, given by the equation $[x]^i = \bigwedge_{s \in preds(i)} F^s([x]^s)$, where $[x]^i$ denotes the abstract state associated with variable x at program point i, F^s is the transfer function associated with program point s. The analysis can be written as a constraint system that binds to each program point i and each $s \in preds(i)$ the equation $[x]^i = [x]^i \wedge F^s([x]^s)$ or, equivalently, the inequation $[x]^i \sqsubseteq F^s([x]^s)$.*

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 state 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. Many data-flow problems are PLV; however, there exists problems that do not fit into this framework, such as the relational analyses on pentagons [30], octagons [32] and general polyhedrons [17]. Nevertheless, we can still handle them conservatively via the technique known as *packing* [18, 32], in which related variables are grouped together, under a single representative.

Definition 2 (Partitioned Lattice per Variable Problem). *The Maximum Fixed Point problem on a data-flow system is a* Partitioned Lattice per Variable Problem *(PLV) 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 i.e. each transfer function F^s can also be decomposed into a product $F^s_{v_1} \times F^s_{v_2} \times \cdots \times F^s_{v_n}$ where $F^s_{v_j}$ is a function from $\mathcal{L}$ to $\mathcal{L}_{v_j}$.*

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* $\bot$*, e.g.,* $F_{v_i}(x) = \bot$

A sparse data-flow analyses propagates information from the point where this information is created directly to the point 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:*

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

A *backward* data-flow analysis combines information that flows out of a node to determine the information that flows into it. A *forward* analysis propagates information in the opposite direction. Some program points are considered *meet nodes*, because they combine the information that comes from two or more regions. For instance, two different definitions of the same variable v might be associated with two different constants; hence, providing two different pieces of information about v.

Definition 5 (Meet Nodes). *Consider a forward (resp. backward) monotone PLV problem, where* (Y_u^j) *is the maximum fixed point solution of variable* u *at program point* j*. We say that a program point* i *is a meet node for variable* v *if, and only if,* i *has* n *predecessors (resp. successors),* $s_1, \ldots, s_n, n > 2$*, and there exists* i*,* j*,* $1 \leq i < j \leq n$*, 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 i if there is a path from i to an instruction *inst'* that uses v, and v is not re-defined along this way. The live range of a variable v, which we denote by *live(v)*, is the collection of program points where that variable is alive.

Definition 6 (Static Single Information). *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]: *each* $s \in$ *live*(v) *such that* F_v^s *is non-trivial nor undefined, should contain a definition (resp. last use) of* v*;*

[SPLIT-MEET]: *each meet node* i *with* n *predecessors* $\{s_1, \ldots, s_n\}$ *(resp. successors) should have a definition (resp. use) of* v *at* i*, and* n *uses (resp. definition) 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* $i \notin$ *live*(v) *should be bound to an undefined transfer function, e.g.,* $F_v^s = \lambda x.\bot$*.*

[LINK]: *each instruction* inst *for which* F_v^{inst} *depends on some* $[u]^s$ *should contain 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 points in three kinds: interior nodes, forks and joins. At each place we use a different notation to denote live range splitting.

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

$$inst \;\; \| \;\; v_1 = v'_1 \;\; \| \;\; \ldots \;\; \| \;\; v_m = v'_m$$

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 program point 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 [20]. The assignment

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

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 assignment. The above statement encapsulates m parallel copies: all the variables $v_1^j, \ldots, v_m^j$ are simultaneously copied into the variables $v_1, \ldots, v_m$.

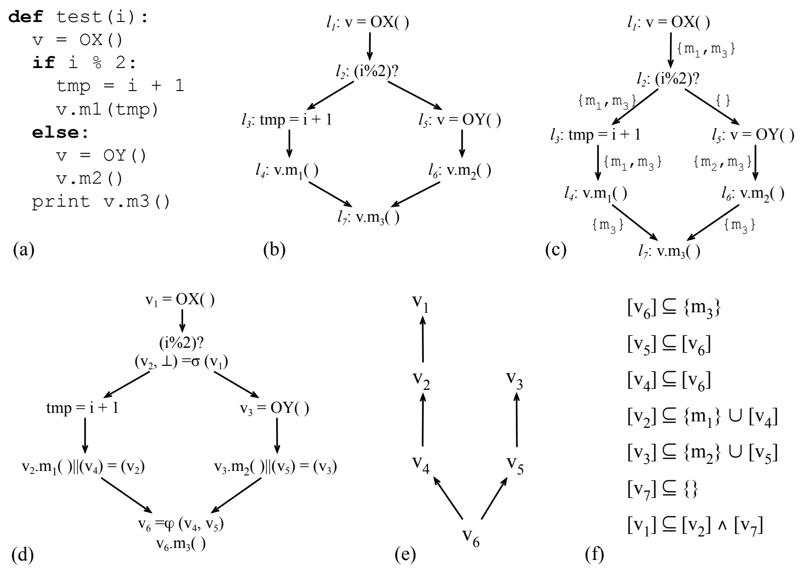
In backward analyses the information that emerges from different uses of a variable may reach the same meet node. To ensure Property [SPLIT-MEET], the use that reaches the definition of a variable must be unique, in the same way that in a SSA-form program the definition that reaches a use is unique. We ensure this property via special instructions that Ananian has called σ-functions [2]. The σ-functions are the dual of ϕ-functions, performing a parallel assignment depending on the execution path taken. The assignment

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

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.

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 this example sparsely.



```
def test(i):
  v = OX()
  if i % 2:
    tmp = i + 1
    v.m1(tmp)
  else:
    v = OY()
    v.m2()
  print v.m3()
```

$[v_6] \subseteq \{m_3\}$

$[v_5] \subseteq [v_6]$

$[v_4] \subseteq [v_6]$

$[v_2] \subseteq \{m_1\} \cup [v_4]$

$[v_3] \subseteq \{m_2\} \cup [v_5]$

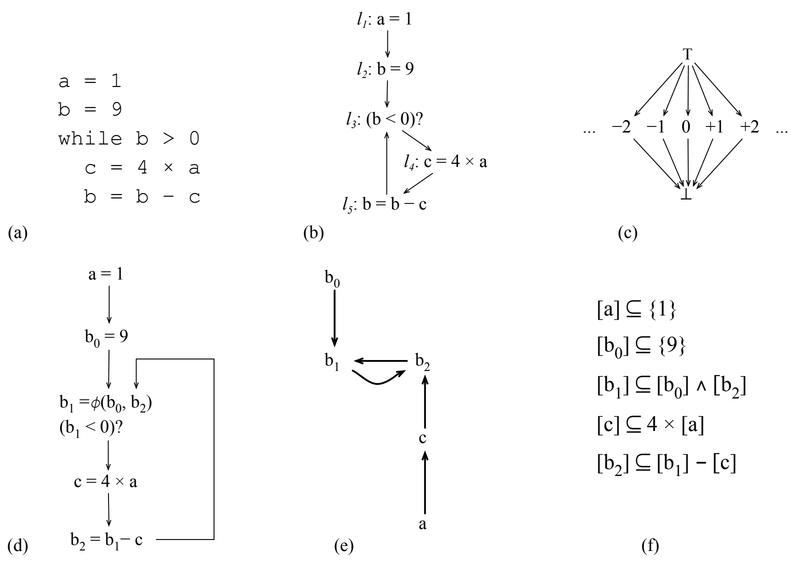
$[v_7] \subseteq \{\}$

$[v_1] \subseteq [v_2] \wedge [v_7]$

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

Class Inference: Some dynamically typed languages, such as Python, JavaScrip, Ruby or Lua, represent objects as hash tables containing methods and fields. It is possible to improve the execution of programs written in these languages if we can replace these simple tables by actual classes with virtual tables [13]. 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. Our live range splitting strategy produces, 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 program points, 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. We want to find out 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



```
a = 1
b = 9
while b > 0
  c = 4 × a
  b = b − c
```

$[a] \subseteq \{1\}$
$[b_0] \subseteq \{9\}$
$[b_1] \subseteq [b_0] \wedge [b_2]$
$[c] \subseteq 4 \times [a]$
$[b_2] \subseteq [b_1] - [c]$

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

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 representation lead to the constraint system shown in Figure 2(f).

Taint analysis: The objective of taint analysis [41] 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 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

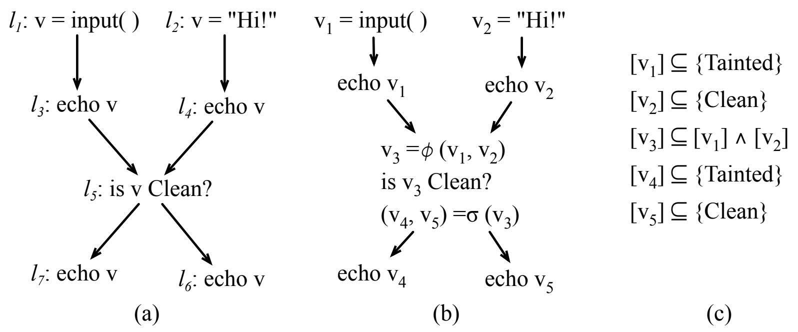


Fig. 3: Taint analysis as an example of forward data-flow analysis that takes information from the definitions of variables and conditional tests on these variables.

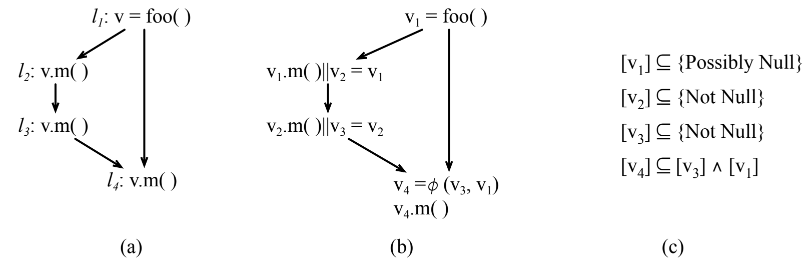


Fig. 4: Null pointer analysis as an example of forward data-flow analysis that takes information from the definitions and uses of variables.

leads 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 [6]. It also suits the ABCD algorithm for array bounds-checking elimination [6], Su and Wagner's range analysis [49] and Gawlitza *et al.*'s range analysis [23].

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 [33]. 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 a set of "oriented" program points. We let $I_\downarrow$ denote a set of points i with forward direction. Similarly, we let $I_\uparrow$ denote a set of points i with backward direction. The live-range of v must be split at least at every point 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.

- **Class inference** is a backward analysis that takes information from the uses of variables; thus, for each variable, the live-range splitting strategy is characterized by the set $\mathit{Uses}_\uparrow$ where Uses is the set of use points. 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 $\mathit{Defs}_\downarrow$ where Defs is the set of definition points. 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 points where variables are defined, and conditional tests that use these variables. For instance, in Figure 3(a), we have that $\mathcal{P}_v = \{l_1, l_2, \mathrm{Out}(l_5)\}_\downarrow$ where $\mathrm{Out}(l_i)$ denotes the exit of l_i.
- Nanda *et al.*'s **null pointer analysis** [33] is a forward flow problem that takes information from definitions and uses. For instance, in Figure 4(a), we have that $\mathcal{P}_v = \{l_1, l_2, l_3, l_4\}_\downarrow$.

The algorithm SSIfy in Figure 6 implements a live range splitting strategy in three steps. Firstly, it splits live ranges, inserting new definitions of variables into the program code. Secondly, it renames these newly created definitions; hence, ensuring that the live ranges of two different re-definitions of the same variable do not overlap. Finally, it removes dead and non-initialized definitions from the program code. We describe each of these phases in the rest of this section.
Splitting live ranges through the creation of new definitions of variables: In order to implement $\mathcal{P}_v$ we must split the live ranges of v at each program point listed by $\mathcal{P}_v$. However, these points 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 2(b), there exist two definitions of variable b: l_2 and l_5, that reach the use of b at l_3. The information that flows forward from l_2 and l_5 collides at l_3, the meet point of the if-then-else. Hence the live-range of b has to be split immediately before l_3, e.g., at $\mathrm{In}(l_3)$, leading to a new definition b_1. In general, the set of program points where information collide can be easily characterized by join sets [19]. The join set of a group of nodes P contains the CFG nodes that can be reached by two or more nodes of

Client	**Splitting strategy $\mathcal{P}$**
Alias analysis, reaching definitions cond. constant propagation [51]	$Defs_\downarrow$
Partial Redundancy Elimination [2, 45]	$Defs_\downarrow \bigcup LastUses_\uparrow$
ABCD [6], taint analysis [41], range analysis [49, 23]	$Defs_\downarrow \bigcup \mathrm{Out}(Conds)_\downarrow$
Stephenson's bitwidth analysis [47]	$Defs_\downarrow \bigcup \mathrm{Out}(Conds)_\downarrow \bigcup Uses_\uparrow$
Mahlke's bitwidth analysis [31]	$Defs_\downarrow \bigcup Uses_\uparrow$
An's type inference [25], Class inference [13]	$Uses_\uparrow$
Hochstadt's type inference [50]	$Uses_\uparrow \bigcup \mathrm{Out}(Conds)_\uparrow$
Null-pointer analysis [33]	$Defs_\downarrow \bigcup Uses_\downarrow$

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.

```
function SSIfy(var $v$, Splitting_Strategy $\mathcal{P}_v$)
    split($v$, $\mathcal{P}_v$)
    rename($v$)
    clean($v$)
```

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

P through disjoint paths. Join sets can be over-approximated by the notion of iterated dominance frontier [52], 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^+),

```
1  function split(var $v$, Splitting_Strategy $\mathcal{P}_v = I_\downarrow \cup I_\uparrow$)
2          "compute the set of split points"
3      $S_\uparrow = \emptyset$
4      foreach $i \in I_\uparrow$:
5          if $i.is\_join$:
6              foreach $e \in incoming\_edges(i)$:
7                  $S_\uparrow = S_\uparrow \bigcup \text{Out}(pDF^+(e))$
8          else:
9              $S_\uparrow = S_\uparrow \bigcup \text{Out}(pDF^+(i))$
10     $S_\downarrow = \emptyset$
11     foreach $i \in S_\uparrow \bigcup \text{Defs}(v) \bigcup I_\downarrow$:
12         if $i.is\_branch$:
13             foreach $e \in outgoing\_edges(i)$
14                 $S_\downarrow = S_\downarrow \bigcup \text{In}(DF^+(e))$
15         else:
16             $S_\downarrow = S_\downarrow \bigcup \text{In}(DF^+(i))$
17     $S = \mathcal{P}_v \bigcup S_\uparrow \bigcup S_\downarrow$
18         "Split live range of $v$ by inserting $\phi$, $\sigma$, and copies"
19     foreach $i \in S$:
20         if $i$ does not already contain any definition of $v$:
21             if $i.is\_join$: insert "$v = \phi(v, ..., v)$" at $i$
22             elseif $i.is\_branch$: insert "$(v, ..., v) = \sigma(v)$" at $i$
23             else: insert a copy "$v = v$" at $i$
```

Fig. 7: Live range splitting. We use $\text{In}(l)$ to denote a program point immediately before l, and $\text{Out}(l)$ to denote a program point immediately after l.

which is the dual of DF^+ [3]. That is, the post-dominance frontier is the dominance frontier in a CFG where direction of edges have been reversed.

Figure 7 shows the algorithm that we use to create new definitions of variables. This algorithm has three main 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 points that originate information. If a program point is a join node, then each of its predecessors will contain the live range of a different definition of v, as we ensure in line 6 of our algorithm. Notice that these new definitions are not placed parallel to an instruction, but in the region immediately after it, which we denote by $\text{Out}(\dots)$. 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 (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. Nevertheless, as we show in the Appendix, our method ensures the SSI properties for any combination of unidirectional problems.

```
function rename(var $v$)
    "Compute use-def & def-use chains"
    "We consider here that stack.peek() $= \bot$ if stack.isempty(),
      and that $def(\bot) = entry$"
    $stack = \emptyset$
    foreach CFG node $n$ in dominance order:
        if exists $v = \phi(v : l^1, \ldots, v : l^q)$ in In($n$):
            stack.set_def($v = \phi(v : l^1, \ldots, v : l^q)$)
        foreach instruction $u$ in $n$ that uses $v$:
            stack.set_use($u$)
        if exists instruction $d$ in $n$ that defines $v$:
            stack.set_def($d$)
        foreach instruction $(\ldots) = \sigma(v)$ in Out($n$):
            stack.set_use($(\ldots) = \sigma(v)$)
        if exists $(v : l^1, \ldots, v : l^q) = \sigma(v)$ in Out($n$):
            foreach $v : l^i = v$ in $(v : l^1, \ldots, v : l^q) = \sigma(v)$:
                stack.set_def($v : l^i = v$)
        foreach $m$ in $successors(n)$:
            if exits $v = \phi(\ldots, v : l^n, \ldots)$ in In($m$):
                stack.set_use($v = v : l^n$)
function stack.set_use(instruction inst):
    while $def$(stack.peek()) does not dominate inst: stack.pop()
    $v_i$ = stack.peek()
    replace the uses of $v$ by $v_i$ in inst
    if $v_i \neq \bot$: set Uses($v_i$) = Uses($v_i$) $\bigcup$ inst
function stack.set_def(instruction inst):
    let $v_i$ be a fresh version of $v$
    replace the defs of $v$ by $v_i$ in inst
    set Def($v_i$) = inst
    stack.push($v_i$)
```

Fig. 8: Versioning

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 [8], which is useful in actual compiler implementations. This algorithm inserts ϕ and σ functions conservatively. Consequently, we may have these special instructions at program points 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 [3, 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

```
1  clean(var $v$)
2        let web = $\{v_i | v_i$ is a version of $v\}$
3        let defined = $\emptyset$
4        let active = $\{$ $inst$ $|inst$ actual instruction and web $\cap$ $inst$.defs $\neq \emptyset\}$
5        while $\exists inst \in$ active s.t. web $\cap$ $inst$.defs$\setminus$defined $\neq \emptyset$:
6              foreach $v_i \in$ web $\cap$ $inst$.defs$\setminus$defined:
7                    active = active $\cup$ Uses($v_i$)
8                    defined = defined $\cup$ $\{v_i\}$
9        let used = $\emptyset$
10       let active = $\{$ $inst$ $|inst$ actual instruction and web $\cap$ $inst$.uses $\neq \emptyset\}$
11       while $\exists inst \in$ active s.t. $inst$.uses$\setminus$used $\neq \emptyset$:
12             foreach $v_i \in$ web $\cap$ $inst$.uses$\setminus$used:
13                   active = active $\cup$ Def($v_i$)
14                   used = used $\cup$ $\{v_i\}$
15       let live = defined $\cap$ used
16       foreach non actual $inst \in$ Def($web$):
17             foreach $v_i$ operand of $inst$ s.t. $v_i \notin$ live:
18                         replace $v_i$ by $\perp$
19             if $inst$.defs = $\{\perp\}$ or $inst$.uses = $\{\perp\}$
20                   remove $inst$
```

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 variable(s) defined by *inst*, and *inst*.uses denote the set of variables used by *inst*.

definition currently on the top of the stack is used to replace all the uses of v that we find during the traversal. If the stack is empty, this means that the variable is not defined at this point. The renaming process replaces the uses of undefined variables by $\perp$ (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 19-20 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 20. 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 15-16 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 either define or use variables that do not reach any actual instruction. We mean by "actual" instructions, 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, this time to add to the active set, 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 $\perp$ in all ϕ, σ, or copy functions where it appears in. This is done by lines 15-18. Finally every useless ϕ, σ, or copy functions are removed by lines 19-20. As a historical curiosity, Cytron *et al.*'s procedure to build SSA form produced what is called *the minimal representation* [19]. Some of the ϕ-functions in the minimal representation define variables that are never used. Briggs *et al.* [9] 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:

Theorem 1 (Semantics). SSIfy *maintains the following property: if a value n written into variable v at program point i' is read at a program point i in the original program, then the same value assigned to a version of variable v at program point i' is read at a program point 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 algorithms in Figures 10 and 11 provide 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 i) we have the equation $[v]^i = [v]^i \wedge F_v^s([v_1]^s, \ldots, [v_n]^s)$. We define a system of sparse equations E_{sparse}^{ssi} as follows:*

- *Let $\{a, \ldots, b\}$ be the variables used (resp. defined) at program point i, where variable v is defined (resp. used). The LINK property ensures that F_v^s depends only on some $[a]^s \ldots [b]^s$. Thus, there exists a function G_v^s defined as the projection of F_v^s on $\mathcal{L}_a \times \cdots \times \mathcal{L}_b$, such that $G_v^s([a], \ldots, [b]) = F_v^s([v_1], \ldots, [v_n])$.*
- *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 s.*

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.*

```
function back_propagate(transfer_functions $\mathcal{G}$)
    $worklist = \emptyset$
    foreach $v \in$ vars: $[v] = \top$
    foreach $i \in$ insts: $worklist$ += $i$
    while $worklist \neq \emptyset$:
        let $i \in worklist$; $worklist$ −= $i$
        foreach $v \in i.uses()$:
            $[v]_{new} = [v] \wedge G^i_v([i.defs()])$
            if $[v] \neq [v]_{new}$:
                $stack$ += $v.defs()$
                $[v] = [v]_{new}$
```

Fig. 10: Backward propagation engine under SSI

```
function forward_propagate(transfer_functions $\mathcal{G}$)
    $worklist = \emptyset$
    foreach $v \in$ vars: $[v] = \top$
    foreach $i \in$ insts: $worklist$ += $i$
    while $worklist \neq \emptyset$:
        let $i \in worklist$; $worklist$ −= $i$
        foreach $v \in i.defs()$:
            $[v]_{new} = [v] \wedge G^i_v([i.uses()])$
            if $[v] \neq [v]_{new}$:
                $stack$ += $v.uses()$
                $[v] = [v]_{new}$
```

Fig. 11: Forward propagation engine under SSI

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* [14] are one of the earliest techniques designed to support sparse analyses. The nodes of 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* [27], or Ramalingan's *Compact Evaluation Graphs* [40]. Nowadays we have efficient algorithms that build such data-structures [26, 37, 38]. 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*. We recall Zadeck's definition below:

Definition 8 (Partitioned Variable Problem). *A PLV problem (Definition 2) is said to be also a Partitioned Variable Problem (PVP) if, and only if,* *each transfer function F^s can be decomposed into a product $F^s_{v_1} \times F^s_{v_2} \times \cdots \times F^s_{v_n}$ where $F^s_{v_j}$ is a function from $\mathcal{L}_{v_j}$ to $\mathcal{L}_{v_j}$.*

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 program point 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 [2]. This representation was later revisited by Jeremy Singer [45] and Boissinot *et al.* [7]. Singer provided new algorithms to construct the SSI form, and Boissinot *et al.* clarified a number of omissions in the related literature. 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 ϕ and σ functions. Consequently, as we will show in Section 5, we insert less ϕ and σ 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, the literature describes other program representations that sparsify specific data-flow analyses. For instance, the *Extended Static Single Assignment* (e-SSA) form was introduced by Bodik *et al.* [6] to implement the "less-than" [30] lattice sparsely. As opposed to SSI and SSA, the e-SSA form supports flow analyses that obtain information from conditional tests. As another example, the the *Static Single Use* form (SSU) supports analyses that extract information from uses. As uses and definitions are not fully symmetric – the live-range can "traverse" a use while it cannot traverse a definition – there exists different variants of SSU [39, 24, 29]. For instance, the "strict" SSU form enforces that each definition reaches a single use, whereas SSI and other variations of SSU allow two consecutive uses of a variable on the same path. Our method is also more general than these previous approaches, as we can produce them via different parameterizations. For SSI we have $\{Defs_\downarrow \cup Uses_\uparrow\}$; for e-SSA we have $Defs_\downarrow \bigcup \mathrm{Out}(Conds)_\downarrow$, and for Strict SSU we have $Defs_\downarrow \bigcup Uses_\downarrow$.

The SSI constrained system might have several inequations for the same left-hand-side, due to the way we insert ϕ and σ functions. Definition 6, as opposed

to the original SSI definition [2, 45], 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 [3]. 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 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. In fact, 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.* [35] 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 [16] and on the octogon lattices [32], and have processed large code bodies. We cannot directly compare our approach with Oh *et al.*'s tool because it is not publicly available. However, 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. In this scenario, Oh et al.'s framework creates four dependence links between the two points where information is produced and the two points 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 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 [28], 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*.

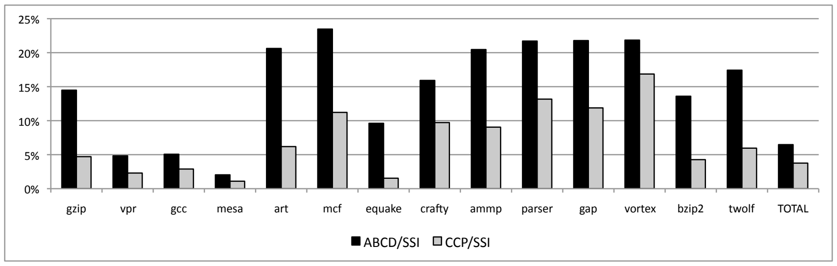


Fig. 12: Comparison of the time taken to produce the different representations. 100% is the time to use the SSI live range splitting strategy. The shorter the bar, the faster the live range splitting strategy. The SSI conversion took 1315.2s in total, the ABCD conversion took 85.2s, and the CCP conversion took 49.4s.

1. *SSI*: Ananian's Static Single Information form [2] is our baseline. We build the SSI program representation via Singer's iterative algorithm.
2. *ABCD*: $(\{def, cond\}_\downarrow)$. This live range splitting strategy generalizes the ABCD algorithm for array bounds checking elimination [6]. An example of this live range splitting strategy is given in Figure 3.
3. *CCP*: $(\{def, cond_{eq}\}_\downarrow)$. This splitting strategy, which supports Wegman *et al.*'s [51] 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, $cond_{eq}(v)$ denotes the conditional tests that check if v equals a given value.

Runtime: The chart in Figure 12 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 bounds check 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. 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 12 in perspective, Figure 13 compares the running time of our live range splitting algorithms with the time to run the other standard optimizations in our baseline compiler[3]. 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,

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

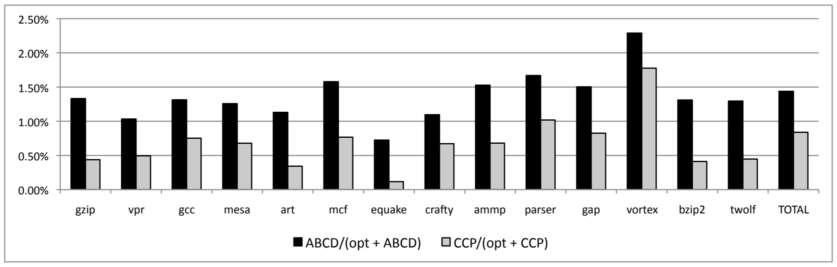


Fig. 13: Execution time of two different live range splitting strategies compared to the total time taken by machine independent LLVM optimizations (`opt`). 100% is the time taken by `opt`. The shorter the bar, the faster the conversion.

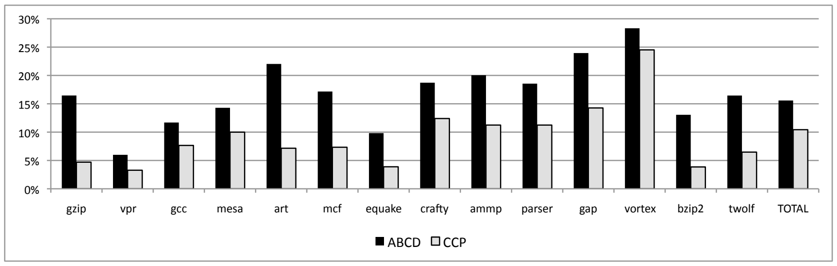


Fig. 14: Number of ϕ and σ-functions produced by different live range splitting strategies. 100% is the number of instructions inserted by the SSI conversion.

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 14 compares the number of ϕ and σ-functions inserted by each splitting strategy. The bars give the sum of these instructions, as inserted by each conversion, divided by the number of σ and ϕ-functions inserted by the SSI live range splitting strategy. The CCP client created 67.3K σ-functions, and 28.4K ϕ-functions. The ABCD client created 98.8K σ-functions, and 42.0K ϕ-functions. The SSI conversion inserted 697.6K σ-functions, and 220.6K σ-functions.

Finally, Figure 15 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

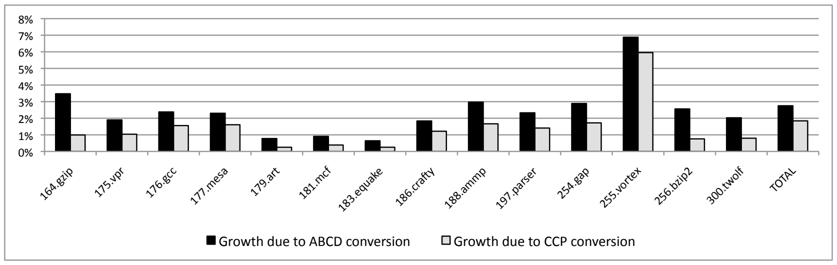


Fig. 15: Growth in program size due to the insertion of new ϕ and σ functions to perform live range splitting.

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 time and space efficiency [14, 40]. 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.* [41], 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 program points 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 [2], and to provide intermediate representations to the ABCD array bounds-check elimination algorithm [6] and to Wegman *et al.*'s Conditional Constant Propagation algorithm [51]. Our framework has been used by Couto *et al.* [22] and by Rodrigues *et al.* [42] in different implementations of range analyses. We have also used our live range splitting algorithm, implemented in the `phc` PHP compiler [4, 5], to provide the Extended Static Single Assignment form necessary to solve the tainted flow problem [41]. Finally, we observe that our program representations are compatible with many previous extensions of the SSA form, such as HSSA-form [15], ψ-SSA form [48] and the gated-SSA form [36].

References

1. W. B. Ackerman. *Efficient Implementation of Applicative Languages*. PhD thesis, MIT, 1984.
2. Scott Ananian. The static single information form. Master's thesis, MIT, September 1999.
3. Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, 2nd edition, 2002.
4. Paul Biggar. *Design and Implementation of an Ahead-of-Time Compiler for PHP*. PhD thesis, Trinity College Dublin, 2009.
5. Paul Biggar, Edsko de Vries, and David Gregg. A practical solution for scripting language compilers. In *SAC*, pages 1916–1923. ACM, 2009.
6. Rastislav Bodik, Rajiv Gupta, and Vivek Sarkar. ABCD: eliminating array bounds checks on demand. In *PLDI*, pages 321–333. ACM, 2000.
7. Benoit Boissinot, Alain Darte, Fabrice Rastello, Benoit Dupont de Dinechin, and Christophe Guillon. Revisiting out-of-SSA translation for correctness, code quality, and efficiency. In *CGO*, pages XX–XX. IEEE, 2009.
8. Benoit Boissinot, Sebastian Hack, Daniel Grund, Benoit Dupont de Dinechin, and Fabrice Rastello. Fast liveness checking for SSA-form programs. In *CGO*, pages 35–44. IEEE, 2008.
9. Preston Briggs, Keith D. Cooper, and Linda Torczon. Improvements to graph coloring register allocation. *TOPLAS*, 16(3):428–455, 1994.
10. Zoran Budimlic, Keith D. Cooper, Timothy J. Harvey, Ken Kennedy, Timothy S. Oberg, and Steven W. Reeves. Fast copy coalescing and live-range identification. In *PLDI*, pages 25–32. ACM, 2002.
11. Victor Hugo Sperle Campos, Raphael Ernani Rodrigues, Igor Rafael de Assis Costa, and Fernando Magno Quintao Pereira. Speed and precision in range analysis. In *SBLP*, pages 42–56. Springer, 2012.
12. Robert Cartwright and Mattias Felleisen. The semantics of program dependence. *SIGPLAN Not.*, 24(7):13–27, 1989.
13. Craig Chambers and David Ungar. Customization: optimizing compiler technology for self, a dynamically-typed object-oriented programming language. *SIGPLAN Not.*, 24(7):146–160, 1989.
14. Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In *POPL*, pages 55–66. ACM, 1991.
15. Fred C. Chow, Sun Chan, Shin-Ming Liu, Raymond Lo, and Mark Streich. Effective representation of aliases and indirect memory operations in ssa form. In *CC*, pages 253–267. Springer-Verlag, 1996.
16. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252. ACM, 1977.
17. P. Cousot and N.. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–96. ACM, 1978.
18. Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. Why does astr&#233;e scale up? *Form. Methods Syst. Des.*, 35(3):229–264, 2009.
19. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. In *POPL*, pages 25–35, 1989.

20. Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *TOPLAS*, 13(4):451–490, 1991.
21. Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *POPL*, pages 207–212, New York, NY, USA, 1982. ACM.
22. Douglas do Couto Teixeira and Fernando Magno Quintao Pereira. The design and implementation of a non-iterative range analysis algorithm on a production compiler. In *SBLP*, pages 45–59. SBC, 2011.
23. Thomas Gawlitza, Jerome Leroux, Jan Reineke, Helmut Seidl, Gregoire Sutre, and Reinhard Wilhelm. Polynomial precise interval analysis revisited. *Efficient Algorithms*, 1:422 – 437, 2009.
24. Lal George and Blu Matthias. Taming the ixp network processor. In *PLDI*, pages 26–37. ACM, 2003.
25. Jong hoon An, Avik Chaudhuri, Jeffrey S. Foster, and Michael Hicks. Dynamic inference of static types for ruby. In *POPL*, pages XX–XX. ACM, 2011.
26. R. Johnson, D. Pearson, and K. Pingali. The program tree structure. In *PLDI*, pages 171–185. ACM, 1994.
27. Richard Johnson and Keshav Pingali. Dependence-based program analysis. In *PLDI*, pages 78–89. ACM, 1993.
28. Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88. IEEE, 2004.
29. Raymond Lo, Fred Chow, Robert Kennedy, Shin-Ming Liu, and Peng Tu. Register promotion by sparse partial redundancy elimination of loads and stores. In *PLDI*, pages 26–37. ACM, 1998.
30. Fancesco Logozzo and Manuel Fahndrich. Pentagons: a weakly relational abstract domain for the efficient validation of array accesses. In *SAC*, pages 184–188. ACM, 2008.
31. S. Mahlke, R. Ravindran, M. Schlansker, R. Schreiber, and T. Sherwood. Bitwidth cognizant architecture synthesis of custom hardware accelerators. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 20(11):1355–1371, 2001.
32. Antoine Miné. The octagon abstract domain. *Higher Order Symbol. Comput.*, 19:31–100, 2006.
33. Mangala Gowri Nanda and Saurabh Sinha. Accurate interprocedural null-dereference analysis for java. In *ICSE*, pages 133–143, 2009.
34. Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 2005.
35. Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. Design and implementation of sparse global analyses for c-like languages. In *PLDI*, pages 1–11. ACM, 2012.
36. Karl J. Ottenstein, Robert A. Ballance, and Arthur B. MacCabe. The program dependence web: a representation supporting control-, data-, and demand-driven interpretation of imperative languages. In *PLDI*, pages 257–271. ACM, 1990.
37. Keshav Pingali and Gianfranco Bilardi. APT: A data structure for optimal control dependence computation. In *PLDI*, pages 211–222. ACM, 1995.
38. Keshav Pingali and Gianfranco Bilardi. Optimal control dependence computation and the roman chariots problem. In *TOPLAS*, pages 462–491. ACM, 1997.
39. John Bradley Plevyak. *Optimization of Object-Oriented and Concurrent Programs*. PhD thesis, University of Illinois at Urbana-Champaign, 1996.
40. G. Ramalingam. On sparse evaluation representations. *Theoretical Computer Science*, 277(1-2):119–147, 2002.

41. Andrei Alves Rimsa, Marcelo D'Amorim, and Fernando M. Q. Pereira. Tainted flow analysis on e-SSA-form programs. In *CC*, pages 124–143. Springer, 2011.
42. Raphael Ernani Rodrigues, Victor Hugo Sperle Campos, and Fernando Magno Quintao Pereira. A fast and low overhead technique to secure programs against integer overflows. In *CGO*. ACM, 2013.
43. Subhajit Roy and Y. N. Srikant. The hot path ssa form: Extending the static single assignment form for speculative optimizations. In *CC*, pages 304–323, 2010.
44. Bernhard Scholz, Chenyi Zhang, and Cristina Cifuentes. User-input dependence analysis via graph reachability. Technical report, Sun, Inc., 2008.
45. Jeremy Singer. *Static Program Analysis Based on Virtual Register Renaming*. PhD thesis, University of Cambridge, 2006.
46. Vugranam C. Sreedhar, Roy Dz ching Ju, David M. Gillies, and Vatsa Santhanam. Translating out of static single assignment form. In *SAS*, pages 194–210. Springer-Verlag, 1999.
47. Mark Stephenson, Jonathan Babb, and Saman Amarasinghe. Bitwidth analysis with application to silicon compilation. In *PLDI*, pages 108–120. ACM, 2000.
48. Arthur Stoutchinin and Francois de Ferriere. Efficient static single assignment form for predication. In *MICRO*, pages 172–181. IEEE, 2001.
49. Zhendong Su and David Wagner. A class of polynomially solvable range constraints for interval analysis without widenings. *Theoretical Computeter Science*, 345(1):122–138, 2005.
50. Sam Tobin-Hochstadt and Matthias Felleisen. The design and implementation of typed scheme. *POPL*, pages 395–406, 2008.
51. Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *TOPLAS*, 13(2), 1991.
52. Michael Weiss. The transitive closure of control dependence: the iterated join. *TOPLAS*, 1(2):178–190, 1992.
53. Frank Kenneth Zadeck. *Incremental Data Flow Analysis in a Structured Program Editor*. PhD thesis, Rice University, 1984.

A Isomorphism to Sparse Evaluation Graphs

Given a control flow graph G, Choi *et al.* define a sparse evaluation graph as a tuple $\langle N_{SG}, E_{SG}, M\rangle$, such that:

– N_{SG} is a set of nodes defined as follows:
  1. N_{SG} contains a node n_s representing the entry point $s \in G$;
  2. N_{SG} contains a node n_p for each point $p \in G$ that is associated with a non-identity transfer function.
  3. N_{SG} contains a node n_m for each point m in the iterated dominance frontier of the points of G used to build the nodes in step (1) and (2). These are called *meet* nodes.

– We let P denote the set of points $p \in G$ used in step 2 above, plus the point $s \in G$ used in step 1 above; we let M denote the set of points $m \in G$ used in step 3 above; if we let $S = P \bigcup M$ then we define E_{SG} as follows:
  1. there is an edge $(n_q, n_m) \in N_{SG}^2$ whenever $m \in M$ and q is, among all the nodes in S, the immediate dominator of one of the CFG predecessors of m. See `search(3b)` and `link(2b)` in Choi *et al* [14];

2. there is an edge $(n_q, n_p) \in N_{SG}^2$ whenever $p \in P$, and q is, among all the nodes in S, the immediate dominator of p. See `search(1)` and `link(2b)` [14];

– The mapping function $M : E_G \mapsto N_{SG}$ associates to each edge (u, v) of the CFG the node $n_q \in N_{SG}$, whenever $q \in S$ is the immediate dominator of $u \in G$. See `search(3a)` [14]. This is done through the recursive function `search` that performs a topological traversal of the CFG (DFS of the dominance tree; See `search(4)` [14]).

Theorem 3 states that, for forward partitioned variable data-flow problems (PVP), the algorithm in Figure 6 can build program representations isomorphic to Sparse Evaluation Graphs. The proof that this result holds for backward data-flow problems, is analogous, and we omit it.

Lemma 1 (CFG cover). *Let $Prog$ be a program with its corresponding CFG G with start node s, and exit node x. Let $Prog'$ be the program that we obtain from $Prog$ by:*

1. *adding a pseudo-definition of each variable to s;*
2. *adding a pseudo-use of each variable to x;*
3. *placing a pseudo-use of a variable v at each point where v is defined;*
4. *converting the resulting program into SSA form.*

If v is a variable in $Prog$, then the live ranges of the different names of v in $Prog'$ completely partition the program points of G. In other words, each program point of G belongs to exactly one live range of v in $Prog'$.

Proof. First, v is alive at every point of G, due to transformations (1), (2) and (3). Therefore, if V is the set of the different names of v after the conversion to SSA form in step (4), then any program point of G belongs to the live range of at least one $v' \in V$. The result follows from a well-know property of Cytron's SSA-form conversion algorithm [20], which, as observed by Sreedhar *et al.* [46], creates variables with non-intersecting live ranges. In other words, after the SSA renaming, two different names of v cannot be simultaneously alive at a program point p.

[Equivalence SSI/SEG - See Theorem 3] 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.

Proof. We argue that the SEG of v is isomorphic to the representation of v in $Prog'$, the program representation that we derive from $Prog$ by applying the transformations 1-3 listed in Lemma 1 in addition to a pass of SSIfy. If we let P, as before, be defined as the set of CFG points associated with non-identity transfer functions, plus the start node s of the CFG, then after we apply the splitting strategy $P_\downarrow$, we have that:

1. there will be exactly one definition per node of P and one definition per node of $DF^+(P)$. So there is an one-to-one correspondence between SSA definitions and SG nodes.
2. From Lemma 1 the live-ranges of the different names of v provides a partitioning of the points of G. If v' is a new name of v, then each program point where v' is alive is dominated by v''s definition[4]. Each program point belongs to the live-range of the name of v whose definition immediately dominates it (among all definitions). Thus, live ranges give origin to a function that maps SSA definitions to program points. Consequently, there is an isomorphism between the live-ranges and the mapping function M.
3. def-use chains on $Prog'$ are isomorphic to the edges in E_{SG}: indeed a SEG node n_p is linked to n_q whenever (i) n_p immediately dominates n_q if $q \in P$; or (ii) n_q is in the dominance frontier of n_p if $q \in M$. In the former case the definition of v at p reaches the (pseudo-)use of v at q. In the latter this definition reaches the use of v at the ϕ-function placed at q by $\mathsf{SSIfy}(v, P_\downarrow)$.

In the proof of Theorem 3 we had to augment the program with a pseudo-definition of v at the CFG's entry point and a pseudo-use at every actual definition of v and at the CFG's exit point. The difference between a code with or without pseudo uses/defs is related to the necessity to compute data-flow information beyond the live-ranges of variables or not. This necessity exists for optimizations such as partial redundancy elimination, which may move, create or delete code.

Figure 16 compares SEG and the forward live range splitting strategy in the example taken from Figure 11 of Choi *et al.* [14], which shows the reaching uses analysis. In the left we see the original program, and in the middle the SEG built for a forward flow analysis that extracts information from uses of variables. We have augmented the edges in the left CFG with the mapping M of SEG nodes to CFG edges. In the right we see the same CFG, augmented with pseudo defs and uses, after been transformed by SSIfy applied on the points $\{S, 4, 5, 7, 11, 12\}_\downarrow$. The edges of this CFG are labeled with the definitions of v live there.

B Correctness of our SSIfication

In this section we consider a unidirectional forward (resp. backward) PLV problem stated as a set of equations $[v]^i = [v]^i \wedge F_v^s(\dots)$ for every variable v, each program point i, and each $s \in \mathit{preds}(i)$ (resp. $s \in \mathit{succs}(i)$). We rely on the nomenclature introduced by Definition 3 in order to prove Theorem 2.

Lemma 2 (Live range preservation). *If variable v is live at a program point i, then there is a version of v live at i after we run* SSIfy*.*

Proof. Split cannot remove any live range of v, as it only inserts "copies" from v to v, e.g., each copy has the same source and destination. Rename removes

[4] This is a classical result of SSA-form. See Budimlic *et al.* [10] for a proof

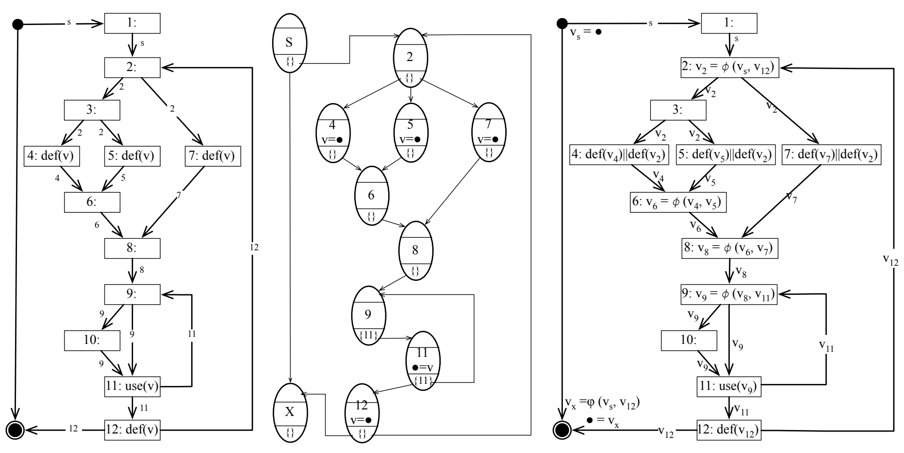


Fig. 16: Example of equivalence between SEGs and our live range splitting strategy for reaching uses.

live ranges of v, but it replaces them with the live ranges of new versions of this variable whenever a use of v is renamed. Clean only removes "copies"; hence, all the original instructions remain in the code. □

Lemma 3 (Non-Overlapping). *Two different versions of v, e.g., v_i and v_j cannot be live at a program point i transformed by* SSIfy.

Proof. The only algorithm that creates new versions of v is rename. Each new version of v is unique, as we ensure in line 27-29 of the algorithm. If rename changes the use of v to v_i at i, then there exists a definition of v_i at some program point i' that dominates i, as we ensure in line 22 of the algorithm. Lets assume that we have two versions of v, e.g., v_i and v_j, live at a program point i, in order to derive a contradiction. in this case, there exist program points i_i where v_i is used, and i_j where v_j is used, reachable from i. And exist a program point i'_i where v_i is defined, and a program point i'_j where v_j is defined, so that i'_i dominates i_i, and i'_j dominates i_j. Now, if neither i'_i dominates i'_j nor vice-versa, then we have a contradiction, because, given that i'_i reaches i_j and i'_j reaches i_i, then neither i'_i would dominates i_i, nor i'_j would dominates i_j. Without loss of generality, lets assume that i'_i dominates i'_j. in this case, rename visits i'_i first, and upon visiting i'_j, places the definition of v_j on top of the definition of v_i in the stack in line 30. Thus, i'_j cannot dominate i_i, or we would have, at that program point, a use of v_j, instead of v_i. In this case, i_j is live past the dominance frontier of i'_i, forcing split (line 14) to create a ϕ-function that dominates i_i, at a program point that is dominated by i'_i; hence, creating a new definition v_ϕ of v. Therefore, at i_i we would have a use of v_ϕ instead of v.□

[**Semantics - Theorem 1**] SSIfy maintains the following property: if a value n written to variable v at program point i' is read at a program point i in the original program, then the same value assigned to a version of variable v at program point i' is read at a program point i after transformation.

Proof. For simplicity, we will extend the meaning of "copy" to include not only the parallel copies placed at interior nodes, but also ϕ and σ-functions. Split cannot create new values, as it only inserts "copies". Clean cannot remove values, as it only removes "copies". From the hypothesis we know that the definition of v that reaches i is live at i. From Lemma 2 we know that there is a version of v live at i. From Lemma 3 we know that only one version of v can be live at i, and so rename cannot send new values to i.□

Now suppose that the program, not necessarily under SSI form, fulfills INFO and LINK from Definition 6 for a system of monotone equations E_{dense}, given as a set of constraints $[v]^i \sqsubseteq F_v^s([v_1]^s, \ldots, [v_n]^s)$. Consider a live range splitting strategy $\mathcal{P}_v$ that *includes* for each variable v the set of program points $I_\downarrow$ (resp. $I_\uparrow$) where F_v^s is non-trivial. The following theorem states that Algorithm SSIfy creates a program form that fulfills the Static Single Information property.
[**Correctness of SSIfy - Theorem 2**] Given the conditions stated above, Algorithm SSIfy(v, $\mathcal{P}_v$) 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.

Proof. We derive from this new program representation a system of equations isomorphic to the initial one by associating trivial transfer functions with the newly created "copies". The INFO and LINK properties are trivially maintained. As only trivial and constant functions have been added, monotonicity is maintained.

To show that we provide SPLIT-DEF, we must first show that each $i \in$ live(v) where F_v^s is non-trivial contains a definition (resp. last use) of v. The function split separates these points in lines 9 and 16, and later, in line 23, inserts definitions in those points. To show that we provide SPLIT-MEET, we must prove that each join (resp. split) node for which E_{dense} has possibly different values on its incoming edges should have a ϕ-function (resp. σ-function) for v. These points are separated in lines 7 and 14 of split. To see why this is the case, notice that line 7 separates the points in the iterated dominance frontier of points that originate information that flows forward. These are, as a direct consequence of the definition of iterated dominance frontier, the points where information collide. Similarly, line 14 separates the points in the post-dominance frontier of regions which originate information that flows backwardly.

We ensure VERSION as a consequence of the SSA conversion. All our program representations preserve the SSA representation, as we include the definition sites of v in line 11 of split. Function rename ensures the existence of only one definition of each variable in the program code (line 27), and that each definition dominates all its uses (consequence of the traversal order). Therefore, the newly created live ranges are connected on the dominance tree of the source program. Function rename also creates a new program representation for which it is straightforward to build a system of equations E^{ssi}_{dense} isomorphic to E_{dense}: Firstly, the constraint variables are renamed in the same way that program variables are. Secondly, for each program variable, new system variables bound to $\perp$ are created for each program point outside of its live-range.□

C Equivalence between sparse and dense analyses.

We have shown that SSIfy transforms a program P into another program P^{ssi} with the same semantics. Furthermore, this representation provides the SSI property for a system of equations E^{ssi}_{dense} that we extract from P^{ssi}. This system is isomorphic to the system of equations E_{dense} that we extract from P. From the so obtained program under SSI for the constrained system E^{ssi}_{dense}, Definition 7 shows how to construct a sparse constrained system E^{ssi}_{sparse}. When transfer functions are monotone and the lattice has finite height, Theorem 4 states the equivalence between the sparse and the dense systems. The purpose of this section is to prove this theorem. We start by introducing the notion of *coalescing*. Let E be a constraint system that associates with each $1 \leq i \leq n$ the constraint $a_i \sqsubseteq H_i(a_1, \ldots, a_n)$, where each a_i is an element of a lattice $\mathcal{L}$ of finite height, and H_i is a monotone function from $\mathcal{L}^n$ to $\mathcal{L}$. Let $(A_1, \ldots, A_n)$ be the maximum solution to this system, and let $1 \leq m \leq n$ such that $\forall i,\ 1 \leq i \leq m,\ A_i = A_m$. We define a "coalesced" constraint system E_{coal} in the following way: for each $1 \leq i \leq m$ we create the constraint $b_m \sqsubseteq H_i(b_m, \ldots, b_m, b_{m+1}, \ldots, b_n)$; for each $m < i \leq n$ we create the constraint $b_i \sqsubseteq H_i(b_m, \ldots, b_m, b_{m+1}, \ldots, b_n)$. Lemma 4 shows that coalescing preserves the maximum solution of the original system.

Lemma 4 (Equivalence with coalescing). *If E is a constraint system with maximum solution $(A_1, \ldots, A_m, \ldots, A_n)$, for any $i, j, 1 \leq i, j \leq m$ we have that $A_i = A_j$, and E_{coal} is the "coalesced" system that we derive from E, then the maximum solution of E_{coal} is $(A_m, \ldots, A_n)$.*

Proof. Both system have a (unique) maximum solution (see e.g. [34]), although the solution of the "coalesced" system has smaller cardinality, e.g., n-m+1. Now, as $(A_m, \ldots, A_m, A_{m+1}, \ldots, A_n)$ is a solution to E, by definition of E_{coal}, $(A_m, \ldots, A_n)$ is a solution to E_{coal}. Let us prove that this solution is maximum, i.e. for any solution $(B_m, \ldots, B_n)$ of E_{coal}, we have $(B_m, \ldots, B_n) \sqsubseteq (A_m, \ldots, A_n)$. By definition of E_{coal}, we have that $(B_m, \ldots, B_m, B_{m+1}, \ldots, B_n)$ is a solution to E. As $(A_1, \ldots, A_n)$ is maximum, we have $(B_m, \ldots, B_m, B_{m+1}, \ldots, B_n) \sqsubseteq (A_1, \ldots, A_n)$. So $(B_m, \ldots, B_n) \sqsubseteq (A_m, \ldots, A_n)$. □

We now prove Theorem 4, which states that there exists a direct mapping between the maximum solution of a dense constraint system associated with a SSI-form program, and the sparse system that we can derive from it, according to Definition 7.

Theorem 4 (sparse $\equiv$ dense). *Consider a program in SSI-form that gives origin to a constraint system E^{ssi}_{dense} associating with each variable v the constraints $[v]^i = [v]^i \wedge F^s_v([v_1]^s, \dots, [v_n]^s)$. Suppose that each F^s_v is a monotone function from $\mathcal{L}^n$ to $\mathcal{L}$ where $\mathcal{L}$ is of finite height. Let $(Y_v)_{v \in variables}$ be the maximum solution of the corresponding sparse constraint system.*

Then, $(X^i_v)_{(v,i) \in variables \times prog\_points}$ with $\begin{cases} X^i_v = Y_v \text{ for } i \in live(v) \\ X^i_v = \bot \text{ otherwise} \end{cases}$ is the maximum solution to E^{ssi}_{dense}.

Proof. The constraint systems E^{ssi}_{dense} and E^{ssi}_{sparse} have a maximum unique solution, because the transfer functions are monotone and $\mathcal{L}$ has finite height

The idea of the proof is to modify the constraint system E^{ssi}_{dense} into a system equivalent to E^{ssi}_{sparse}. To accomplish this transformation, we (i) replace each F^s_v by G^s_v, where G^s_v is constructed as in Definition 7; (ii) for each v, coalesce $[v]^i_{i \in live(v)}$ into $[v]$; (iii) coalesce all other constraint variables into $[v_\bot]$.

The LINK property allows us to replace F^s_v by G^s_v. Due to SPLIT-DEF, a new variable is defined at each point where information is generated, and due to VERSION there is only one live range associated with each variable. Hence, $([v]^i)_{i \in live(v)}$ is invariant. Due to INFO, we have that $([v]^i)_{i \notin live(v)}$ is bound to $\bot$. Due to Lemma 4, we know that this new constraint system has a maximum solution $(Y_v)_{v \in variables \cup \bot}$: X^i_v equals Y_v for all $i \in live(v)$, and $Y_\bot$ otherwise.

We translate each constraint $[v]^i \sqsubseteq F^s_v([v_1]^s, \dots, [v_n]^s)$, in the original system, to a constraint in the "coalesced" one in the following way:

$$\begin{cases} \text{if } i \in live(v) : \text{if } s \in defs(v) : [v] \sqsubseteq G^s_v([a], \dots, [b]) & (1) \\ \qquad\qquad\qquad\quad \text{else} \qquad\quad\; : [v] \sqsubseteq [v] & (2) \\ \text{otherwise} \qquad\qquad\qquad\qquad\;\; : [v_\bot] \sqsubseteq \bot & (3) \end{cases}$$

Case (1) follows from LINK, case (2) follows from SPLIT-DEF, and case (3) follows from INFO. By ignoring $y_\bot$ that appears only in (3), and by removing the constraints produced by (2), which are useless, we obtain E^{ssi}_{sparse}.