Symbol-Specific Sparsification of Interprocedural Distributive Environment Problems

Kadiray Karakaya
Heinz Nixdorf Institute
Paderborn University
Paderborn, Germany
kadiray.karakaya@upb.de

Eric Bodden
Heinz Nixdorf Institute
Paderborn University & Fraunhofer IEM
Paderborn, Germany
eric.bodden@upb.de

ABSTRACT

Previous work has shown that one can often greatly speed up static analysis by computing data flows not for every edge in the program's control-flow graph but instead only along definition-use chains. This yields a so-called *sparse* static analysis. Recent work on SparseDroid has shown that specifically taint analysis can be "sparsified" with extraordinary effectiveness because the taint state of one variable does not depend on those of others. This allows one to soundly omit more flow-function computations than in the general case.

In this work, we now assess whether this result carries over to the more generic setting of so-called Interprocedural Distributive Environment (IDE) problems. Opposed to taint analysis, IDE comprises distributive problems with large or even *infinitely broad* domains, such as typestate analysis or linear constant propagation. Specifically, this paper presents Sparse IDE, a framework that realizes sparsification for any static analysis that fits the IDE framework.

We implement Sparse IDE in SparseHeros, as an extension to the popular Heros IDE solver, and evaluate its performance on real-world Java libraries by comparing it to the baseline IDE algorithm. To this end, we design, implement and evaluate a linear constant propagation analysis client on top of SparseHeros. Our experiments show that, although IDE analyses can only be sparsified with respect to symbols and not (numeric) values, Sparse IDE can nonetheless yield significantly lower runtimes and often also memory consumptions compared to the original IDE.

KEYWORDS

static analysis, sparse analysis, IFDS, IDE, constant propagation

ACM Reference Format:
Kadiray Karakaya and Eric Bodden. 2024. Symbol-Specific Sparsification of Interprocedural Distributive Environment Problems. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597503.3639092

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04.
https://doi.org/10.1145/3597503.3639092

1 INTRODUCTION

Static program analysis has proven useful for diverse purposes including compiler optimization [18], program comprehension [9] and developer assistance [38]. It is now an essential part of software engineering for assuring bug-free [4], secure [22] and quality software[12]. The key strength of static program analysis is to account for all possible executions of a target program. But this imposes two often competing challenges: precision and scalability. Static analyses yield more precise results by tracking statement ordering and by distinguishing different calling contexts.

IDE (Interprocedural Distributive Environment) [30], with its extensions [2, 24, 33], is a state-of-the-art precise interprocedural static analysis framework. It covers a wide class of data-flow problems ranging from variations of classical taint analysis [16] to typestate [11, 20] and constant propagation [25] analyses. IDE represents data-flow analysis problems on an *exploded supergraph* and models data-flow facts as environments. Environments are mappings from symbols (often program variables) to domain values. The exploded supergraph is a data-flow graph induced by the inter-procedural control-flow graph (ICFG) for the whole program. Its nodes are pairs (s, d) of program statements and data-flow facts. A data-flow fact d holds at a statement s if in the exploded supergraph the corresponding node (s, d) is reachable from the start node. The edges of the exploded supergraph represent the effects of program statements on a data-flow fact. IDE computes over the exploded supergraph by tracking all data-flow facts *densely* across all program points. As previous work [1, 15, 19, 41] has shown, this approach does not scale well for large-scale real-world programs. A key observation is, however, that in practice many program statements do not affect the analysis result. Such statements thus can be safely ignored, e.g. by *sparsifying* the exploded supergraph.

Sparsification is a well-known technique for scaling data-flow analyses [13, 14, 26, 31, 35, 36] while still maintaining their precision. Sparsification approaches create sparse versions of the original CFGs of a target program by removing statements that are irrelevant to the analysis and then computing over the sparse CFGs. Recent on-demand approaches take sparsification further by utilizing the information available during the analysis. SparseBoomerang [17] accelerates demand-driven pointer analysis by computing over sparse CFGs specialized to the alias queries. SparseDroid [15] accelerates taint analysis by computing over sparse CFGs specialized to individual data-flow facts. Both approaches demonstrate sparsification on IFDS-based problems, that focus on mere symbol reachability, without considering value computation.

The IFDS (Interprocedural Finite Distributive Subset) [29] framework is the "small brother" of IDE. It reduces the data-flow analysis

problems to a pure graph reachability problem. Yet, IFDS is limited to data-flow problems with finite domains: all IFDS problems can be encoded as IDE problems, but only a subset of IDE problems can be encoded as IFDS problems [30]. As an example, consider the statement `a = a + 1`. Here, using IFDS one can encode a simple taint analysis inferring that `a` is tainted/reachable after the statement if and only if it was previously tainted/reachable. Efficient computation of `a`'s numeric value, however, requires one to *compute values* within the infinitely broad domain of integers, going beyond pure reachability. As we show, this has implications for sparsification: while the statement `a = a + 1` can be safely considered irrelevant w.r.t. `a`'s reachability, and will be disregarded in sparsification approaches for IFDS [15, 17], it *is* a relevant statement when constant propagation is considered: it changes `a`'s value. This observation is not limited to constant propagation analysis, it applies to other data-flow analysis problems that require value mappings. For instance, a sparse typestate analysis must retain statements that alter a symbol's associated state value. Based on this observation, we generalize the recent work on SparseDroid, i.e., on sparse IFDS [15]: we propose *Sparse IDE*, a symbol-specific sparsification of the IDE framework, that enables efficient sparsification, even in the presence of arbitrarily large value domains. In addition, we also show the limits of sparsification in IDE: while one can effectively sparsify with respect to symbols, such sparsification cannot be performed with respect to values.

We formalize Sparse IDE, and show how this formalization covers also IFDS data-flow analysis problems as a special case. We implement Sparse IDE in a tool SparseHeros, extending the popular Heros IDE solver [5]. We compare both implementations in terms of performance, and show that sparsification maintains correctness. To this end, we implement a linear constant propagation analysis client that uses both implementations. To validate SparseHeros's correctness, we run both on ConstantBench, a novel microbenchmark suite for integer linear constant propagation analysis. To evaluate its performance impact, we run the analysis client on real-world Java libraries using both Heros and SparseHeros. The analysis client produces the same results in both cases while terminating significantly faster when using SparseHeros.

To summarize, this paper presents the following original contributions, whose implementations are open-sourced[1]:

- A formalization of Sparse IDE and its implementation in SparseHeros on top of Heros and Soot [37],
- its correctness evaluation on the ConstantBench microbenchmark suite for linear constant propagation analysis, and
- its performance evaluation on real-world Java libraries.

The remainder of the paper is organized as follows. In Section 2, we present the background. In Section 3, we introduce Sparse IDE and in Section 4, we instantiate it on linear constant propagation analysis. In Section 5, we present the evaluation results. In Section 6, we discuss the limitations of our approach and threats to its validity. In Section 7, we discuss the related work and we conclude with Section 8.

[1]https://github.com/secure-software-engineering/SparseIDE

2 BACKGROUND

This section briefly introduces the background that our work builds on. We begin with the IFDS and IDE frameworks. Then we introduce sparse data-flow analysis and discuss why it is an effective alternative. Finally, we explain how the recent approaches sparsify further by utilizing the information available during the analysis runtime.

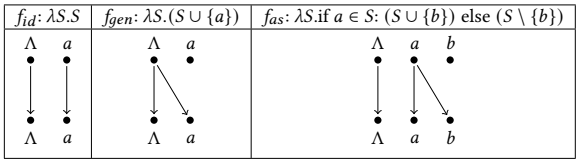


Figure 1: Flow functions (reproduced from [29]).

2.1 IFDS and IDE

IFDS [29] and IDE [30] are two frameworks for interprocedural flow- and context-sensitive data-flow analysis. IFDS represents data-flow analysis problems as graph reachability on an exploded supergraph, whose nodes are pairs of program statements and data-flow facts. The individual edges in the exploded supergraph constitute *flow functions*; they show each statement's effect on each data-flow fact's reachability. A flow function determines whether a data-flow fact is being generated, propagates to the next statement, spawns another fact, or gets killed.

Figure 1 shows how the flow functions are represented as edges in the exploded supergraph. The data-flow fact above the edge means that it holds before applying the function; the fact below means that it holds after. A special fact, Λ holds always. Facts connected to it are newly generated. The identity function, f_{id}, leaves data-flow facts unchanged. The function f_{gen} shows the case where data-flow fact a is being generated. The function f_{as} shows how the existing fact, a creates another fact, b, e.g. at an assignment, `b = a`.

IDE generalizes the IFDS framework by computing domain values that symbols map to. It does so in two phases: first it determines whether symbols are reachable, just like IFDS, and then computes their values. IDE achieves this by annotating the individual exploded supergraph edges with so-called *edge functions*, which constitute environment transformers.

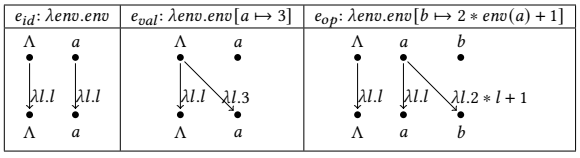


Figure 2: Edge functions (reproduced from [30]).

Figure 2 shows how the edge functions are represented. The environment transformer e_{id} keeps the values as they are. e_{val} shows the case where data-flow fact, a is mapped to a domain value, e.g. through a constant assignment, `a = 3`. e_{op} shows how the value of b is calculated depending on the value of a, e.g. through a linear

arithmetic operation, `b = 2*a + 1`. IDE can only compute *linear* equations precisely.

IFDS and IDE apply to a wide class of data-flow analysis problems. IFDS requires data-flow problems to be defined with flow functions that are distributive over the merge operator. Many reachability problems such as taint, reaching definitions, or live variables analysis fall into this category. IDE, on the other hand, also requires data-flow problems to be expressed with distributive environment transformers. IFDS suits better the problems with a binary value domain, e.g. taint analysis where the domain simply consists of two values, *tainted* or *not tainted* [3]. It has been applied to more complex domains, e.g. for typestate analysis where the domain contains arbitrary object states [23]. The drawback of IFDS is that it represents data-flow facts as symbol-value pairs, which blows up the data-flow fact space with increasing size of the domain. Because of this representation, IFDS's runtime performance depends on the value domain's size. Further, it may not terminate when the value domain is infinitely broad, e.g., in constant propagation analysis, where the domain contains all integers. IDE, on the other hand, restricts data-flow facts to static symbols and computes their (approximated) runtime values using the edge functions along the path where the symbols are reachable in the exploded supergraph. Therefore, IDE can terminate efficiently even with infinitely broad value domains—only the set of symbols must be finite.

2.2 Sparse Data-flow Analysis

Data-flow analysis techniques aim to produce precise results while remaining scalable within a reasonable time budget. Techniques that prioritize scalability often resort to sacrificing precision aspects: flow-insensitive analyses ignore control-flow ordering [40], field-insensitive analyses approximate field accesses[8], and context-insensitive analyses do not distinguish different calling contexts [21]. Sparse data-flow analyses, on the other hand, often improve a *dense* data-flow analysis' scalability while *maintaining* its precision. They sparsify a target program's control-flow graph by removing program statements that provably do not affect the analysis result. Sparsification often uses a cheaper pre-analysis stage to aid a more expensive analysis [14, 31, 35]. Recent *on-demand* sparse data-flow analyses sparsify further by exploiting the information that is only available during analysis runtime [15, 17].

2.3 Fact-Specific On-Demand Sparsification

When IFDS and IDE compute a data-flow fact's reachability, starting from the statement that generates the data-flow fact, they propagate it along all statements as long as it is not killed. At each statement, they check whether the statement is relevant for all the data-flow facts that have reached it. Figure 3 shows how the reachability is computed for an example constant-propagation analysis setting. The *fact-specific id edges* and *non-id edges* show the edges which IFDS and IDE create when propagating data-flow facts. The data-flow facts actually only need to be propagated to the *required nodes*. For instance, data-flow fact **a** only needs to propagate to the statement `b = a;` all other statements are redundant for **a**. Similarly, **b** only needs to propagate to the statement, `c = b + 1`. Based on this observation, He et al. [15] introduced the sparse IFDS algorithm in their implementation SparseDroid. Instead of propagating all the data-flow facts to the next statement, it propagates them simply to the next statement that uses the facts. Sparse IFDS keeps all *non-id edges* and replaces the *fact-specific id edges* with *sparse id edges*, effectively keeping all *required nodes* and skipping over all *redundant nodes*.

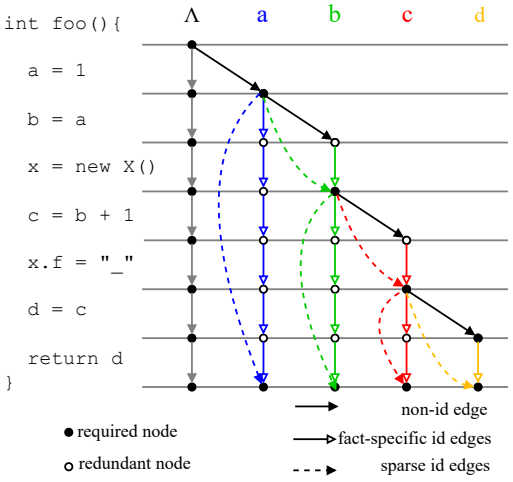


Figure 3: Original and sparse propagations after applying fact-specific on-demand sparsification.

Fact-specific on-demand sparsification allows effective propagation of the data-flow facts along the sparse CFGs specific to them, which is not limited to data-flow analysis. Recent work [17] has applied it to pointer analysis, where the variable in alias queries is treated as the initial data-flow fact and propagated along its query-specific sparse CFGs. So far, however, fact-specific on-demand sparsification has only been applied to the analysis problems that deal with fact reachability. In this work, we expand the scope of fact-specific on-demand sparsification to include the data-flow analyses that compute over an additional value domain, specifically IDE.

3 SYMBOL-SPECIFIC ON-DEMAND SPARSIFICIATION WITH SPARSE IDE

In this section, we first explain the original IDE algorithm [30] in detail. We then introduce the Sparse IDE algorithm by highlighting the modifications to the original IDE algorithm.

3.1 The Original IDE Algorithm

Sagiv et al. [30] define an IDE problem instance formally as $IP = (G^*, D, L, M)$, where

- G^* is the program supergraph (ICFG), which consists of control flow graphs (CFG), G_p of individual procedures,
- D is a finite set of program symbols,
- L is a finite-height lattice (which can be infinitely broad), and
- $M : E^* \xrightarrow{d} (Env(D, L) \rightarrow Env(D, L))$ is an assignment of distributive environment transformers to the edges of G^*.

The original IDE algorithm [30] solves such an IDE problem, IP, in two phases. In Phase I, it creates the jump functions that show the reachability of each $d \in D$, by assuming that their initial

mappings to L are always $\lambda l.\top$. In Phase II, it computes each d's actual value mapping to L by evaluating the edge functions defined in M.

According to Sagiv et al. [30], the total cost of the IDE algorithm is bounded by $O(|E||D|^3)$, which is the cost of Phase I. Since D is the set of symbols, it should not change if correctness is preserved. We, therefore, apply our sparsification approach in Phase I, where the jump functions are created by reducing E, the set of edges. Phase II is oblivious to how the jump functions are created—it automatically benefits from the sparsification of Phase I.

Figure 4 shows the algorithm for Phase I. Each procedure p's CFG, G_p consists of a *start* node s_p, an *exit* node e_p, and *normal* (non-call) nodes m or n. Procedure calls are represented with two nodes: the *call-site* node c denotes the point right before the procedure call, and the *return-site* node r denotes the point right after. Program symbols, e.g. variables, access paths, etc., are denoted with $d', d \in D \cup \{\Lambda\}$ including the special symbol Λ. Λ is required for generating new symbols at arbitrary program points.

Initialization. In lines 2–5, jump and summary functions are initialized. Jump functions, denoted by $JumpFn$, correspond to the *same-level realizable paths* (SLRPs) from the start node s_p of a procedure p to a node m in p. Summary functions, denoted by $SummaryFn$, summarize the effect of a procedure call through same-level realizable paths from the call-site c to return-site r. In line 3, $JumpFn(\langle s_p, d'\rangle \rightarrow \langle m, d\rangle) = \lambda l.\top$ states that the jump function from the node $\langle s_p, d'\rangle$ to each $\langle m, d\rangle$ is initialized to $\lambda l.\top$. In line 5, $SummaryFn(\langle c, d'\rangle \rightarrow \langle r, d\rangle) = \lambda l.\top$ states that the summary function from each call-site node $\langle c, d'\rangle$ to its corresponding return-site $\langle r, d\rangle$ is initialized to $\lambda l.\top$. Line 6 initializes the *PathWorkList* to $\{\langle s_{main}, \Lambda\rangle \rightarrow \langle s_{main}, \Lambda\rangle\}$ representing a self-loop edge on the start node of the *main* procedure whose jump function is the identity function, *id*. The jump function from the start node s_p until the current statement n is denoted with f.

Call nodes. Lines 12-19 handle the case where n is a call-site node in p, calling a procedure q. In line 14, the self-loop edge on the start node of the callee procedure q is initialized with *id*. In line 17, the edge from s_p the corresponding return-site r is computed by composing the f, the jump function until n and the edge function from n to r. In line 19, the edge from s_p the corresponding return-site r is computed by composing f and f_3, the corresponding summary function when it is not mapping to $\top$.

Exit nodes. Lines 20-30 handle the case where n is the exit node of p. Edges from each call-site node c to the start node s_p (shown with f_4) and from the exit node, e_p to each caller's return-site r (shown with f_5) must be computed. In line 25, a new summary function f' is computed by composing f_5, f, and f_4 and merging the existing summary function for the same c and r. When it is a new summary, a new jump function is computed from the caller procedure's start node s_q to the node return-site node r by composing the f' with the existing jump function f_3 from s_q to call-site node c.

Normal nodes. Lines 31-33 handle the case where n is a non-call or intraprocedural node. Edges from the start node s_p to each node m, which is the statement that appears directly after n in procedure p, are computed by composing the edges from s_p to n (shown with f) and the edges from n to m.

```
1  Function ForwardComputeJumpFunctionsSLRPs():
2    for ⟨s_p, d'⟩, ⟨m, d⟩ s.t. m occurs in proc. p and d', d ∈ D ∪ {Λ} do
3      JumpFn(⟨s_p, d'⟩ → ⟨m, d⟩) = λl.⊤
4    for corresponding call-return pairs (c, r) and d', d ∈ D ∪ {Λ} do
5      SummaryFn(⟨c, d'⟩ → ⟨r, d⟩) = λl.⊤
6    PathWorkList := {⟨s_main, Λ⟩ → ⟨s_main, Λ⟩}
7    JumpFn(⟨s_main, Λ⟩ → ⟨s_main, Λ⟩) := id
8    while PathWorkList ≠ ∅ do
9      Select and remove an item ⟨s_p, d1⟩ → ⟨n, d2⟩ from PathWorkList
10     let f = JumpFn(⟨s_p, d1⟩ → ⟨n, d2⟩)
11     switch (n) do
12       case n is a call node in p, calling a procedure q do
13         for d3 s.t. ⟨n, d2⟩ → ⟨s_q, d3⟩ ∈ E# do
14           Propagate(⟨s_q, d3⟩ → ⟨s_q, d3⟩, id)
15         let r be the return-site node that corresponds to n
16         for d3 s.t. e = ⟨n, d2⟩ → ⟨r, d3⟩ ∈ E# do
17           Propagate(⟨s_p, d1⟩ → ⟨r, d3⟩, EdgeFn(e) ∘ f)
18         for d3 s.t. f3 = SummaryFn(⟨n, d2⟩ → ⟨r, d3⟩) ≠ λl.⊤ do
19           Propagate(⟨s_p, d1⟩ → ⟨r, d3⟩, f3 ∘ f)
20       case n is the exit node of p do
21         for call node c that calls p with corresponding return-site node r do
22           for d4, d5 s.t. ⟨c, d4⟩ → ⟨s_p, d1⟩ ∈ E# and ⟨e_p, d2⟩ → ⟨r, d5⟩ ∈ E# do
23             let f4 = EdgeFn(⟨c, d4⟩ → ⟨s_p, d1⟩) and
24             f5 = EdgeFn(⟨e_p, d2⟩ → ⟨r, d5⟩) and
25             f' = (f5 ∘ f ∘ f4) ⊓ SummaryFn(⟨c, d4⟩ → ⟨r, d5⟩)
26             if f' ≠ SummaryFn(⟨c, d4⟩ → ⟨r, d5⟩) then
27               SummaryFn(⟨c, d4⟩ → ⟨r, d5⟩) := f'
28               let s_q be the start node of c's procedure
29               for d3 s.t. f3 = JumpFn(⟨s_q, d3⟩ → ⟨c, d4⟩) ≠ λl.⊤ do
30                 Propagate(⟨s_q, d3⟩ → ⟨r, d5⟩, f' ∘ f3)
31       case n is an intraprocedural node in p do
32         for ⟨m, d3⟩ s.t. ⟨n, d2⟩ → ⟨m, d3⟩ ∈ E# do
33           Propagate(⟨s_p, d1⟩ → ⟨m, d3⟩,
34                     EdgeFn(⟨n, d2⟩ → ⟨m, d3⟩) ∘ f)
35
36 Function Propagate(e, f):
37   let f' = f ⊓ JumpFn(e)
38   if f' ≠ JumpFn(e) then
39     JumpFn(e) := f'
40     Insert e into PathWorkList
```

Figure 4: The original IDE algorithm for Phase I (reproduced from [30]).

3.2 The Sparse IDE Algorithm

In the original IDE algorithm, each symbol $d \in D \cup \{\Lambda\}$ at a statement n is propagated to its direct successor statement m. As also pointed out in previous work [15], this behavior is desired when n is a call and exit node. For these nodes, the reachability of each d in different contexts is left to the data-flow function definition. *call-flow functions* propagate each d into the context of the callee procedure. *return-flow functions* propagate each d back to the context of the caller procedure. *call-to-return-flow functions* propagate each d from before a procedure is called to after the procedure is called. However, when n is a non-call node, each d can safely be propagated to d's next use statement.

Figure 5 shows the modifications for the Sparse IDE algorithm for Phase I. We replace line 17 from the original IDE algorithm with lines 17-19 in the Sparse IDE algorithm. Instead of propagating d_3 to the direct return site node r, we obtain r' which is the next use statement of d_3 in its *symbol-specific* sparse control flow

```
1  Function ForwardComputeSparseJumpFunctionsSLRPs():
2      ...
8      while PathWorkList ≠ ∅ do
9          Select and remove an item ⟨s_p, d1⟩ → ⟨n, d2⟩ from PathWorkList
10         let f = JumpFn(⟨s_p, d1⟩ → ⟨n, d2⟩)
11         switch (n) do
12             case n is a call node in p, calling a procedure q do
13                 ...
15                 let r be the return-site node that corresponds to n
16                 for d3 s.t. e = ⟨n, d2⟩ → ⟨r, d3⟩ ∈ E# do
17                     let r' = NextUse(p, d3, r)
18                     Propagate(⟨s_p, d1⟩ → ⟨r', d3⟩,
19                               EdgeFn(⟨n, d2⟩ → ⟨r, d3⟩) ∘ f)
20             ...
31             case n is an intraprocedural node in p do
32                 for ⟨m, d3⟩ s.t. ⟨n, d2⟩ → ⟨m, d3⟩ ∈ E# do
33                     let m' = NextUse(p, d3, n)
34                     Propagate(⟨s_p, d1⟩ → ⟨m', d3⟩,
35                               EdgeFn(⟨n, d2⟩ → ⟨m, d3⟩) ∘ f)
36
41 Function NextUse(p, d, n):
42     let G_{p,d} be the sparse CFG of d in procedure p
43     let C be the sparse CFG cache with (p, d) typed keys and G_{p,d} as values
44     if G_{p,d} ∉ C then
45         construct G_{p,d} and add to C
46     return the next statement after n from G_{p,d}
```

Figure 5: Modifications for Sparse IDE algorithm for Phase I (mirrors the design from [15]).

graph. Similarly, we replace line 33 with lines 33-35, to propagate d_3 to its next use statement m' its sparse control flow graph. Our sparsification approach mirrors that of sparse IFDS algorithm [15], however, since we generalize it to IDE, we also account for edge function composition.

3.3 Sparse IFDS Revisited

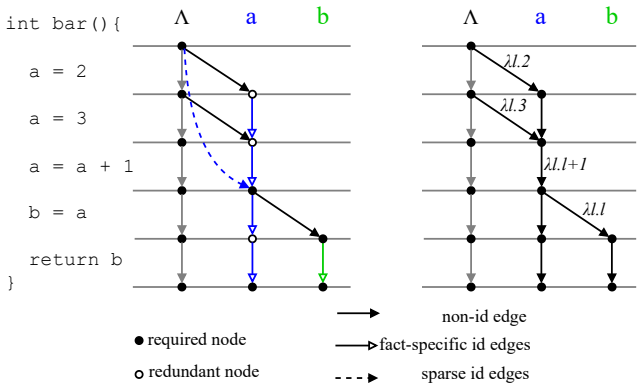
As shown in Figure 3, a statement can behave as *identity function*, meaning it does not affect any data-flow fact, $d \in D$. However, as shown by He et al. [15], many statements only affect a few data-flow facts, often even just a single fact. Their flow functions can be considered *fact-specific identity functions* for the facts that they do not affect. Sparse IFDS defines fact-specific identity functions as follows [15]:

Given a symbol, $d \in D$ and a flow function, $f \in 2^D \to 2^D$, f is a *d-specific identity function* if the following conditions hold:

$$\forall X \in 2^D : d \in X \Rightarrow d \in f(X) \tag{1.1}$$

$$\forall X \in 2^{D \setminus \{d\}} : f(X) \setminus \{d\} = f(X \cup \{d\}) \setminus \{d\} \tag{1.2}$$

Condition 1.1 states that d is not affected by other facts when applying f, and 1.2 states that d does not affect the other facts when applying f. However, these conditions only apply to symbols from D and ignore mappings from D to the value domain L, and, if applied to IDE problems, one would wrongly treat such flow functions that are annotated with non-identity edge functions as d-specific identity functions as well.



(a) Sparse IFDS (b) Sparse IDE

Figure 6: Comparison of the Sparsification Approaches of Sparse IFDS and Sparse IDE

Figure 6 shows two important cases where sparse IFDS would sparsify incorrectly. First, reassignments: `a = 3` reassigns `a`, but sparse IFDS recognizes that a already exists (is "tainted"), and therefore it treats this statement as a-specific identity. Second, value updates: `a = a + 1` updates a's value, but sparse IFDS has no notion of values, therefore, from its perspective, this statement is "identity" as well. Sparse IDE, on the other hand, is aware of the effects on the value domain and retains both statements.

3.4 Fact-Specific Identity Transformers

To generalize fact-specific sparsification to the IDE framework, we define symbol-specific identity transformers that take into account the environments that map the symbols from domain D to the values from domain L. Given a symbol $d \in D$ and a value $l \in L$, $env = [d \mapsto l]$ is an environment env mapping from d to l, i.e., $env(d) = l$. Then env is an element of the set of environments $Env(D, L)$. An environment transformer, $t \in Env(D, L) \to Env(D, L)$ is a *d-specific identity transformer*, denoted by $t \equiv t^d$, if the following holds:

First, the transformer t keeps all d-specific mappings intact:

$$\text{given } d \in D : \forall env \in Env(D, L) : \\ env(d) = t(env(d)) \tag{2.1}$$

Second, for all other mappings, t produces identical results no matter whether or not d-specific mappings are present:

$$\text{given } d \in D : \forall env \in Env(D, L).\ \forall d' \in D \setminus \{d\}.\ \forall l \in L : \\ t(env(d')) = t(env[d \mapsto l](d')) \tag{2.2}$$

We test the edge functions from Figure 2 on these conditions. e_{id} is an a-specific identity transformer ($e_{id} \equiv e^a_{id}$), because applying $\lambda env.env$ does not change a's previous mapping. e_{val} is not an a-specific identity transformer ($e_{val} \not\equiv e^a_{val}$), because applying $\lambda env.env[a \mapsto 3]$ changes a's previous mapping. e_{op} is also not an a-specific identity transformer ($e_{op} \not\equiv e^a_{op}$) because applying $\lambda env.env[b \mapsto 2 * env(a) + 1]$ changes another value's mapping

(for b) depending on what a maps to, and because it changes b's value e_{op} is not a b-specific identity transformer either ($e_{op} \not\equiv e^b_{op}$). Note that, importantly, a transformer can only be considered a d-identity transformer if the above restrictions hold *irrespective* of any concrete $l \in L$ that might be associated with b: (2.2) quantifies over all $l \in L$. This is necessary because IDE produces procedure summaries that must be sound with respect to all l, and thus their creation must not be made dependent on l. In other words, IDE can support symbol-specific but not value-specific sparsification!

3.5 Determining symbol-specific identity

When propagating fact d, we consider only those statements as irrelevant statements for d that fulfil conditions (2.1) and (2.2). But since these conditions are value-agnostic—they quantify over all $l \in L$, this allows one to determine *ahead of time* the statements whose environment transformers adhere to both conditions, structurally. First, by Condition 2.1, a statement's corresponding environment transformer t is *not* a d-specific identify transformer if t affects d's value mapping in any way, i.e., $t = \lambda env.env[\boldsymbol{d} \mapsto \_]$. Second, by Condition 2.2, t is *not* a d-specific identity transformer either, if t uses d's value mapping $env(d)$ to compute another fact's value, i.e. $t = \lambda env.env[\_ \mapsto \ldots env(\boldsymbol{d}) \ldots]$.

Naturally, sparsification effectiveness is closely tied to the analysis-specific environment-transformer definitions. The environment transformer for the statement `a = a + 1` is $t \equiv t^a$ for taint analysis, where $t = \lambda env.env$. For constant propagation analysis, however, $t \not\equiv t^a$, where $t = \lambda env.env[env(a) + 1]$.

Sparse IDE strictly generalizes Sparse IFDS as presented in SparseDroid. One can easily define sparse IFDS as an instantiation of sparse IDE by restricting the value domain L to $\{\bot, \top\}$, where symbols that map to $\bot$ are considered reachable. In this setting, our definitions (2.1) and (2.2) become equivalent to (1.1) and (1.2).

4 APPLICATION TO LINEAR CONSTANT PROPAGATION

As Sagiv, Reps and Horwitz explain in their seminal work [30], constant propagation analysis is the perfect problem setting where IDE outperforms IFDS [29]. This is not only because the problem's lattice is larger than the binary domain, but also it is infinitely broad where IFDS cannot terminate. We are, therefore, motivated to apply the Sparse IDE framework to linear constant propagation analysis. Heros, and thus SparseHeros, are generic tools and they are independent of the target language and their intermediate representations (IRs). In this work, we use Soot [37] static program analysis framework for Java and its intermediate representation Jimple. Therefore, in the following, we explain our implementation based on the Jimple IR.

4.1 Analysis Definition

Linear constant propagation analysis handles the linear expressions that generate a new data-flow fact by using just a single other fact, e.g. `a = b` or `a = 2*b + 1`. Full constant propagation analysis involves statements such as `a = b + c`. Such a statement's flow function is not distributive; it cannot be precisely computed within the IDE framework. Our linear constant propagation analysis implementation handles the assignment statements shown in Table 1.

IR. The IR always ensures binary operation (*binop*) representation by reducing more complex operations to binary operations. For instance, `a = 2*b + 1` would be reduced to `s1 = 2 * b` and `a = s1 + 1`. The IR also reduces longer access paths to multiple assignments with a single access path (*n=1*). For instance, a statement such as `a = b.f1.f2` would be reduced to `s1 = b.f1`, `s2 = s1.f2`, and `a = s2`. The same reduction applies to procedure invocations as well.

Flow functions. We *generate* a symbol when it is assigned with a *constant*. As discussed, we handle the binary operations in the linear form. We distinguish between the assignments that require alias handling and the ones that do not. The assignments such as *local, field load, static field load,* and *array load*, overwrite the local variable, a, on their left-hand side and therefore do not need to know a's aliases. The assignments such as *field store, static field store,* and *array store*, on the other hand, require handling the aliases of the base variables or the array references. To handle aliasing we use the Boomerang [34] demand-driven pointer analysis framework. When necessary, we query the aliases of the base variables and add them to the set of propagated symbols. Note that in Table 1, the alias sets contain the query variable as well. The IDE framework requires three types of flow functions to model the effects of invoke statements. The *call* flow function propagates the symbol for the actual parameter to the context of the callee procedure, by mapping it to the procedure's corresponding formal parameter. The *return* flow function propagates the symbol for the returned variable to the context of the caller procedure, by mapping it to the symbol on the left-hand side of the invoke expression. The *call-to-return* flow function propagates the symbols that are not passed to the context of the callee procedure, to the next statement after the invoke statement.

Edge functions. For most statements, the edge functions map the target symbol to the value of the source symbol, acting as *identity transformers*. The *constant* and *binop* statements are the only exceptions. The constant statement maps the target symbol, a to the given constant value, $Const$. The binop statement maps the target symbol, a to a new value. The value is computed by simulating the operation $\odot$ using the source symbol's value, $env(b)$ and the constant operand, $Const$. Edge functions must be composed and reduced to a simple value mapping when computing the actual values. Given $f_1, f_2 \in Env(D, L)$ and f_1 appears before f_2 as an edge in the exploded supergraph, we compose the edge functions as follows:

$$f_2 \circ f_1 := \begin{cases} f_2 & \text{if } f_1 = \lambda env.env \\ f_1 & \text{if } f_2 = \lambda env.env \\ f_2 & \text{if } f_2 = \lambda env.env[a \mapsto Const] \\ f_2(f_1) & \text{if } f_2 = \lambda env.env[a \mapsto env(b) \mathbin{\hat{\odot}} Const] \end{cases}$$

If an edge function is the identity transformer, we always apply the other function by the first two conditions. We always apply the subsequent edge function if it is a constant assignment, by the third condition. If the subsequent edge is a binop, we compute its value immediately in place by applying the preceding edge first, as suggested in previous work [5].

Table 1: Statements for Linear Constant Propagation Analysis with Corresponding IRs and Flow/Edge Functions.

Statement	IR	Flow Function	Edge Function
constant	$a \leftarrow Const$	$\lambda S.\{S \cup \{a\}\}$	$\lambda env.env[a \mapsto Const]$
binop	$a \leftarrow b \odot Const$	$\lambda S.\begin{cases} S \cup \{a\} & \text{if } b \in S \\ S \setminus \{a\} \end{cases}$	$\lambda env.env[a \mapsto env(b) \hat{\odot} Const]$
local	$a \leftarrow b$	$\lambda S.\begin{cases} S \cup \{a\} & \text{if } b \in S \\ S \setminus \{a\} \end{cases}$	$\lambda env.env[a \mapsto env(b)]$
field load	$a \leftarrow b.f$	$\lambda S.\begin{cases} S \cup \{a\} & \text{if } b.f \in S \\ S \setminus \{a\} \end{cases}$	$\lambda env.env[a \mapsto env(b.f)]$
field store	$a.f \leftarrow b$	$\lambda S.\begin{cases} S \cup \{p.f \mid p \in aliases(a)\} & \text{if } b \in S \\ S \setminus \{p.f \mid p \in aliases(a)\} \end{cases}$	$\lambda env.env[p.f \mapsto env(b)]$
static field load	$a \leftarrow T.f$	$\lambda S.\begin{cases} S \cup \{a\} & \text{if } T.f \in S \\ S \setminus \{a\} \end{cases}$	$\lambda env.env[a \mapsto env(T.f)]$
static field store	$T.f \leftarrow b$	$\lambda S.\begin{cases} S \cup \{p.f \mid p \in aliases(T)\} & \text{if } b \in S \\ S \setminus \{p.f \mid p \in aliases(T)\} \end{cases}$	$\lambda env.env[p.f \mapsto env(b)]$
array load	$a \leftarrow A[i]$	$\lambda S.\begin{cases} S \cup \{a\} & \text{if } A[i] \in S \\ S \setminus \{a\} \end{cases}$	$\lambda env.env[a \mapsto env(A[i])]$
array store	$A[i] \leftarrow b$	$\lambda S.\begin{cases} S \cup \{p[i] \mid p \in aliases(A)\} & \text{if } b \in S \\ S \setminus \{p[i] \mid p \in aliases(A)\} \end{cases}$	$\lambda env.env[p[i] \mapsto env(b)]$
call	$r \leftarrow b.m(a_i)$	$\lambda S.\begin{cases} S \cup \{p_i\} & \text{if } a_i \in S \wedge a_i \mapsto p_i \text{ in } m \\ S \setminus \{p_i\} \end{cases}$	$\lambda env.env[p_i \mapsto env(a_i)]$
return	$r \leftarrow b.m(a_i)$	$\lambda S.\begin{cases} S \cup \{r\} & \text{if } r' \in S \wedge m \text{ returns } r' \\ S \setminus \{r\} \end{cases}$	$\lambda env.env[r \mapsto env(r')]$
call-to-return	$r \leftarrow b.m(a_i)$	$\lambda S.\begin{cases} S \setminus \{a_i\} & \text{if } a_i \in S \wedge a_i \mapsto p_i \text{ in } m \\ S \end{cases}$	$\lambda env.env$

Lattice. We perform the linear constant propagation on integers. Therefore the lattice is $\mathbb{Z}^{\top}_{\bot}$. Given $l_1, l_2 \in \mathbb{Z}^{\top}_{\bot}$, we define the meet operator as follows:

$$l_1 \sqcap l_2 = \begin{cases} l_1 & \text{if } l_2 = \top \\ l_2 & \text{if } l_1 = \top \\ \bot & \text{if } l_1 = \bot \ \vee \ l_2 = \bot \\ \top & \text{if } l_1 = \top \ \wedge \ l_2 = \top \end{cases}$$

If a value is $\top$, the meet operator yields the other value by the first two conditions. If either of the values is $\bot$, the meet yields $\bot$, and if both of the values are $\top$ it yields $\top$ by the third and fourth conditions respectively.

4.2 Sparsification for Constant Propagation

Our sparsification approach has much in common with the one proposed by He et al. [15], though modifications were necessary. We build the sparse control flow graphs (CFGs) by ignoring symbol-specific identity functions. Given a procedure, p, G_p is its original *dense* CFG. We build sparse CFGs specific to each symbol, d in p, denoted as $G_{p,d}$ and propagate d across its own sparse CFG. As shown with the IR in Table 1, d can be a local, an instance field or static field, or an array access. $G_{p,d}$ is constructed by determining whether each statement's corresponding flow function in G_p is a d-specific identity function.

As a major modification, and most importantly, we account for a statement's effect on the value domain. In addition to determining whether each statement's corresponding flow function is a *d-specific identity function*, we determine whether its edge function is a *d-specific identity transformer* with the assumptions explained in Section 3.3. Further, we propagate the *tautological* fact, Λ, (sparsely) to the statements that can generate new data-flow facts, e.g. $a \leftarrow Const$. Otherwise, it is impossible to generate new facts at arbitrary program points. Finally, we soundly retain all branching statements to keep the original CFGs' control flow as it is.

5 EVALUATION

We next explain the research questions that guide our evaluation and its experimental setup, and then we discuss the evaluation results. Sparse data-flow analyses promise extensive performance improvements, while still maintaining the precision of their non-sparse counterparts. Therefore, first, we compare the sparse analysis results against the non-sparse analysis results. Second, we measure whether the sparse analysis produces the promised performance benefits. Third, we investigate the factors contributing to the performance impact. Therefore, we focus on the following research questions:

- RQ1: Does Sparse IDE produce the same results as the original IDE?
- RQ2: How does the sparsification impact the performance in terms of runtime and memory?
- RQ3: To what extent does the number of propagations correlate with the performance impact?

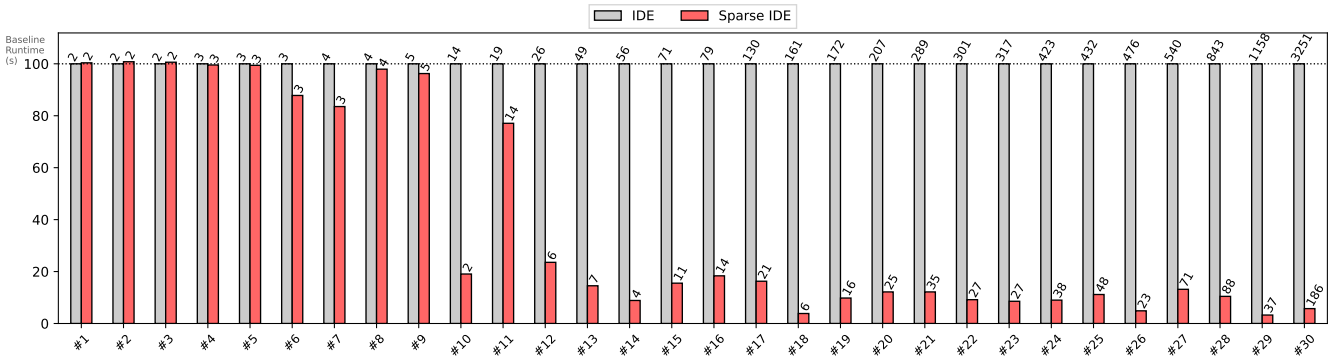


Figure 7: Relative runtime of Sparse IDE compared to the baseline original IDE in %, annotated with exact runtimes in seconds, sorted by original IDE's runtime

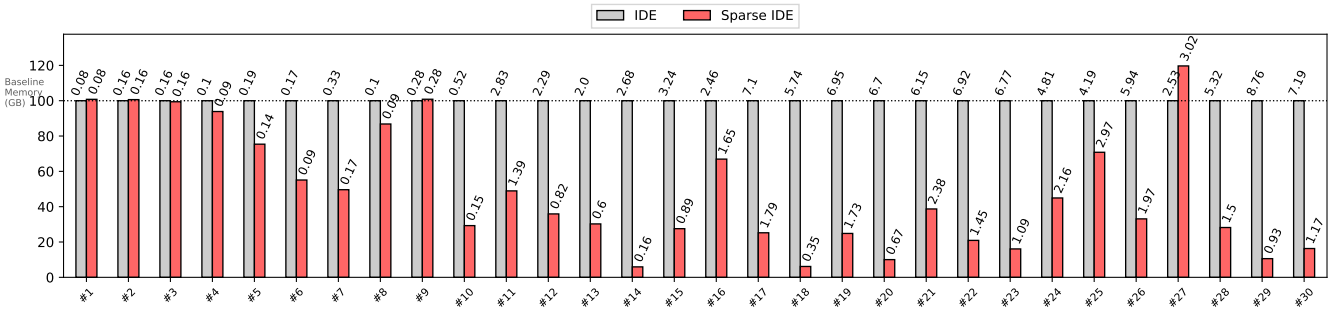


Figure 8: Memory consumption of Sparse IDE compared to the baseline original IDE in %, annotated with exact memory consumptions in GB, using the same sorting as Figure 7

5.1 Experimental Setup

We implement the proposed approach in SparseHeros, by extending the open source Heros IDE solver's latest version, at the time of writing (`e7e4a85`) [32]. Using SparseHeros and the Soot static analysis framework [37], we implement a linear constant propagation analysis. To handle aliasing, we integrate our client analysis with the Boomerang [34] demand-driven pointer analysis, using its latest version (1179227) [7]. Heros, and thus SparseHeros, support multi-threading, yet, because Boomerang is single-threaded, our client analysis uses a single-thread. Therefore, our evaluation results present single-thread performance.

As benchmark subjects we use:

- **ConstantBench**: A benchmark suite for constant propagation analysis targeting Java, did not previously exist. We, therefore, created ConstantBench as a micro-benchmark suite for integer linear constant propagation analysis. We run both Heros and SparseHeros on this benchmark suite and compare the analysis results that they produce.
- **Real-world Libraries**: We include real-world Java libraries to investigate the performance of our approach under the workload of large-scale and complex programs. As opposed to applications, libraries do not have a specific *entry* method. We follow the *closed package assumption* [27] for analyzing library code, and treat public methods of the libraries as entry methods. We consider a method as an entry method if it adheres to the following entry method selection criteria:
  - **c1:** The method is a public instance method that is not abstract, native or a constructor,
  - **c2:** The method contains an integer assignment statement.

  We selected the most downloaded (>5000) Java libraries from the maven repository [28]. We discarded the libraries that do not contain any entry methods according to the selection criteria, and the ones that caused an error in the underlying static analysis tool, Soot [37]. In the end, we retained 30 libraries.
- **Replication Package**: We set up a replication package, available at https://zenodo.org/records/10498325

We have performed the evaluations on an Intel i7 Quad-Core at 2,3 GHz with 32GB memory. We configured the JVM with 25GB maximum heap size (`-Xmx25g`) and 1GB stack size (`-Xss1g`).

5.2 RQ1: Does Sparse IDE produce the same results as the original IDE?

ConstantBench consists of 40 target programs with various program properties and sensitivity-testing edge cases, as listed in Table 2. *Assignment* cases test possible flow and edge functions, as well as flow sensitivity. *Branching* and *Loops* cases test the meet operation. *Field sensitivity* cases test field sensitivity and aliasing scenarios. *Context sensitivity* cases test various calling contexts. *Array* cases

Table 2: ConstantBench Test Cases

Assignment	**Field Sensitivity**
Constant	LoadConstant
ConstantBinop	StoreConstant
LocalBinop	StoreViaAlias
LocalMultipleBinop	StoreBinop
Overwrite	FieldToField
Increment	StoreBinopViaAlias
Operators	StoreLocalViaAlias
AssignmentChain	**Context Sensitivity**
Static	Id
Branching	Increment
SameValueMergedAndUsed	Add
SameValueMergedNotUsed	Nested
SameValueMergedAndUsedInBinop	AssignFieldInCallee
DiffValuesMergedAndUsed	AssignStaticInCallee
DiffValuesMergedNotUsed	**Array**
DiffValuesMergedAndUsedInBinop	LoadConstant
Loops	StoreConstant
ForLoopFixedBound	ArrayToArray
ForLoopUnkownBound	AliasedArrays
WhileTrue	LargeIndex
WhileUnknown	**Non-Linear**
NestedLoops	Binop
	HashCode

test array handling and *NonLinear* cases test analysis' behavior under unanticipated non-linear operations. The results validate the correctness of Sparse IDE by showing that SparseHeros produces the same outputs as the non-sparse Heros.

5.3 RQ2: How does the sparsification impact the performance in terms of runtime and memory?

Figure 7 shows the relative analysis runtime spent by Sparse IDE in comparison to the runtime of the baseline original IDE algorithm. We sorted the results for each library by the time spent by the original IDE algorithm. Note that we keep the same sorting for the rest of the paper. This sorting highlights the fact that our Sparse IDE approach pays off better for the cases where the original IDE's runtime is relatively larger. Sparse IDE, compared to the original IDE algorithm, performs up to 30.7x faster. We measure the mean speedup as 7.9x, and the median speedup as 6.7x. The concrete measurements are presented in Table 3. Results show that, in terms of runtime, Sparse IDE outperforms the original IDE in each run, except for the libraries #1-#3 (jcl-over-slf4j, slf4j-api, lombok), which have the shortest analysis time. In each run, Sparse CFG construction overhead is lower than 1% of the Sparse IDE total analysis runtime, which is substantially smaller than the achieved speedups.

Figure 8 shows the relative memory consumption of Sparse IDE in comparison to the memory consumption of the original IDE algorithm. We have measured up to 94% reduction in memory consumption in the best case, and up to a 19% increase in the worst. The Sparse IDE algorithm, compared to the original IDE,

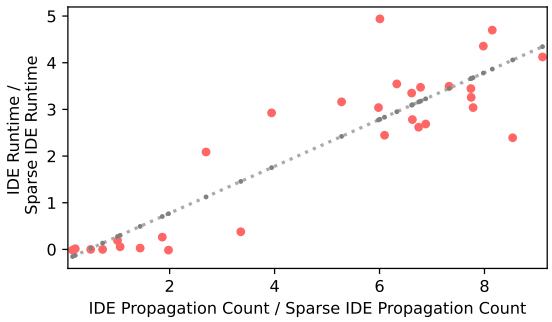


Figure 9: Ratio of data-flow fact propagations and corresponding speedup ratios, in log scale

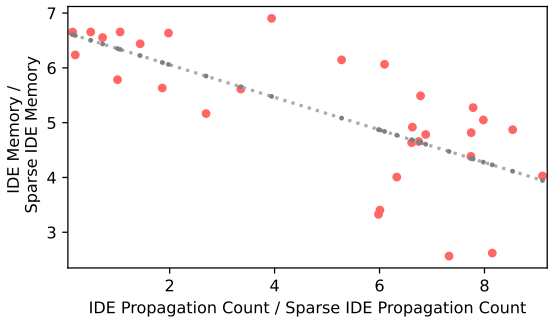


Figure 10: Ratio of data-flow fact propagations and corresponding memory consumption ratios, in log scale

associates data-flow facts with fewer statements, therefore, we anticipated memory improvements. On the other hand, because we cache sparse CFGs ($G_{d,p}$) per each symbol and procedure pair (d, p), for some input programs memory consumption increases. However, as shown in Figure 8, these cases are limited to a few outliers. Moreover, the mean and median impacts on memory consumption are 51% and 63% reduction, respectively.

We statistically assess the significance of the Sparse IDE algorithm's impact on runtime and memory improvements. According to Wilcoxon signed-rank test [39] at 0.05 significance level, Sparse IDE significantly improves both the runtime (p = 6.1e−08) and memory consumption (p = 5.7e−07) of the original IDE algorithm.

5.4 RQ3: To what extent does the number of propagations correlate with the performance impact?

The essence of the Sparse IDE approach is that, compared to the original IDE algorithm, it propagates data-flow facts to fewer statements. We investigate to what extent this contributes to improving the scalability of the original IDE algorithm. Figure 9, shows how the ratio of data-flow fact propagations in IDE and Sparse IDE correlate with the ratio of runtime speedups. We observe that reducing the number of propagations is an effective approach to improving IDE's scalability in terms of runtime. Similarly, Figure 10 correlates the same with the ratio of memory consumptions in IDE and Sparse IDE. We observe a comparable trend but not to the same degree.

Table 3: Performance of Sparse IDE compared to the baseline original IDE algorithm

#	Library	Version	#Entry Methods	Runtime (s)			Memory (GB)			#Propagations			Sparse CFG		
				IDE	SP	IDE/SP	IDE	SP	SP/IDE (%)	IDE	SP	IDE/SP	Count	Const. (ms)	%Runtime
1	jcl-over-slf4j	2.0.7	1	2	2	1.00	0.08	0.08	100.78	48	34	1.41	2	0	0.01
2	slf4j-api	2.0.7	7	2	2	0.99	0.16	0.16	100.62	104	94	1.11	13	0	0.00
3	lombok	1.18.26	5	2	2	0.99	0.16	0.16	99.40	894	227	3.94	13	0	0.02
4	commons-logging	1.2	14	3	3	1.00	0.10	0.09	93.87	1,509	917	1.65	41	0	0.00
5	junit-jupiter-api	5.9.2	10	3	3	1.01	0.19	0.14	75.39	182	158	1.15	20	0	0.00
6	jackson-annotations	2.14.2	79	3	3	1.14	0.17	0.09	55.10	13,115	6,511	2.01	190	0	0.00
7	maven-plugin-api	3.9.1	13	4	3	1.20	0.33	0.17	49.61	17,353	4,780	3.63	294	4	0.14
8	junit-jupiter-engine	5.9.2	23	4	4	1.02	0.10	0.09	86.81	3,204	1,181	2.71	105	0	0.02
9	osgi.core	8.0.0	124	5	5	1.04	0.28	0.28	100.83	58,675	28,247	2.08	664	7	0.15
10	jakarta.servlet-api	6.0.0	12	14	2	5.25	0.52	0.15	29.28	126,656	341	371.43	33	0	0.00
11	commons-io	2.11.0	178	19	14	1.30	2.83	1.39	48.94	156,595	15,290	10.24	1,279	116	0.78
12	commons-codec	1.15	77	26	6	4.25	2.29	0.82	35.90	652,560	100,866	6.47	532	13	0.21
13	json	20230227	33	49	7	6.88	2.00	0.60	30.24	1,071,045	10,846	98.75	407	0	0.00
14	logback-classic	1.4.7	93	56	4	11.28	2.68	0.16	5.92	1,286,543	8,027	160.28	372	12	0.24
15	logback-core	1.4.7	218	71	11	6.44	3.24	0.89	27.55	1,739,303	14,767	117.78	925	0	0.00
16	gson	2.10.1	147	79	14	5.45	2.46	1.65	66.93	2,009,909	29,391	68.39	1,586	54	0.37
17	commons-lang3	3.12.0	318	130	21	6.14	7.10	1.79	25.22	3,418,491	31,856	107.31	1,144	0	0.00
18	commons-beanutils	1.9.4	109	161	6	25.97	5.74	0.35	6.15	5,855,012	20,640	283.67	648	2	0.04
19	mockito-core	5.3.1	235	172	16	10.20	6.95	1.73	24.85	5,025,407	51,374	97.82	1,663	119	0.71
20	junit-jupiter-params	5.9.2	293	207	25	8.22	6.70	0.67	10.03	6,266,620	99,285	63.12	1,506	109	0.43
21	assertj-core	3.24.2	334	289	35	8.22	6.15	2.38	38.71	10,033,236	45,563	220.21	2,418	37	0.11
22	commons-collections4	4.4	620	301	27	10.90	6.92	1.45	20.91	9,140,963	42,741	213.87	1,796	1	0.01
23	testng	7.7.1	246	317	27	11.68	6.77	1.09	16.08	9,329,214	116,084	80.37	2,910	15	0.06
24	joda-time	2.12.5	375	423	38	11.11	4.81	2.16	44.93	15,151,487	137,705	110.03	3,227	69	0.18
25	guice	5.1.0	336	432	48	8.95	4.19	2.97	70.80	15,141,525	390,634	38.76	3,918	58	0.12
26	hamcrest-all	1.3	290	476	23	20.48	5.94	1.97	33.10	17,953,051	71,200	252.15	1,105	28	0.12
27	log4j-core	2.20.0	512	540	71	7.60	2.53	3.02	119.69	18,746,154	1,218,580	15.38	4,666	64	0.09
28	jackson-databind	2.14.2	844	843	88	9.57	5.32	1.50	28.20	35,842,682	166,906	214.75	7,884	5	0.01
29	okhttp	4.10.0	717	1,158	37	30.69	8.76	0.93	10.58	37,431,312	581,852	64.33	5,928	69	0.18
30	guava-31.1	jre	1,332	3,251	186	17.43	7.19	1.17	16.31	131,993,565	239,589	550.92	12,200	4	0.00

Given these findings, in the future, one could investigate the potential synergies between our approach and recent approaches that improve the scalability, in particular, in terms of memory [1, 19].

6 LIMITATIONS AND THREATS TO VALIDITY

By definition, Sparse IDE can solve the same data-flow problems as the original IDE framework [30]. It requires data-flow analysis problems to be expressible as distributive environment problems. Many popular static analyses, such as taint analysis for vulnerability detection [3] or typestate analysis for API misuse detection [10], are expressible as distributive environment problems. Just like other fact-specific sparsification approaches [15, 17], Sparse IDE also exploits analysis domain knowledge. Domain-specific analysis semantics must be correctly encoded with flow and edge function definitions within the IDE framework.

Sparse IDE should theoretically lead to a similar performance impact on other data-flow analysis problems where IDE is applicable. For instance, when performing a typestate analysis, Sparse IDE would safely omit the statements that have no impact on the tracked state. However, due to space constraints, we were not able to empirically show whether our evaluation results carry over to other analysis problems.

The reported evaluation results might depend on the selected set of Java libraries, and entry-method selection criteria. Nevertheless, for real-world library selection, we followed the systematic procedure described in Section 5.1. To account for variations in runtime and memory measurements, we conducted three runs and presented the average across these runs.

A direct comparison to SparseDroid [15] was not possible for many reasons. It extends an existing taint analysis client (FlowDroid [3]) that has a basic integrated alias analysis, whereas our analysis client utilizes a sophisticated external demand-driven pointer analysis [34]. Moreover, SparseDroid's implementation is not publicly available, and most importantly, IFDS may not terminate when the value domain is infinitely broad.

7 RELATED WORK

The IFDS [29] and IDE [30] frameworks enabled precise interprocedural data-flow analyses that are flow- and context-sensitive. Previous works have extended these frameworks with diverse goals. Naeem et al. [24] proposed four extensions to the IFDS framework, to improve its scalability and precision under certain practical analysis conditions. Heros [5] introduced a Java-based generic IFDS and IDE solver. Reviser [2] proposed an algorithm to adapt IFDS and IDE to incremental program updates. CleanDroid [1] introduced a technique for reducing the memory footprint of IFDS-based data-flow analyses. DiskDroid [19] applied a disk-assisted computing approach for improving the scalability of IFDS-based taint analysis.

Sparsification has been applied to improve the scalability of static analyses. Choi et al. [6] introduced sparse data-flow evaluation graphs based on SSA (static-single assignment). Oh et al. [26] presented an abstract interpretation-based framework for designing generic sparse analyses, which guarantees to preserve the precision of the non-sparse analysis through data dependencies. Pinpoint [31], SVF [35] and SFS [14] utilize cheaper pre-analyses to sparsify pointer analyses. Recent on-demand sparsification approaches exploit the data-flow facts that become available during the analysis runtime for further sparsification. SparseBoomerang [17] exploits the variables in alias queries during demand-driven pointer analysis, to create query-specific sparse CFGs. The sparse IFDS algorithm [15] exploits data-flow facts to create fact-specific sparse CFGs and propagate each fact on its own sparse CFG. In this work, we present the more generic Sparse IDE algorithm that efficiently solves not just IFDS-based reachability problems, but also IDE problems that require value computation.

8 CONCLUSION AND FUTURE WORK

In this work, we presented the Sparse IDE framework as a scalable alternative to the original IDE framework. Sparse IDE is the first fact-specific sparsification approach that allows for computations on infinitely broad domains. The essence of Sparse IDE is creating symbol-specific sparse control flow graphs on-demand, and propagating data-flow facts sparsely through these graphs. Sparse IDE produces equally precise results as the original IDE, while significantly improving its scalability. We also explicitly discuss the limits of sparsification for IDE: while symbol-specific sparsification is possible and useful, one cannot sparsify with respect to the (typically numeric and infinite) value domain.

In the future, we plan to apply the Sparse IDE framework to other data-flow analysis problems and investigate problem-specific requirements for building sparse CFGs. We also plan to combine Sparse IDE with other scalability-improving techniques that are orthogonal to our sparsification approach.

ACKNOWLEDGMENTS

We gratefully acknowledge the support of Martin Mory and Marcus Hüwe in this work. We thank Martin for the enlightening discussions and for the encouragement to conclude this work. We thank Marcus for sharing his expertise on the formal notation.

REFERENCES

[1] Steven Arzt. 2021. Sustainable Solving: Reducing The Memory Footprint of IFDS-Based Data Flow Analyses Using Intelligent Garbage Collection. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 1098–1110.

[2] Steven Arzt and Eric Bodden. 2014. Reviser: efficiently updating IDE-/IFDS-based data-flow analyses in response to incremental program changes. In *Proceedings of the 36th International Conference on Software Engineering*. 288–298.

[3] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 259–269. https://doi.org/10.1145/2594291.2594299

[4] Nathaniel Ayewah, William Pugh, David Hovemeyer, J. David Morgenthaler, and John Penix. 2008. Using Static Analysis to Find Bugs. *IEEE Software* 25, 5 (2008), 22–29. https://doi.org/10.1109/MS.2008.130

[5] Eric Bodden. 2012. Inter-Procedural Data-Flow Analysis with IFDS/IDE and Soot. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis* (Beijing, China) *(SOAP '12)*. Association for Computing Machinery, New York, NY, USA, 3–8. https://doi.org/10.1145/2259051.2259052

[6] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. 1991. Automatic construction of sparse data flow evaluation graphs. In *Proceedings of the 18th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 55–66.

[7] CodeShield. [n. d.]. CodeShield-Security/SPDS: Efficient and Precise Pointer-Tracking Data-Flow Framework. https://github.com/CodeShield-Security/SPDS. (Accessed on 03/30/2023).

[8] Alain Deutsch. 1994. Interprocedural May-Alias Analysis for Pointers: Beyond k-Limiting. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation* (Orlando, Florida, USA) *(PLDI '94)*. Association for Computing Machinery, New York, NY, USA, 230–241. https://doi.org/10.1145/178243.178263

[9] T. Eisenbarth, R. Koschke, and D. Simon. 2001. Aiding program comprehension by static and dynamic feature analysis. In *Proceedings IEEE International Conference on Software Maintenance. ICSM 2001.* 602–611. https://doi.org/10.1109/ICSM.2001.972777

[10] Michael Emmi, Liana Hadarean, Ranjit Jhala, Lee Pike, Nicolás Rosner, Martin Schäf, Aritra Sengupta, and Willem Visser. 2021. RAPID: checking API usage for the cloud in the cloud. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1416–1426.

[11] Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. 2008. Effective Typestate Verification in the Presence of Aliasing. *ACM Trans. Softw. Eng. Methodol.* 17, 2, Article 9 (may 2008), 34 pages. https://doi.org/10.1145/1348250.1348255

[12] Jeffrey S Foster, Michael W Hicks, and William Pugh. 2007. Improving software quality with static analysis. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. 83–84.

[13] Ben Hardekopf and Calvin Lin. 2009. Semi-Sparse Flow-Sensitive Pointer Analysis. *SIGPLAN Not.* 44, 1 (jan 2009), 226–238. https://doi.org/10.1145/1594834.1480911

[14] Ben Hardekopf and Calvin Lin. 2011. Flow-sensitive pointer analysis for millions of lines of code. In *International Symposium on Code Generation and Optimization (CGO 2011)*. IEEE, 289–298.

[15] Dongjie He, Haofeng Li, Lei Wang, Haining Meng, Hengjie Zheng, Jie Liu, Shuangwei Hu, Lian Li, and Jingling Xue. 2019. Performance-Boosting Sparsification of the IFDS Algorithm with Applications to Taint Analysis. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 267–279. https://doi.org/10.1109/ASE.2019.00034

[16] N. Jovanovic, C. Kruegel, and E. Kirda. 2006. Pixy: a static analysis tool for detecting Web application vulnerabilities. In *2006 IEEE Symposium on Security and Privacy (S&P'06)*. 6 pp.–263. https://doi.org/10.1109/SP.2006.29

[17] Kadiray Karakaya and Eric Bodden. 2023. Two Sparsification Strategies for Accelerating Demand-Driven Pointer Analysis. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*.

[18] Gary A. Kildall. 1973. A Unified Approach to Global Program Optimization *(POPL '73)*. Association for Computing Machinery, New York, NY, USA, 194–206. https://doi.org/10.1145/512927.512945

[19] Haofeng Li, Haining Meng, Hengjie Zheng, Liqing Cao, Jie Lu, Lian Li, and Lin Gao. 2021. Scaling up the IFDS algorithm with efficient disk-assisted computing. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 236–247.

[20] Tuo Li, Jia-Ju Bai, Yulei Sui, and Shi-Min Hu. 2022. Path-sensitive and alias-aware typestate analysis for detecting OS bugs. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 859–872.

[21] Yue Li, Tian Tan, Anders Møller, and Yannis Smaragdakis. 2020. A Principled Approach to Selective Context Sensitivity for Pointer Analysis. *ACM Trans. Program. Lang. Syst.* 42, 2, Article 10 (may 2020), 40 pages. https://doi.org/10.1145/3381915

[22] V Benjamin Livshits and Monica S Lam. 2005. Finding Security Vulnerabilities in Java Applications with Static Analysis.. In *USENIX security symposium*, Vol. 14. 18–18.

[23] Nomair A. Naeem and Ondrej Lhotak. 2008. Typestate-like Analysis of Multiple Interacting Objects. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications* (Nashville, TN, USA) *(OOPSLA '08)*. Association for Computing Machinery, New York, NY, USA, 347–366. https://doi.org/10.1145/1449764.1449792

[24] Nomair A Naeem, Ondřej Lhoták, and Jonathan Rodriguez. 2010. Practical extensions to the IFDS algorithm. In *Compiler Construction: 19th International Conference, CC 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings 19*. Springer, 124–144.

[25] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. 2015. Composite constant propagation: Application to android inter-component communication analysis. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 77–88.

[26] Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. 2012. Design and Implementation of Sparse Global Analyses for C-like Languages. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (Beijing, China) *(PLDI '12)*. Association for Computing Machinery, New York, NY, USA, 229–238. https://doi.org/10.1145/2254064.2254092

[27] Michael Reif, Michael Eichberg, Ben Hermann, Johannes Lerch, and Mira Mezini. 2016. Call Graph Construction for Java Libraries. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 474–486. https://doi.org/10.1145/2950290.2950312

[28] Maven Repository. [n. d.]. Maven Repository: Search/Browse/Explore. https://mvnrepository.com/. (Accessed on 03/30/2023).

[29] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 49–61.

[30] Mooly Sagiv, Thomas Reps, and Susan Horwitz. 1996. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Science* 167, 1-2 (1996), 131–170.

[31] Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. 2018. Pinpoint: Fast and Precise Sparse Value Flow Analysis for Million Lines of Code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 693–706.

https://doi.org/10.1145/3192366.3192418
[32] soot oss. [n. d.]. soot-oss/heros: IFDS/IDE Solver for Soot and other frameworks. https://github.com/soot-oss/heros. (Accessed on 03/30/2023).
[33] Johannes Späth, Karim Ali, and Eric Bodden. 2017. Ide al: Efficient and precise alias-aware dataflow analysis. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–27.
[34] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. 2016. Boomerang: Demand-driven flow-and context-sensitive pointer analysis for java. In *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.
[35] Yulei Sui and Jingling Xue. 2016. SVF: Interprocedural Static Value-Flow Analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction* (Barcelona, Spain) *(CC 2016)*. Association for Computing Machinery, New York, NY, USA, 265–266. https://doi.org/10.1145/2892208.2892235
[36] Yulei Sui, Sen Ye, Jingling Xue, and Pen-Chung Yew. 2011. SPAS: Scalable Path-Sensitive Pointer Analysis on Full-Sparse SSA. In *Programming Languages and Systems*, Hongseok Yang (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 155–171.
[37] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: A Java bytecode optimization framework. In *CASCON First Decade High Impact Papers*. 214–224.
[38] Carmine Vassallo, Sebastian Proksch, Timothy Zemp, and Harald C Gall. 2020. Every build you break: developer-oriented assistance for build failure resolution. *Empirical Software Engineering* 25 (2020), 2218–2257.
[39] Frank Wilcoxon. 1992. *Individual comparisons by ranking methods*. Springer.
[40] Sen Ye, Yulei Sui, and Jingling Xue. 2014. Region-Based Selective Flow-Sensitive Pointer Analysis. In *Static Analysis*, Markus Müller-Olm and Helmut Seidl (Eds.). Springer International Publishing, Cham, 319–336.
[41] Xiaodong Yu, Fengguo Wei, Xinming Ou, Michela Becchi, Tekin Bicer, and Danfeng Yao. 2020. GPU-Based Static Data-Flow Analysis for Fast and Scalable Android App Vetting. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 274–284. https://doi.org/10.1109/IPDPS47924.2020.00037