IncIDFA: An Efficient and Generic Algorithm for Incremental Iterative Dataflow Analysis

AMAN NOUGRAHIYA, IIT Madras, India
V. KRISHNA NANDIVADA, IIT Madras, India

Iterative dataflow analyses (IDFAs) are important static analyses employed by tools like compilers for enabling program optimizations, comprehension, verification, and more. During compilation of a program, optimizations/transformations can render existing dataflow solutions stale, jeopardizing the optimality and correctness of subsequent compiler passes. Exhaustively recomputing these solutions can be costly. Since most program changes impact only small portions of the flowgraph, several incrementalization approaches have been proposed for various subclasses of IDFAs. However, these approaches face one or more of these limitations: (i) loss of precision compared to exhaustive analysis, (ii) inability to handle arbitrary lattices and dataflow functions, and (iii) lacking fully automated incrementalization of the IDFA. As a result, mainstream compilers lack frameworks for generating precise incremental versions of arbitrary IDFAs, leaving analysis writers to create *ad hoc* algorithms for incrementalization – an often cumbersome and error-prone task.

To tackle these challenges, we introduce IncIDFA, a novel algorithm that delivers precise and efficient incremental variants of any monotone IDFA. IncIDFA utilizes a two-pass approach to maintain precision. Unlike prior works, IncIDFA avoids resetting the dataflow solutions to least informative values when dealing with strongly-connected regions and arbitrary program changes. We formally prove the precision guarantees of IncIDFA for arbitrary dataflow problems and program changes. IncIDFA has been implemented in the IMOP compiler framework for parallel OpenMP C programs. To showcase its generality, we have instantiated IncIDFA to ten specific dataflow analyses, without requiring any additional code for incrementalization. We present an evaluation of IncIDFA on a real-world set of optimization passes, across two different architectures. As compared to exhaustive recomputation, IncIDFA resulted in a speedup of up to 11× (geomean 2.6×) in incremental-update time, and improvement of up to 46% (geomean 15.1%) in the total compilation time.

CCS Concepts: • **Software and its engineering → Automated static analysis**; *Compilers.*

Additional Key Words and Phrases: incremental compilation, static analysis

ACM Reference Format:
Aman Nougrahiya and V. Krishna Nandivada. 2025. IncIDFA: An Efficient and Generic Algorithm for Incremental Iterative Dataflow Analysis. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 102 (April 2025), 53 pages. https://doi.org/10.1145/3720436

1 Introduction

Iterative dataflow analyses cover substantial number of static analyses that compute meaningful information (for example, reaching definitions, live variables, points-to information, and so on) by propagating the information across any flow graph of the program (such as control-flow graphs and call-graphs) until a fixed-point is reached. A general lattice-theoretic framework for IDFA was first given by Kildall [70], followed by a plethora of other improvements [26, 28, 37, 49, 50, 53, 59, 62, 63, 66–69, 89, 96, 97, 105, 130, 146]. Given the generality and ease of use of the IDFA framework,

Authors' Contact Information: Aman Nougrahiya, Department of CSE, IIT Madras, Chennai, India, amannoug@cse.iitm.ac.in; V. Krishna Nandivada, Department of CSE, IIT Madras, Chennai, India, nvk@iitm.ac.in.

ACM 2475-1421/2025/4-ART102
https://doi.org/10.1145/3720436

it is fairly common for compiler writers to implement custom instantiations of IDFA with arbitrary lattices to meet their specific requirements.

These analyses results are used by many optimizations invoked during compilation, which in turn lead to many programs changes. These changes may render existing dataflow solutions stale, jeopardizing the optimality, or even correctness, of the downstream compiler passes. Exhaustively recomputing the dataflow solutions in response to such program changes by rerunning the analyses from scratch can be cost prohibitive. Similar challenges arise throughout a program's lifecycle, such as during development and debugging in an IDE or CI pipelines, or even during execution by managed runtimes (such as during hot-code replacement in JIT compilers).

Since most program changes typically affect only a small region of the program or a small part of the dataflow solution, prior research has proposed different incremental analysis algorithms for *specific* IDFA problems [17, 24, 25, 45, 48, 72, 75, 76, 79, 82, 101, 122, 127, 144, 149, 150, 152]. An incremental analysis takes the old dataflow solution, the set of program changes performed, and the updated program, to obtain the new dataflow solution without rerunning the dataflow analysis on the updated program from scratch. Clearly, designing custom incremental algorithms for each new dataflow problem can be tedious and error-prone.

To address these challenges, for nearly four decades, researchers have explored *generic* incrementalization approaches for different subclasses of IDFA [7, 27, 32, 42, 65, 67, 83, 90, 98, 138, 139, 151]. Unfortunately, despite extensive research, current incrementalization approaches[1] suffer from one or more of the following limitations:

- *From-scratch precision.* While incremental approaches like those by Cooper and Kennedy [27], Ghoddsi [42], Nichols et al. [90], Van der Plas et al. [139] (and its improvement by Van der Plas et al. [138]) offer faster convergence to a fixed-point, they do not guarantee the precision achieved by complete invalidation and exhaustive recomputation of the dataflow solution from scratch. This loss of precision can hinder the optimality of various client optimization passes. A critical analysis of the precision guarantees of contemporary incrementalization algorithms has been provided by Burke and Ryder [16].
- *Arbitrary lattices and dataflow functions.* Many prior works are limited by the types of lattices or dataflow functions they can handle. Examples include works of Keables et al. [65] and Zadeck [151] (a subset of set-union problems); Ghoddsi [42] and Khedker [67] (bit-vector analyses/gen-kill/separable analyses); Pollock and Soffa [98], Marlowe and Ryder [83] (distributive dataflow problems); Arzt and Bodden [7], and Do et al. [32] (IFDS/IDE [89, 105] problems). Further, many restrictively assume that changes in the amount of information at a basic block (such as updated GEN and KILL sets [87]) can be determined by inspecting only that block, without requiring the dataflow results from the other blocks. This limits their applicability to various widely-used and important dataflow problems, such as points-to analysis, constant-propagation analysis, and so on.
- *Automated incrementalization of new dataflow problems.* Most existing approaches that preserve precision, such as those by Marlowe and Ryder [83], Pollock and Soffa [98], Zadeck [151], rely on detailed case analysis of the type of program changes performed, and their implications for specific dataflow analyses under consideration (unlike Khedker [67]). For new or arbitrary dataflow problems, such change-impact relationship depends on the semantics of the abstract domain, and must be manually provided by the analysis writers, making automation difficult, and incrementalization error-prone. The lack of formal correctness proofs in many approaches [7, 27, 32, 42, 65, 83, 90, 98, 138, 139, 151] casts further doubt on their applicability to new dataflow problems.

[1] In this section, we focus solely on incrementalization schemes for *iterative* methods for *full* dataflow analysis. Other approaches, such as logic-programming-based, elimination-based, or demand-driven methods are discussed in Section 7.

```
/* Commit Summary (b323f4f)
 * [LoopRotate] Fix DomTree update logic for unreachable nodes. Fix PR33701.
 * LoopRotate manually updates the DoomTree by iterating over all predecessors of a basic
     block & computing the Nearest Common Dominator. When a predecessor happens to be
     unreachable, `DT.findNearestCommonDominator` returns nullptr.
 * This patch teaches LoopRotate to handle this case and fixes PR33701. */
...
// Brute force incremental dominator tree update. Call findNearestCommonDominator on all
// CFG predecessors of each child of the original header.
do {
  Changed = false;
  for (unsigned I = 0, E = HeaderChildren.size(); I != E; ++I) { ...
-     pred_iterator PI = pred_begin(BB); BasicBlock *NearestDom = *PI;
-     for (pred_iterator PE = pred_end(BB); PI != PE; ++PI)
-       NearestDom = DT->findNearestCommonDominator(NearestDom, *PI);
+     BasicBlock *NearestDom = nullptr;
+     for (BasicBlock *Pred : predecessors(BB)) {
+       // Consider only reachable basic blocks.
+       if (!DT->getNode(Pred)) continue;
+       ...
+       NearestDom = DT->findNearestCommonDominator(NearestDom, Pred);
+     } ...
  }
  // If the dominator changed, this may have an effect on other predecessors,
  // continue until we reach a fixpoint.
} while (Changed);
```

(a) LoopRotation.cpp:469-515 (LLVM), showing a bug-fixing commit; multi-line comments are the commit-message.

```
// We do not update postdominators, so free them unconditionally.
free_dominance_info(CDI_POST_DOMINATORS);
// If we removed paths in the CFG, then we need to update dominators as well.
// I haven't investigated the possibility of incrementally updating dominators.
if (cfg_altered) free_dominance_info(CDI_DOMINATORS);
```

(b) tree-ssa-dce.c:1693-1711 (GCC), where the comments show the need for a generic incremental IDFA.

Fig. 1. Example snippets from LLVM and GCC, illustrating the need for a generic incremental iterative dataflow algorithm. The comments are written by the pass writers.

Given these limitations, it is not surprising that mainstream compilers, such as LLVM [74, 78], GCC [40, 128], Soot [125, 137], Rose [100], JIT compilers (OpenJ9 [57], HotSpot [95], V8 [43]), and so on, do not implement a framework that automatically generates incremental versions of arbitrary IDFAs without sacrificing precision guarantees. This leaves analysis writers with two options: design their own ad hoc incremental variants for various standard and custom analyses, which is cumbersome and error-prone, or resort to exhaustive recomputation of dataflow solutions, which is cost-prohibitive. Fig. 1 illustrates two such scenarios from real-world compilers, LLVM and GCC. Snippet 1a demonstrates the error-prone nature of ad hoc incremental update implementations – the commit summary, and the lines marked with "-" and "+", indicate relevant bug-fixes in an LLVM commit [77]. Snippet 1b reveals how developers resort to invalidating (and later, exhaustively recomputing) the dataflow solutions due to the lack of an incremental update mechanism in GCC.

Proposed Solution. To address these challenges, we introduce IncIDFA, a novel algorithm that provides a precise and efficient incremental version of any monotone IDFA. IncIDFA ensures from-scratch precision using a methodical two-pass approach on each strongly-connected component (SCC) of the program in topological order. The first pass, termed *initialization-pass*, efficiently updates the dataflow solution to a state that guarantees no loss of precision. Unlike prior two-pass approaches [42, 98, 151], IncIDFA avoids resetting the dataflow solutions to least informative values when dealing with strongly-connected regions and arbitrary program changes. The second pass, *stabilization*, derives the final fixed-point solution by applying the standard iterative method. To prove that the final solution is the maximum fixed-point (MFP) solution, we also present a formal description of IncIDFA, proving its soundness and precision guarantees for any monotone IDFA with arbitrary lattices and dataflow functions. IncIDFA is fully automated, requiring only the standard definition of the exhaustive IDFA.

It is important to clarify that this manuscript does not aim to propose a compiler design that identifies (i) when to trigger the incremental update of a specific dataflow solution, or (ii) which program changes to handle during such triggers. Instead, our goal is to present a generic incremental update algorithm that can be instantiated to various specific incremental IDFAs, which can then be invoked by any compiler framework at appropriate compilation points with required arguments.

Contributions.

- We propose IncIDFA, a generic algorithm for fully automated incrementalization of any monotone unidirectional IDFA with arbitrary abstract domain, handling arbitrary program changes. IncIDFA guarantees that the obtained solution matches the maximum fixed-point (MFP) solution.
- We provide a formal description of IncIDFA, along with proofs for its soundness and precision guarantees. The proofs ascertain the applicability of IncIDFA to any monotone unidirectional IDFA with arbitrary dataflow functions and finite-height lattices.
- We introduce a novel heuristic, termed *accessed-keys heuristic,* to further reduce the frequency of transfer-function applications. This heuristic proposes an option to avoid invoking transfer functions, when the relevant portions of the incoming flow-map remain unchanged since after the last processing of a node (a common case).
- We have implemented IncIDFA in the self-stabilizing [93] IMOP compiler framework [92] for OpenMP C programs. We have also created ten concrete instantiations of IncIDFA, without requiring any additional incrementalization-specific code for them, thus reaffirming its generality.
- We evaluated IncIDFA using a set of real-world optimization passes (BarrElim; see [93]) on 13 benchmark programs from three standard OpenMP C suites (NPB-OMP 3.0 [140], SPEC OMP 2001 [8], and Sequoia [120]). Across two architectures, our evaluation shows up to 11× speedup (geomean 2.6×) in IDFA-update time, and up to to 46% improvement (geomean 15.1%) in the total compilation time, when using IncIDFA as compared to exhaustive rerun of the dataflow analyses.

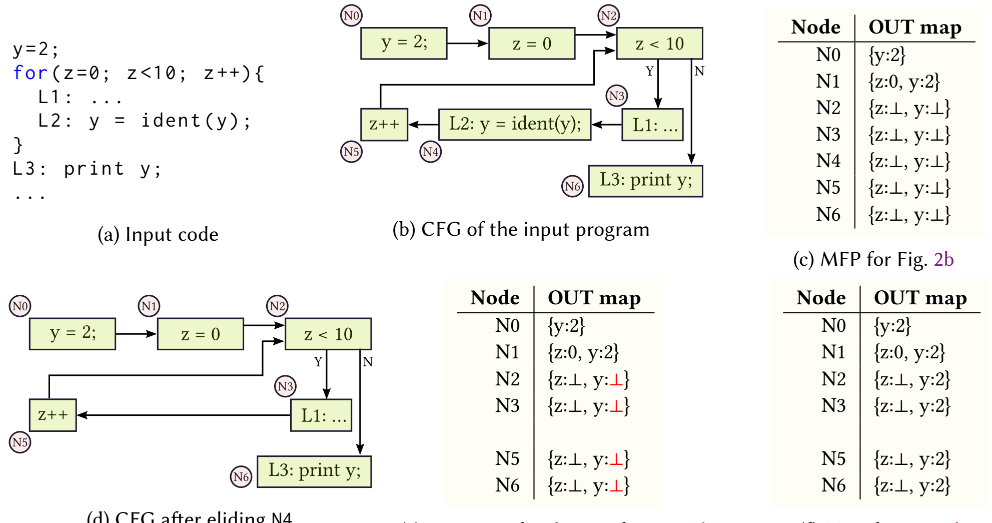
2 Background And Terminology

In this section, we informally describe a few key concepts used in the paper.

A **flowgraph** statically captures the runtime behaviour of a program, with nodes representing units of computation (such as basic blocks in control-flow graphs, or functions in call-graphs), and edges representing the flow of control. For simplicity, we assume that each node is unique, contains a single statement or predicate, and is immutable. Note that immutability is not a limiting assumption: mutations can be modeled by replacing the old node with a modified one.

Iterative Dataflow Analysis (IDFA) computes meaningful information (called *flow-facts*) at each node of the flowgraph by propagating data across the flowgraph until a fixed-point is reached. Each node n has two flow-facts: $\mathtt{IN}_n$ and $\mathtt{OUT}_n$. The dataflow equations for IDFA are standard [70, 87]: $\mathtt{IN}_n := \bigsqcap_{p \in \mathrm{pred}(n)} \mathtt{OUT}_p$, and $\mathtt{OUT}_n := \tau_n(\mathtt{IN}_n)$, where $\bigsqcap$ is the meet operation defined by the associated lattice, and τ_n is the *transfer-function* of n, reflecting its impact on $\mathtt{IN}_n$. Informally, the least fixed-point solution of these equations, starting with the uninitialized value ($\top$) for all nodes, is called the **maximum fixed-point (MFP)** solution. Any solution that, when applying the equations until fixed-point, converges to the MFP solution, is regarded as a **safe-initial-estimate**. Without loss of generality, we focus solely on *forward*-dataflow analyses. All results in this manuscript apply analogously to backward-dataflow analyses as well.

Program changes obtained upon transforming a flowgraph P to P' can be defined using four sets: (i) nodes added, (ii) nodes removed, (iii) edges added, and (iv) edges removed. For the given program changes, the set of **seed** nodes contains those that may be directly impacted (that is,



```
y=2;
for(z=0; z<10; z++){
    L1: ...
    L2: y = ident(y);
}
L3: print y;
...
```

(a) Input code

(b) CFG of the input program

Node	OUT map
N0	{y:2}
N1	{z:0, y:2}
N2	{z:⊥, y:⊥}
N3	{z:⊥, y:⊥}
N4	{z:⊥, y:⊥}
N5	{z:⊥, y:⊥}
N6	{z:⊥, y:⊥}

(c) MFP for Fig. 2b

(d) CFG after eliding N4

Node	OUT map
N0	{y:2}
N1	{z:0, y:2}
N2	{z:⊥, y:⊥}
N3	{z:⊥, y:⊥}
N5	{z:⊥, y:⊥}
N6	{z:⊥, y:⊥}

(e) Imprecise fixed-point for Fig. 2d

Node	OUT map
N0	{y:2}
N1	{z:0, y:2}
N2	{z:⊥, y:2}
N3	{z:⊥, y:2}
N5	{z:⊥, y:2}
N6	{z:⊥, y:2}

(f) MFP for Fig. 2d

Fig. 2. An example to illustrate the imprecision issues with restarting-iterations. Fig. 2a shows an input program, Fig. 2b shows its control-flow graph (CFG), and Fig. 2d shows the CFG after eliding N4. Fig. 2c and Fig. 2f show the constant-propagation MFP flow maps at various nodes, for Fig. 2b and Fig. 2d, respectively. Fig. 2e shows the imprecise fixed-point flowmaps obtained using restarting iterations for Fig. 2d.

require recalculation) due to the changes. For forward IDFAs, this set includes (i) newly-added nodes, and (ii) nodes whose predecessors-set, pred, has changed. We use the term **IDFA-stabilization** to refer to updating the stale dataflow solution to conform to the modified program P'. Let v be the fixed-point solution for the original program P. To compute the new fixed-point v' for P', there are naturally two options: (i) complete invalidation and re-computation (**CompIDFA**) of fixed-point solution v' by applying the exhaustive IDFA on P' from scratch, or (ii) incremental update, where v' is computed from v, by taking the program changes into consideration. Incremental updates generally involve fewer computations, and are thus expected to be faster.

Restarting iterations is one of the simplest approaches to incremental updates [27, 42, 90, 138, 139], where the standard IDFA is restarted from the seed nodes to reach a fixed-point solution. This technique produces sound but often imprecise solutions [16], as shown in Section 3.1. While Ryder et al. [109] outlined sufficient conditions under which this technique obtains precise solutions, the conditions are impractical since they require prior knowledge of the new solution [83].

3 Intuition for IncIDFA

In this section, we first explore the causes of imprecision in restarting iterations (Section 3.1). We then gradually build the intuition for the proposed IncIDFA algorithm in Sections 3.2, 3.3, and 3.4, before giving a formal description of IncIDFA in Section 4.

3.1 Ghost Mappings: Imprecision Issues with Restarting Iterations

Though the *restarting-iterations* approach (see Section 2) for incrementalizing an IDFA is typically faster than CompIDFA (see Section 2), it may produce imprecise results in practice. We now discuss these imprecision issues by demonstrating the problem of *ghost mappings*.

Example 3.1 (Ghost Mappings). Consider the example program shown in Fig. 2a, along with its control-flow graph (CFG) in Fig. 2b. Let us assume that the code at L1 does not write to the variable y. The compiler derives the MFP solution at each program node for intra-procedural flow-sensitive constant-propagation analysis [87] as shown in Fig. 2c. Note that at L2 the intra-procedural analysis assigns ⊥ to y. Later, if the compiler determines that the function 'ident' is an identity function, then a compiler pass may identify the assignment at L2 as redundant and elide it, resulting in the removal of node N4 from the CFG. This changes the immediate predecessor of node N5 to node N3, making the set of *seed-nodes* due to this transformation {N5}.

As discussed in Section 2, the restarting-iterations approach resumes the standard IDFA algorithm with the seed-nodes ({N5}), keeping the existing OUT maps of all nodes unchanged. When processing N5, its new IN is computed from the OUT of its predecessor (N3). However, since the OUT of N3 still holds the stale value for y (⊥), the new IN(y) and OUT(y) for N5 do not change during this pass of restarting-iterations, resulting in an imprecise fixed-point solution that is not the MFP solution (shown in Fig. 2e and Fig. 2f, respectively). Consequently, the compiler cannot replace the variable y with the literal constant 2 at L3. □

Notably, there is no node in Fig. 2d that assigns ⊥ to y, yet this mapping appears in the OUT flowmap of all nodes within the cycle N2 ⟶ N3 ⟶ N5 ⟶ N2, as well as in node N6. We term such mappings in the affected flowmaps as *ghost mappings*.

Let us understand why ghost mappings exist in the fixed-point solution obtained using the restarting-iterations approach: While processing a node (say n) with this approach, the new IN flowmap for n is obtained by taking the meet of the OUT flowmaps of all its predecessors. If a predecessor p of node n is reachable from any of the seed-nodes, the OUT flowmap of p might not correspond to a safe-initial-estimate (see Section 2) until p itself has been processed at least once. Hence, if node n is processed before node p, the new IN flowmap for n may become an unsafe-initial-estimate due to the stale OUT flowmap of node p. When node n and p are part of a cycle in the CFG, the order of processing of nodes does not resolve the issue, allowing the unsafe-initial-estimate to propagate within the cycle, leading to ghost mappings. This is one of the main causes of imprecision[2] in the fixed-point solution obtained using the restarting-iterations approach.

3.2 Ensuring Precision with Naïve Two-Pass Approach

Observation A. *The presence of cycles in the control flow graph (CFG) can cause unsafe-initial-estimates to propagate during incremental updates, leading to the occurrences of ghost mappings.*

Proposal A: Naïve `InitRestart` Approach. This naïve two-pass approach is based on the observation that the IN and OUT flowmaps of only those nodes may get updated in the final MFP solution that are reachable from any of the seed-nodes. In the first pass, the OUT flowmaps of all nodes reachable from the seed-nodes are reset to map all domain-elements to the initial value (⊤). In the second pass, the restarting-iterations approach is applied until fixed-point is reached.

Note that the OUT flowmaps from the first pass are trivially safe-initial-estimates. Since there are no unsafe-initial-estimates, ghost mappings cannot occur in the final solution. Hence, the obtained fixed-point solution is also the maximum fixed-point (MFP) solution.

Node	OUT map
N0	{y:2}
N1	{z:0, y:2}
N2	{z:⊤, y:⊤}
N3	{z:⊤, y:⊤}
~~N4~~	~~{}~~
N5	{z:⊤, y:⊤}
N6	{z:⊤, y:⊤}

Fig. 3. After the first-pass of Proposal A on Fig. 2d.

Example 3.2. Revisiting the compilation in Fig. 2, when the compiler uses Proposal A to update the dataflow solution for Fig. 2d, it correctly computes the MFP in Fig. 2f. The nodes reachable from seed-node N5 are {N2, N3, N6, N5}. In the first-pass, the OUT flowmaps for these nodes are reset to map all domain-elements to ⊤, as shown in Fig. 3. With these safe-initial-estimates, the

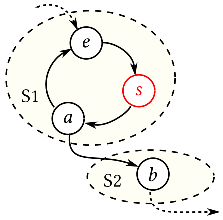
[2] We conjecture (but do not claim) that ghost-mappings are the sole source of imprecision. Note that our formal proofs in Section 4 cover imprecision arising out of any source.

```
y = 2;
z = 0;
do {
  e: ...
  x: y = y;
  s: z++;
} while (a: z < 5);
b: print y;
...
```

(a) Input code

Node	OUT map
e	{z:⊥, y:2}
x	{z:⊥, y:2}
s	{z:⊥, y:2}
a	{z:⊥, y:2}
b	{z:⊥, y:2}

(b) MFP for Fig. 4a



(c) CFG of Fig. 4a after removing node ‘x’

Node	OUT map
e	{z:⊤, y:⊤}
s	{z:⊤, y:⊤}
a	{z:⊤, y:⊤}
b	{z:⊤, y:⊤}

(d) Result of the first-pass of Proposal A, for Fig. 4c

Node	OUT map
e	{z:⊤, y:⊤}
s	{z:⊤, y:⊤}
a	{z:⊤, y:⊤}
b	{z:⊥, y:2}

(e) Result of the initialization-step of Proposal B, for Fig. 4c

Node	OUT map
e	{z:⊥, y:2}
s	{z:⊥, y:2}
a	{z:⊥, y:2}
b	{z:⊥, y:2}

(f) MFP solution for the modified program

Fig. 4. An example to demonstrate the benefits of applying two-step processing to one SCC at a time.

MFP solution shown in Fig. 2f is obtained in the second-pass, enabling the compiler to replace y with the literal constant 2 in node N6. □

Although this proposal obtains the MFP solution by reusing a part of the existing dataflow solution (for unreachable nodes from the seed-nodes), it may still be inefficient: Starting from the seed-nodes (which can even occur at the beginning of the CFG), this proposal resembles CompIDFA. In fact, due to the additional first-pass, this proposal may perform even worse than CompIDFA.

3.3 Utilizing SCC Decomposition Graphs for Efficiency

Observation B. *Proposal A can lead to redundant processing of many nodes due to naïve resetting of the* OUT *flowmaps of all nodes reachable from the seed-nodes, as well as due to the order in which these nodes are processed. We illustrate the first part of this observation with an example.*

Example 3.3. Consider the example program in Fig. 4a, and its MFP solution for intra-procedural constant-propagation analysis in Fig. 4b. Assume that the compiler removes the redundant copy operation at node ‘x’. Fig. 4c shows the resulting CFG snippet after the transformation. The set of seed nodes is {s}. The dashed ovals in the figure represent two SCCs S1 and S2 from the CFG. When employing Proposal A, all nodes reachable from ‘s’ are reset (shown in Fig. 4d). Note that after reaching the fixed point within S1, the OUT flowmap for node ‘a’ remains same as its old OUT flowmap (from before the first pass), as shown in Fig. 4f. The OUT flowmaps of node ‘b’ and beyond required no change (compared to their OUT flowmaps before the first pass). Yet, with Proposal A, nodes in S2 and beyond are redundantly processed as they were eagerly reset. □

Proposal B: InitRestart-SCC. To reduce the number of nodes processed, in this approach we selectively process one SCC at a time. Recall that the issue of ghost mappings originates due to the propagation of unsafe-initial-estimates within the cycles of a CFG. All cycles are contained within individual SCCs of the SDG. Further, the SDG of a CFG is always a directed-acyclic graph. Hence, by processing SCCs in topological sort order (an order popularized by Horwitz et al. [53]), no unsafe-initial-estimates from an unprocessed SCC can pollute the solution.

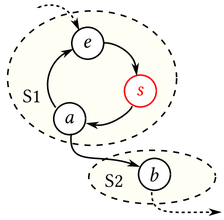
In Proposal B, an SCC is processed only if it contains seed nodes or if any entry node’s IN flowmap has changed. The processing happens in two steps – (i) *initialization-step*: first, the OUT flowmaps of all nodes in the SCC are reset to map all the domain-elements to ⊤ (a trivially safe-initial-estimate),

```
p = 0;
do {
  e: ...
  x: p = p;
  s: p = increment(p);
} while (a: p < 5);
b: print p;
```

(a) Input code

Node	OUT map
e	{p: ⊥}
x	{p: ⊥}
s	{p: ⊥}
a	{p: ⊥}
b	{p: ⊥}

(b) MFP for Fig. 5a



(c) CFG of Fig. 5a after removing node 'x'.

Fig. 5. An example to illustrate the optimization I/II of the initialization-step in Proposal C.

and (ii) *stabilization-step*: the standard IDFA processing is applied on all nodes within the SCC until the fixed-point is reached within the SCC. Upon reaching the fixed-point within the SCC, if the OUT flowmap of any of its exit nodes differs from its value before the initialization-step, then the corresponding successor SCCs are marked for processing. This solution avoids processing any SCC more than once and also minimizes the number of SCCs processed compared to Proposal A.

Example 3.4. For the CFG in Fig. 4c, Proposal B will first process the SCC S1 (which contains the seed-node 's') as follows: In the initialization-step, the OUT flowmaps of nodes 'e', 's', and 'a' are reset to map all the domain-elements to ⊤ (as shown in Fig. 4d). Then, in the stabilization-step, standard IDFA processing is applied on these three nodes until fixed-point (see Fig. 4f). Since the OUT flowmap of node 'a' does not change from its value before the initialization-step, SCC S2 (and beyond) is not processed, thereby reducing the overheads. □

3.4 IncIDFA: Optimized Two-Pass Approach per SCC

Observation C. *The size of an SCC can be quite large, leading to significant overheads in Proposal B, since the amount of work done within an SCC is similar to that in the full-recomputation approach. This is due to the naïve initialization-step, which resets* OUT *flowmaps of all the nodes in the SCC. These overheads can be reduced if MFP solution within each SCC is obtained incrementally, without resetting the* OUT *flowmaps of all the nodes.*

Proposal C: IncIDFA. In this proposal, we discuss two optimizations to speed up the initialization-step and another optimization for the stabilization-step.

3.4.1 Optimizing the Initialization-Step I/II. Instead of resetting all OUT flowmaps, we can use a worklist-based initialization-step. The worklist starts with the seed nodes in the SCC and any entry nodes of the SCC whose IN flowmaps may change due to updates in the predecessor SCCs. When processing a node n from the worklist, we can still obtain a safe-initial-estimate for n, if we ensure that during the calculation of the new IN flowmap, the OUT flowmap for those predecessors of n are *ignored* which (i) are present within the same SCC as n, *and* (ii) have not been processed even once during this incremental update. This ensures that while the flowmaps of n may be an under-approximation, they will still be safe-initial-estimates. If the OUT flowmap of n changes, its unprocessed successors from the same SCC are added to the worklist. Note that any node is processed at most once in the initialization-step. We demonstrate this optimization with an example.

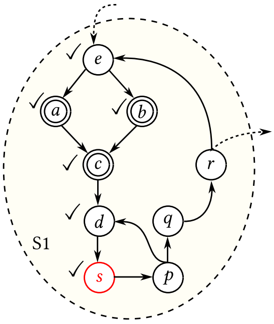
Example 3.5. Consider the example program shown in Fig. 5a, and its constant-propagation flowmaps in Fig. 5b. As before, assume that the compiler removes the redundant node 'x', resulting in the CFG snippet shown in Fig. 5c. Unlike Proposal B, the initialization-step for SCC S1 with this optimization does *not* reset the OUT flowmaps of nodes 'e', 's', and 'a'. Instead, a worklist is initialized with the seed-node 's'. When calculating the new IN flowmap for 's', the OUT flowmap

```
z = 0;
do {
  if (e: z < c1)
    a: z = 0;
  else
    b: z = 10;
  c: ...
  do {
    d: ...
    x: z = z;
    s: z++;
  } while (p: z < c2);
  q: ...
} while (r : z < c3);
```

(a) Input code

Phase	Node	OUT map
PhA	e	{z: ⊥}
PhA	a	{z: 0}
PhA	b	{z: 10}
PhA	c	{z: ⊥}
PhA	d	{z: ⊥}
-	x	{z: ⊥}
PhB	s	{z: ⊥}
-	p	{z: ⊥}
-	q	{z: ⊥}
-	r	{z: ⊥}

(b) MFP for Fig. 6a. 'Phase' denotes when a node gets processed.



(c) CFG of Fig. 6a after removing node 'x'. Nodes with a ✓ have been *visited*, and those in concentric circles have been *marked*.

Fig. 6. An example to illustrate the benefits of the two phases, PhA and PhB, of the initialization-step in Proposal C. Assume that c1, c2, and c3 are some constant literals.

of node 'a' is ignored as 'a' has not been processed yet. This trivially makes the IN flowmap of 's' a safe-initial-estimate. The OUT flowmap of 's' does not change upon processing – variable 'p' continues to map to ⊥, due to the function call. In this case, the worklist remains empty, and the initialization-step terminates. Note that in contrast with Proposal B, nodes 'e' and 'a' are not explicitly reset and redundantly processed. □

3.4.2 Optimizing the Initialization-Step II/II. When processing an entry node of an SCC in the first pass, at least one safe OUT flowmap is guaranteed to be available via its predecessors from the earlier SCCs. In contrast, for a seed-node (if not also an entry node), it is possible that none of the OUT flowmaps of the predecessors are safe. Thus, the domain-elements of the new IN flowmaps of such seed-nodes will be initially mapped to ⊤. This may significantly under-approximate the flowmaps of nodes being processed in the initialization-step, thereby increasing the number of times various nodes will have to be processed during the stabilization-step.

To address this challenge, we perform the initialization-step in two different phases, namely PhA and PhB. In the first phase, PhA, we *unconditionally* process all the nodes between the entry nodes and the seed-nodes exactly once, in their topological-sort order in the SCC. This guarantees that at least one safe OUT flowmap will be available to the seed nodes in the second phase. In PhB, we simply process the nodes starting with the seed-nodes as per optimization "I/II". To reduce the overheads while processing a node n in PhA, we avoid invoking its transfer function if the OUT flowmaps of all its predecessors are safe and unchanged, as illustrated next.

Example 3.6. Consider an example program in Fig. 6a and its MFP solution for constant-propagation analysis in Fig. 6b. Assume that the compiler removes the redundant node 'x', resulting in Fig. 6c. As per this optimization, all the nodes starting the entry node 'e', up until the seed node 's', are processed in PhA. Note that the OUT flowmaps of nodes 'a' and 'b' will not change. Regardless, node 'c' gets added to the worklist. However, since both the predecessors of 'c' provide safe and unchanged OUT flowmaps, its transfer function is not invoked. Note that the flowmaps of node 'd' are recalculated as one of its predecessors, node 'p', is unsafe. Now since 'd' has been visited, the new IN flowmap of node 's', in PhB, will not map its domain-elements with the init-value ⊤.

Further, with this optimization the OUT flowmap of node 's' will not change upon processing. The initialization-step will end with the processing of node 's'; subsequent nodes like 'p', 'q' and 'r' will not need to be processed to be considered safe. On the other hand, even if the OUT flowmap of node ‘s’ had changed, it would have been more likely that due to its non-$\top$ IN, the nodes starting ‘p’ and beyond within the SCC will need to be processed fewer times in the stabilization-step. □

Note that compared to this proposed optimization, a naïve approach of simply adding all the entry nodes of the SCC to the worklist (along with the seed-nodes) will not suffice: when processing a seed-node there will be no guarantee that the OUT flowmaps of at least one of its predecessor is safe. This can also be observed for the CFG shown in Fig. 6c, where with the naïve approach the new IN flowmap of node s will be initially mapped to $\top$.

3.4.3 Optimizing the Stabilization-Step. We observe that for certain nodes, the modified initialization-step (as per the first optimization) ignores the OUT flowmaps of one or more of their predecessors; consequently the IN flowmaps of such nodes may be under-approximated. We utilize this observation to speedup the stabilization-step: instead of starting the stabilization-step by redundantly populating the worklist with all the nodes of the SCC, we populate the worklist with *only* such under-approximated nodes. Note that at the beginning of the stabilization-step, the remaining nodes within the current SCC are not added to the worklist, as reprocessing them (in the absence of any changes to the OUT flowmaps of their predecessors) is redundant and will not change their flowmaps. The rest of the stabilization-step continues as before. We refer to the processing in this step as phase PhC in Section 4.

With our proposed fixes, we aim to reduce the work required to achieve the MFP solution during incremental update within each SCC (and consequently across the whole CFG).

4 Formal Description and Correctness of IncIDFA

In Section 4.1, we formalize key concepts related to IDFA and its incrementalization, as relevant to this formalism. We present a formal description of IncIDFA in Section 4.2. In Section 4.3, we provide proofs for its termination, soundness, and precision guarantees. For the sake of completeness, we also present a formal model of CompIDFA in Appendix A.2.

4.1 Background and Terminology, Formally

In this section, we formalize fundamental concepts related to IDFA and its incrementalization. The formalism is given for forward-IDFAs; results for backward-IDFAs are analogous.

Flowgraph. Each program is represented as a flowgraph, $P = (N, E, n_0)$, where N is the set of basic block nodes, $E \subseteq N \times N$ is the set of edges denoting the control flow, and $n_0 \in N$ is the entry node of the program. Note that the set E may also contain data-flow edges, such as the ones used in def-use graphs, and SSA graphs. Such edges are used in sparse dataflow analyses, as well as analyses of parallel programs. The set of all possible nodes is denoted by $\mathcal{N}$. The successors and predecessors of a node n in the flowgraph P are denoted by $\mathsf{succ}_P(n)$ and $\mathsf{pred}_P(n)$, respectively.

Strongly Connected Components. Let SDG_P denote the SCC Decomposition Graph of the flowgraph $P = (N, E, n_0)$; for simplicity, we assume that each node which is not part of any cycle corresponds to a singleton SCC. We denote the number of SCC-nodes in SDG_P by $|\mathsf{SDG}_P|$. The SCC at k^{th}-index in the topological sort ordering (0-indexed) of the SDG_P is denoted by $\mathsf{scc}_P(k)$. The set of program nodes in $\mathsf{scc}_P(k)$ is denoted by $\mathsf{sccNodes}_P(k)$, and its set of entry-nodes is defined as follows: $\mathsf{sccEntryNodes}_P(k) = \{n \in \mathsf{sccNodes}_P(k) | \exists m \in \mathsf{pred}_P(n), m \notin \mathsf{sccNodes}_P(k)\}$. Informally, entry-nodes of an SCC refer to those program nodes which have at least one predecessor from some other SCC; exit-nodes are defined analogously. For a given node, say n, the index of its SCC in the topological sort order of SDG_P is denoted by $\mathsf{sccID}_P(n)$, that is, $\forall n \in \mathsf{sccNodes}_P(k), \mathsf{sccID}_P(n) = k$. The set of successors and predecessors of n that are present in the same SCC as that of n, are represented by $\mathsf{sameSccSucc}_P(n)$ and $\mathsf{sameSccPred}_P(n)$, respectively.

Dataflow Analysis and Solution.

Definition 4.1 (Dataflow analysis). A dataflow analysis is defined as a 3-tuple, $\mathbb{A} = (\mathcal{L}, \mathcal{F}, \mathcal{M})$, where

- $\mathcal{L} = (V, \sqcap, \top, \bot, \sqsubseteq)$ is a bounded lattice consisting of set V of dataflow facts associated with the analysis $\mathbb{A}$, the binary meet operation $\sqcap$, the top element $\top$, the bottom element $\bot$ and the partial order $\sqsubseteq$ induced by $\sqcap$ on the elements of V.
- $\mathcal{F}$ denotes the set of all monotonic functions on the lattice. A function $f : V \longrightarrow V$ is *monotonic* if $\forall a, b \in V, a \sqsubseteq b \Rightarrow f(a) \sqsubseteq f(b)$.
- $\mathcal{M} : \mathcal{N} \longrightarrow \mathcal{F}$ maps each node of the language to some function in $\mathcal{F}$. For any node n, $\mathcal{M}(n)$ is termed as the *transfer function* or *flow function* of n.

Definition 4.2 (Analysis instance). An *analysis instance* of a dataflow analysis is a 2-tuple, $\mathcal{A} = (P, \vartheta_e)$, where,

- $P = (N, E, n_0)$ is the flowgraph of program on which the analysis is being performed.
- ϑ_e is the value of dataflow fact available at the beginning of the entry node n_0.

Definition 4.3 (Dataflow state). Consider a dataflow analysis $\mathbb{A} = (\mathcal{L}, \mathcal{F}, \mathcal{M})$, and its analysis instance $\mathcal{A} = (P, \vartheta_e)$, where $P = (N, E, n_0)$. We define a *dataflow state* of $\mathcal{A}$ to be a 2-tuple, $d = (\mathtt{IN}, \mathtt{OUT})$, where, $\mathtt{IN} : N \longrightarrow V$, and $\mathtt{OUT} : N \longrightarrow V$, map each node to its dataflow facts. Note: storing only IN or OUT suffices in practice, for simplicity in this formalism, we maintain both.

Definition 4.4 (Fixed-point solution). Consider a dataflow analysis $\mathbb{A} = (\mathcal{L}, \mathcal{F}, \mathcal{M})$, and its analysis instance $\mathcal{A} = (P, \vartheta_e)$, where $P = (N, E, n_0)$. A dataflow state, $d_{\text{fp}} = (\mathtt{IN}_{\text{fp}}, \mathtt{OUT}_{\text{fp}})$, is a *fixed-point solution* of $\mathcal{A}$, if the following hold $\forall n \in N$:

$$\mathtt{IN}_{\text{fp}}(n) = \begin{cases} \vartheta e, & \text{if } n = n_0 \\ \displaystyle\bigsqcap_{p \in \text{pred}_P(n)} \mathtt{OUT}_{\text{fp}}(p), & \text{otherwise} \end{cases} \quad \Bigg| \quad \mathtt{OUT}_{\text{fp}}(n) = \mathcal{T}_n(\mathtt{IN}_{\text{fp}}(n)), \text{where } \mathcal{T}_n = \mathcal{M}(n)$$

Note that an analysis instance may have multiple fixed-point solutions.

Definition 4.5 (Maximum fixed-point solution). Consider a dataflow analysis $\mathbb{A} = (\mathcal{L}, \mathcal{F}, \mathcal{M})$, and its analysis instance $\mathcal{A} = (P, \vartheta_e)$, where $P = (N, E, n_0)$. We term a fixed-point solution of $\mathcal{A}$, say $d_{\text{mfp}} = (\mathtt{IN}_{\text{mfp}}, \mathtt{OUT}_{\text{mfp}})$, as its *maximum fixed-point (MFP) solution* if for every other fixed-point solution of $\mathcal{A}$, say $d' = (\mathtt{IN}', \mathtt{OUT}') \neq d_{\text{mfp}}$, the following holds: $\forall n \in N, \mathtt{IN}'(n) \sqsubseteq \mathtt{IN}_{\text{mfp}}(n)$, and $\mathtt{OUT}'(n) \sqsubseteq \mathtt{OUT}_{\text{mfp}}(n)$.

Note that each analysis instance, $\mathcal{A}$, has a unique MFP solution, denoted by $\mathtt{mfp}(\mathcal{A})$.

Worklists. For a program $P = (N, E, n_0)$, a *worklist* is defined as an unordered subset of N, as is standard. In addition to standard operations such as union, intersection, and set-minus, we define two special non-deterministic methods on a worklist: (i) firstNode, which returns any node from the worklist, and (ii) firstNodeWithLeastSCCId, which returns any node n that has the least sccId among all nodes in the worklist. Both methods return an empty symbol, ϕ, if the worklist is empty.

Program Transformations and Incremental Analysis.

Definition 4.6 (Program change). Consider a program $P = (N, E, n_0)$, obtained by applying transformations to a program $P_{\text{old}} = (N_{\text{old}}, E_{\text{old}}, n_0)$; for simplicity, we assume that the entry node of a program cannot be changed during transformations. The program change between P and P_{old} is denoted by a 4-tuple, $\Delta(P_{\text{old}}, P) = <\delta_{\text{na}}, \delta_{\text{nr}}, \delta_{\text{ea}}, \delta_{\text{er}}>$, where, to obtain P from P_{old}, $\delta_{\text{na}} = N \setminus N_{\text{old}}$ is the minimal set of nodes added, $\delta_{\text{nr}} = N_{\text{old}} \setminus N$ is the minimal set of nodes removed, $\delta_{\text{ea}} = E \setminus E_{\text{old}}$ is the minimal set of edges added, and $\delta_{\text{er}} = E_{\text{old}} \setminus E$ is the minimal set of edges removed.

Note that our proposed approach is generic enough to handle all kinds of program changes, including structural modifications to the flowgraph.

Definition 4.7 (Seed nodes). Consider two programs $P = (N, E, n_0)$ and $P_{\text{old}} = (N_{\text{old}}, E_{\text{old}}, n_0)$, and the program-change between them, say $\Delta(P_{\text{old}}, P) = <\delta_{\text{na}}, \delta_{\text{nr}}, \delta_{\text{ea}}, \delta_{\text{er}}>$. The set of *seed nodes* for program change is defined as follows: $\text{seedNodes}(P_{\text{old}}, P) = \delta_{\text{na}} \cup \{n \in N | (*, n) \in \delta_{\text{er}}\} \cup \{n \in N | (*, n) \in \delta_{\text{ea}}\}$, where $*$ may represent any node in $N \cup N_{\text{old}}$.

Informally, any node n which is either new, or whose predecessor-set, $\text{pred}_P(n)$, has changed, is considered a seed node. The flow-maps at these nodes are directly impacted as a result of the transformations, and may require recalculation.

Definition 4.8 (Incremental-analysis instance). Consider a dataflow analysis $\mathbb{A} = (\mathcal{L}, \mathcal{F}, \mathcal{M})$we define an *incremental-analysis instance* as a 3-tuple, $\mathcal{I} = (\mathcal{A}, \text{seeds}, d_{\text{old}})$, where:

- $\mathcal{A} = (P, \vartheta_e)$ is an analysis instance of $\mathbb{A}$, where $P = (N, E, n_0)$ has been derived by transforming an old program $P_{\text{old}} = (N_{\text{old}}, E_{\text{old}}, n_0)$,
- seeds = $\text{seedNodes}(P_{\text{old}}, P)$, and
- d_{old} is the MFP solution of the analysis instance $\mathcal{A}_{\text{old}} = (P_{\text{old}}, \vartheta_e)$ for $\mathbb{A}$.

4.2 Formal Model for IncIDFA

Given an incremental-analysis instance $\mathcal{I} = (\mathcal{A}, \text{seeds}, d_{\text{old}})$, as defined in Definition 4.8, the aim of the `IncIDFA` algorithm is to obtain the MFP solution for $\mathcal{A}$. In this section, we formally define `IncIDFA` which was intuitively described in Section 3.4; we do so by expressing it as a fixed-point application of a function, named eval_{Inc}. Note that in order to assist the compiler developers in implementing `IncIDFA` in any framework, we have provided a more-readable, imperative-style, version of the algorithm in Appendix B. As evident in both the formalism as well as the algorithmic description, an analysis writer needs to provide the definitions for only the standard IDFA (lattice, transfer-functions, etc.), in order to use `IncIDFA`. We first explain the concepts of SCC-phase, `IncIDFA`-state, and initial `IncIDFA`-state, before presenting the evaluation rules of eval_{Inc}.

Definition 4.9 (SCC-phase). `IncIDFA` processes each SCC in three distinct phases, namely, PhA, PhB, and PhC, in order. These correspond to the three phases described in Sections 3.4.2 and 3.4.3. We term these phases as SCC-phases.

Definition 4.10 (`IncIDFA`-state). During application of `IncIDFA`, the state of the analysis, termed `IncIDFA`-state, is denoted by a 9-tuple, $\text{SI}_i = <k, W_k, d, \mathsf{S}_k, \mathsf{U}_k, \mathsf{M}_k, \mathcal{O}_k, \text{ph}, W_g>$[3], where

- k is the (0-based) index of the SCC being processed, in the topological order of the SDG,
- W_k denotes an intra-SCC worklist, containing nodes from $\text{sccNodes}_P(k)$ to be processed,
- d represents the current dataflow state,
- $\mathsf{S}_k \subseteq \text{sccNodes}_P(k)$ contains nodes that have been processed at least once during this incremental update, and, thereby, contain safe-initial-estimates,
- $\mathsf{U}_k \subseteq \text{sccNodes}_P(k)$ contains nodes for which recalculation of `IN` dataflow fact had ignored the `OUT` dataflow fact of at least one predecessor, and hence may be under-approximated,
- $\mathsf{M}_k \subseteq \text{sccNodes}_P(k)$ contains nodes whose `OUT` flow fact did not change upon recalculation,
- $\mathcal{O}_k : \text{sccExitNodes}_P(k) \rightarrow V$, maps each exit-node of the current SCC to its `OUT` dataflow fact prior to the processing of the SCC,
- ph $\in \{\text{PhA}, \text{PhB}, \text{PhC}\}$ denotes the current SCC-phase for the SCC being processed, and
- W_g denotes a global worklist of nodes not in the current SCC, which need to be processed.

We represent the set of all `IncIDFA`-states by $\mathcal{S}_I$.

[3] An easy mnemonic for the ordering of the four sets/maps, **S**afe, **U**nder-approximated, **M**arked, and **O**ut-maps, is SUMO.

$$\dfrac{\begin{array}{c} \mathrm{S}_I = \langle k, W_k, d, \mathrm{S}_k, \mathrm{U}_k, \mathrm{M}_k, O_k, \textsc{PhA}, W_g \rangle \qquad W_k \neq \emptyset \\ n = \mathsf{firstNode}(W_k) \qquad n \notin \mathsf{seeds} \qquad \mathsf{pred}_P(n) \subseteq \mathrm{M}_k \\ \mathrm{S}_k{}' = \mathrm{S}_k \cup \{n\} \qquad \mathrm{M}_k{}' = \mathrm{M}_k \cup \{n\} \\ W_k' = (W_k \setminus \{n\}) \cup (\mathsf{sameSccSucc}_P(n) \setminus \mathrm{S}_k{}' \setminus \mathsf{seeds}) \end{array}}{\mathbf{eval}_{\mathbf{Inc}}(\mathrm{S}_I) \Rightarrow \langle k, W_k', d, \mathrm{S}_k{}', \mathrm{U}_k, \mathrm{M}_k{}', O_k, \textsc{PhA}, W_g \rangle} \; [\textsc{PhA}\text{-Mark}]$$

$$\dfrac{\begin{array}{c} \mathrm{S}_I = \langle k, W_k, d, \mathrm{S}_k, \mathrm{U}_k, \mathrm{M}_k, O_k, \textsc{PhA}, W_g \rangle \qquad W_k \neq \emptyset \\ n = \mathsf{firstNode}(W_k) \qquad n \in \mathsf{seeds} \lor \mathsf{pred}_P(n) \not\subseteq \mathrm{M}_k \\ d = (\mathtt{IN}, \mathtt{OUT}) \qquad Q_p = (\mathsf{sameSccPred}_P(n) \cap \mathrm{S}_k) \cup (\mathsf{pred}_P(n) \setminus \mathsf{sameSccPred}_P(n)) \\ d' = (\mathtt{IN}', \mathtt{OUT}') = \mathsf{processNode}(d, n, Q_p) \qquad \mathrm{S}_k{}' = \mathrm{S}_k \cup \{n\} \\ \mathrm{U}_k{}' = \begin{cases} \mathrm{U}_k \cup \{n\}, & \text{if } Q_p \neq \mathsf{pred}_P(n) \\ \mathrm{U}_k, & \text{otherwise} \end{cases} \qquad \mathrm{M}_k{}' = \begin{cases} \mathrm{M}_k \cup \{n\}, & \text{if } \mathtt{OUT}'(n) = \mathtt{OUT}(n) \\ \mathrm{M}_k, & \text{otherwise} \end{cases} \\ W_k' = (W_k \setminus \{n\}) \cup (\mathsf{sameSccSucc}_P(n) \setminus \mathrm{S}_k{}' \setminus \mathsf{seeds}) \end{array}}{\mathbf{eval}_{\mathbf{Inc}}(\mathrm{S}_I) \Rightarrow \langle k, W_k', d', \mathrm{S}_k{}', \mathrm{U}_k{}', \mathrm{M}_k{}', O_k, \textsc{PhA}, W_g \rangle} \; [\textsc{PhA}\text{-Proc}]$$

$$\dfrac{\mathrm{S}_I = \langle k, \emptyset, d, \mathrm{S}_k, \mathrm{U}_k, \mathrm{M}_k, O_k, \textsc{PhA}, W_g \rangle \qquad W_k' = ((\mathsf{seeds} \cap \mathsf{sccNodes}_P(k)) \setminus \mathrm{S}_k)}{\mathbf{eval}_{\mathbf{Inc}}(\mathrm{S}_I) \Rightarrow \langle k, W_k', d, \mathrm{S}_k, \mathrm{U}_k, \emptyset, O_k, \textsc{PhB}, W_g \rangle} \; [\textsc{PhA}\text{-FP}]$$

Fig. 7. Evaluation rules for the $\mathsf{eval}_{\mathsf{Inc}}$ function for marking or processing the nodes in PhA.

$$\dfrac{\begin{array}{c} d = (\mathtt{IN}, \mathtt{OUT}) \qquad \mathtt{IN}_n' = \begin{cases} \bigsqcap_{p \in Q} \mathtt{OUT}(p), & \text{if } Q \neq \emptyset \\ \top & \text{otherwise} \end{cases} \\ \mathtt{OUT}_n' = \tau_n(\mathtt{IN}_n'), \text{ where } \tau_n = \mathcal{M}(n) \qquad d' = (\mathtt{IN}[n \leftarrow \mathtt{IN}_n'], \mathtt{OUT}[n \leftarrow \mathtt{OUT}_n']) \end{array}}{\mathsf{processNode}(d, n, Q) = d'}$$

Fig. 8. Helper function, processNode, to recalculate dataflow facts using the given set of predecessors.

Definition 4.11 (Initial IncIDFA*-state).* Consider a program $P_{\text{old}} = (N_{\text{old}}, E_{\text{old}}, n_0)$ and its modified version $P = (N, E, n_0)$. The initial IncIDFA-state for an incremental-analysis instance $\mathcal{I} = (\mathcal{A}, \mathsf{seeds}, d_{\text{old}} = (\mathtt{IN}_{\text{old}}, \mathtt{OUT}_{\text{old}}))$, is represented as $\mathsf{initState}_{\mathtt{IncIDFA}}(\mathcal{I}) = \langle k, W_k, d_{\text{init}}, \emptyset, \emptyset, \emptyset, O_k, \textsc{PhA}, W_g \rangle$, where:

$$k = \min_{\forall s \in \mathsf{seeds}} (\mathsf{sccID}_P(s)), \qquad W_k = \mathsf{sccEntryNodes}_P(k), \qquad d_{\text{init}} = (\mathtt{IN}_{\text{init}}, \mathtt{OUT}_{\text{init}}),$$
$$\forall n \in N \cap N_{\text{old}}, \mathtt{IN}_{\text{init}}(n) = \mathtt{IN}_{\text{old}}(n), \text{ and } \mathtt{OUT}_{\text{init}}(n) = \mathtt{OUT}_{\text{old}}(n),$$
$$\forall n \in N \setminus N_{\text{old}}, \mathtt{IN}_{\text{init}}(n) = \mathtt{OUT}_{\text{init}}(n) = \top,$$
$$\forall x \in \mathsf{sccExitNodes}_P(k) \cap N_{\text{old}}, O_k(x) = \mathtt{OUT}_{\text{old}}(x), \textit{and} \qquad W_g = \mathsf{seeds} \setminus \mathsf{sccNodes}_P(k)$$

This initial state ensures that IncIDFA is first applied on the least-id SCC that contains a seed-node.

Definition 4.12 ($\mathsf{eval}_{\mathsf{Inc}}$ function). We formally describe the IncIDFA algorithm using a function $\mathsf{eval}_{\mathsf{Inc}} : \mathcal{S}_I \longrightarrow \mathcal{S}_I$, which takes the current IncIDFA-state, and performs one step of the IncIDFA algorithm to generate the next IncIDFA-state.
For an incremental-analysis instance, $\mathcal{I}$, the application of IncIDFA algorithm can be seen as a fixed-point application of $\mathsf{eval}_{\mathsf{Inc}}$, denoted by $\mathsf{eval}_{\mathsf{Inc}}{}^{\infty}$, on the initial IncIDFA-state $\mathsf{initState}_{\mathtt{IncIDFA}}(\mathcal{I})$.

The evaluation rules for the $\mathsf{eval}_{\mathsf{Inc}}$ function are shown in Figs. 7-10, as discussed next.
Evaluation Rules for $\mathbf{eval}_{\mathbf{Inc}}$ in PhA (see Fig. 7). In phase PhA, the goal is to ensure that before processing any seed node in PhB, at least one safe OUT flow fact is available from its predecessors.

$$\frac{\begin{array}{c} \textsc{Si} = <k, W_k, d, \mathsf{S}_k, \mathsf{U}_k, \emptyset, O_k, \textsc{PhB}, W_g> \qquad W_k \neq \emptyset \qquad n = \mathrm{firstNode}(W_k) \\ d = (\mathtt{IN}, \mathtt{OUT}) \quad Q_p = (\mathrm{sameSccPred}_P(n) \cap \mathsf{S}_k) \cup (\mathrm{pred}_P(n) \setminus \mathrm{sameSccPred}_P(n)) \\ d' = (\mathtt{IN}', \mathtt{OUT}') = \mathtt{processNode}(d, n, Q_p) \quad \mathsf{S}_k{}' = \mathsf{S}_k \cup \{n\} \\ \mathsf{U}_k{}' = \begin{cases} \mathsf{U}_k \cup \{n\}, & \text{if } Q_p \neq \mathrm{pred}_P(n) \\ \mathsf{U}_k, & \text{otherwise} \end{cases} \\ W_k' = \begin{cases} (W_k \setminus \{n\}) \cup (\mathrm{sameSccSucc}_P(n) \setminus \mathsf{S}_k{}'), & \text{if } \mathtt{OUT}'(n) \neq \mathtt{OUT}(n) \text{ or } \mathtt{IN}(n) = \top \\ W_k \setminus \{n\}, & \text{otherwise} \end{cases} \end{array}}{\mathbf{eval}_{\mathbf{Inc}}(\textsc{Si}) \Rightarrow <k, W_k', d', \mathsf{S}_k{}', \mathsf{U}_k{}', \emptyset, O_k, \textsc{PhB}, W_g>} \ [\textsc{PhB-Proc}]$$

$$\frac{\textsc{Si} = <k, \emptyset, d, \mathsf{S}_k, \mathsf{U}_k, \emptyset, O_k, \textsc{PhB}, W_g>}{\mathbf{eval}_{\mathbf{Inc}}(\textsc{Si}) \Rightarrow <k, \mathsf{U}_k, d, \emptyset, \emptyset, \emptyset, O_k, \textsc{PhC}, W_g>} \ [\textsc{PhB-FP}]$$

Fig. 9. Evaluation rules for the $\mathrm{eval}_{\mathrm{Inc}}$ function for processing the nodes in PhB.

[PhA-Mark] Given an IncIDFA-state in PhA, if the intra-SCC worklist, W_k, is non-empty, the $\mathrm{eval}_{\mathrm{Inc}}$ function non-deterministically picks a node, say n, from W_k. Rule [PhA-Mark] is applied when node n is not a seed node and all its predecessors are present in M_k. In this case, recalculating the dataflow facts of n would not results in any changes. Hence, instead of applying the transfer function, we simply mark n as processed (and *safe*) by adding it to $\mathsf{S}_k{}'$ (see Section 3.4.2). Further, since the OUT dataflow fact of n has not changed, we add n to M_k. Finally, each successor of n in the current SCC is unconditionally added to the intra-SCC worklist, except if the successor (i) is a seed node, or (ii) has already been processed once during this incremental update.

[PhA-Proc] If node n is a seed node, or has at least one predecessor that is not in M_k, its dataflow facts may require recalculation. In line with the optimization "I/II" discussed in Section 3.4.1, this recalculation considers OUT flow facts from only those predecessors that have been processed at least once in this incremental update. This set, Q_p, includes all predecessors of n that are either in set S_k, or are from the prior SCCs. The recalculation of dataflow facts is done using the helper method processNode, which takes d, n, and Q_p, and returns the modified dataflow state, by invoking the flow function τ_n, as shown in Fig. 8. As before, node n is added to the set $\mathsf{S}_k{}'$, since it has now been processed. It is also added to the set $\mathsf{U}_k{}'$ if even one of its predecessors was ignored during the processing of n. Further, if the recalculated OUT dataflow fact for n did not change from its old value, then n is added to the set $\mathsf{M}_k{}'$. The worklist is updated unconditionally as in rule [PhA-Mark].

[PhA-FP] If the intra-SCC worklist is empty in PhA, the processing in phase PhA is complete. Note that when a seed node is not an entry node of its SCC, it does not get processed in PhA. Therefore, $\mathrm{eval}_{\mathrm{Inc}}$ switches to phase PhB, initializing the worklist with unprocessed seed nodes of the current SCC. Further, since the set M_k is not used in PhB, we simply pass an empty set.

Evaluation Rules for $\mathbf{eval}_{\mathbf{Inc}}$ in PhB (see Fig. 9). The goal of phase PhB is to remove all unsafe-initial-estimates from the SCC.

[PhB-Proc] In PhB, if the intra-SCC worklist, W_k, is not empty, $\mathrm{eval}_{\mathrm{Inc}}$ non-deterministically selects a node, say n, from W_k. The rule [PhB-Proc] is similar to [PhA-Proc], with the main difference being how the intra-SCC worklist is updated *conditionally*: Unlike [PhA-Proc], the worklist is updated only if either the OUT dataflow fact of node n changed, or if the IN dataflow fact for n was $\top$ (indicating n is a newly added node). In this case, successors of n that belong to the current SCC, and have not yet been processed in this incremental update, are added to W_k.

[PhB-FP] If the intra-SCC worklist is empty in PhB, $\mathrm{eval}_{\mathrm{Inc}}$ switches to phase PhC, initializing the worklist with the set U_k. Since sets S_k and U_k are not used in PhC, we pass empty sets.

$$\frac{\begin{array}{c} \textsc{Si} = <k, W_k, d, \emptyset, \emptyset, \emptyset, O_k, \textsc{PhC}, W_g> \qquad W_k \neq \emptyset \qquad n = \mathsf{firstNode}(W_k) \\ d = (\mathtt{IN}, \mathtt{OUT}) \quad d' = (\mathtt{IN}', \mathtt{OUT}') = \mathtt{processNode}(d, n, \mathsf{pred}_P(n)) \\ W'_k = \begin{cases} (W_k \setminus \{n\}) \cup \mathsf{sameSccSucc}_P(n), & \text{if } \mathtt{OUT}'(n) \neq \mathtt{OUT}(n) \\ W_k \setminus \{n\}, & \text{otherwise} \end{cases} \end{array}}{\mathbf{eval_{Inc}}(\textsc{Si}) \Rightarrow <k, W'_k, d', \emptyset, \emptyset, \emptyset, O_k, \textsc{PhC}, W_g>}\;[\textsc{PhC-Proc}]$$

$$\frac{\begin{array}{c} \textsc{Si} = <k, \emptyset, d, \emptyset, \emptyset, \emptyset, O_k, \textsc{PhC}, W_g> \quad d = (\mathtt{IN}, \mathtt{OUT}) \\ R = \{r \mid \exists x \in \mathsf{sccExitNodes}_P(k), O_k(x) \neq \mathtt{OUT}(x) \wedge r \in \mathsf{succ}_P(x) \setminus \mathsf{sameSccSucc}_P(x)\} \\ W'_g = (W_g \cup R) \quad W'_g \neq \emptyset \quad f = \mathsf{firstNodeWithLeastSCCId}(W'_g) \quad m = \mathsf{sccID}_P(f) \\ W_m = \mathsf{sccEntryNodes}_P(m) \quad W''_g = W'_g \setminus \mathsf{sccNodes}_P(m) \quad \forall x \in \mathsf{sccExitNodes}_P(m), O_m(x) = \mathtt{OUT}(x) \end{array}}{\mathbf{eval_{Inc}}(\textsc{Si}) \Rightarrow <m, W_m, d, \emptyset, \emptyset, \emptyset, O_m, \textsc{PhA}, W''_g>}\;[\textsc{PhC-FP}]$$

$$\frac{\textsc{Si} = <k, \emptyset, d, \emptyset, \emptyset, \emptyset, O_k, \textsc{PhC}, \emptyset> \quad d = (\mathtt{IN}, \mathtt{OUT}) \quad \forall x \in \mathsf{sccExitNodes}_P(k), O_k(x) = \mathtt{OUT}(x)}{\mathbf{eval_{Inc}}(\textsc{Si}) \Rightarrow \textsc{Si}}\;[\text{Inc-Global-FP}]$$

Fig. 10. Evaluation rules for the $\mathsf{eval_{Inc}}$ function for processing the nodes in PHC.

Evaluation Rules for $\mathbf{eval_{Inc}}$ in PHC (see Fig. 10). The goal of PHC is the MFP computation.

[PHC-Proc] In PHC, if the intra-SCC worklist, W_k, is non-empty, $\mathsf{eval_{Inc}}$ selects a node, say n, non-deterministically from W_k. Standard recalculation of dataflow facts of n, ignoring no predecessor, is done using processNode. In case of changes to its OUT dataflow fact, the successors of n from the current SCC are added to W_k. Note that a node may get processed multiple times in this phase.

[PHC-FP] When W_k is empty in PHC, $\mathsf{eval_{Inc}}$ has reached a local fixed point for the current SCC. Before processing the next SCC, the global worklist is updated as follows: for any exit-node x whose OUT flow fact has changed from its stored value in $O_k(x)$, the successors of x not belonging to the current SCC are added to the global worklist. If the resulting global worklist is not empty, the index m of the next SCC to be processed is obtained using the method firstNodeWithLeastSCCId. Recall that this method returns any node with least SCC-id in the given list. Before processing the SCC at index m, its nodes are removed from the global worklist, and its entry-nodes are added to the intra-SCC worklist.

[Inc-Global-FP] In PHC, $\mathsf{eval_{Inc}}$ reaches a global fixed-point when: (i) both the intra-SCC and global worklists are empty, and (ii) there are no exit-nodes whose OUT flow fact changed from their stored values in O_k. In this case, $\mathsf{eval_{Inc}}$ no longer alters the current IncIDFA-state.

4.3 Correctness Proofs for IncIDFA

We now outline the proof methodology for ensuring correctness of IncIDFA, focusing on its termination and precision guarantees. We show that the fixed-point application of $\mathsf{eval_{Inc}}$ on the initial IncIDFA-state of the given incremental-analysis instance produces the maximum fixed-point solution. Detailed definitions and proofs are present in Appendix A.1.

The correctness arguments for IncIDFA are based on two fundamental ideas, namely *consistency* and *safety*, defined next.

Definition A.1 (Consistent node). Consider a dataflow state $d_i = (\mathtt{IN}_i, \mathtt{OUT}_i)$, for some analysis instance $\mathcal{A} = (P, \vartheta_e)$, of a dataflow analysis $\mathbb{A} = (\mathcal{L}, \mathcal{F}, \mathcal{M})$. In the context of $\mathcal{A}$, or some incremental-analysis instance $\mathcal{I} = (\mathcal{A}, \mathsf{seeds}, d_{\mathrm{old}})$, for any node $n \in N$, we term n as a *consistent node* with respect to d_i, denoted as $\mathsf{consistent}(n, d_i)$, if and only if the following relations hold:

$$\mathtt{IN}_i(n) = \begin{cases} \vartheta e, & \text{if } n = n_0 \\ \displaystyle\bigsqcap_{p \in \mathsf{pred}_P(n)} \mathtt{OUT}_i(p), & \text{otherwise} \end{cases} \quad \Bigg| \quad \mathtt{OUT}_i(n) = \mathcal{T}_n(\mathtt{IN}_i(n)), \text{where } \mathcal{T}_n = \mathcal{M}(n)$$

When all nodes in an SCC at index k are consistent with respect to d_i, we say that $\mathsf{consistentSCC}(k, d_i)$ holds.

Note that when a node is consistent with respect to a dataflow state, its dataflow facts remain unchanged upon recalculation. Conversely, if the OUT dataflow fact changes, its successors may become inconsistent with respect to the updated dataflow state.

Theorem A.3 (Fixed-point via consistency). *A dataflow state $d_i = (\mathtt{IN}_i, \mathtt{OUT}_i)$ is a fixed-point solution for the analysis instance $\mathcal{A} = (P, \vartheta_e)$, as well as for all the incremental-analysis instances of the form $\mathcal{I} = (\mathcal{A}, *, *)$, if and only if the following holds: $\forall n \in N, \mathsf{consistent}(n, d_i)$.*

(The proof follows directly from Definitions 4.4 and A.1.)

Definition A.4 (Safe node). Consider a dataflow state $d_i = (\mathtt{IN}_i, \mathtt{OUT}_i)$, for some analysis instance $\mathcal{A} = (P, \vartheta_e)$, or for some incremental-analysis instance $\mathcal{I} = (\mathcal{A}, \mathsf{seeds}, d_{\text{old}})$. Let $d_{\text{mfp}} = (\mathtt{IN}_{\text{mfp}}, \mathtt{OUT}_{\text{mfp}})$ be the MFP solution of $\mathcal{A}$. For any node $n \in N$, we term n as a *safe node* with respect to d_i, denoted as $\mathsf{safe}(n, d_i)$, if and only if the following holds:

$$\mathtt{IN}_{\text{mfp}}(n) \sqsubseteq \mathtt{IN}_i(n), \text{ and } \mathtt{OUT}_{\text{mfp}}(n) \sqsubseteq \mathtt{OUT}_i(n)$$

When all nodes in an SCC at index k are safe with respect to d_i, we say that $\mathsf{safeSCC}(k, d_i)$ holds.

Theorem A.7 (Maximum fixed-point as consistency and safety). *A dataflow state $d_i = (\mathtt{IN}_i, \mathtt{OUT}_i)$ is the maximum fixed-point solution for the analysis instance $\mathcal{A}$, as well as for all the incremental-analysis instances of the form $\mathcal{I} = (\mathcal{A}, *, *)$, if and only if the following holds: $\forall n \in N, \mathsf{consistent}(n, d_i) \wedge \mathsf{safe}(n, d_i)$. (The proof relies on Theorem A.3 and Definitions 4.4, A.4, and 4.5.)*

Lemma A.19 (PhB leads to consistent or underapproximated nodes). *Consider an incremental-analysis instance $\mathcal{I} = (\mathcal{A}, \mathsf{seeds}, d_{\text{old}})$, as discussed above. Let $\mathrm{SI}_i = <k, \emptyset, d_i, S_k, U_k, \emptyset, O_k, \mathrm{PhB}, W_g>$ be the final state in PhB that is reached upon zero or more applications of $\mathsf{eval}_{\mathsf{Inc}}$. We note that in SI_i, $\forall n \in \mathsf{sccNodes}_P(k), \mathsf{consistent}(n, d_i) \vee n \in U_k$.*

Lemma A.22 (PhB leads to safe nodes). *Consider an incremental-analysis instance $\mathcal{I} = (\mathcal{A}, \mathsf{seeds}, d_{\text{old}})$, as discussed above. Let $\mathrm{SI}_i = <k, \emptyset, d_i, S_k, U_k, \emptyset, O_k, \mathrm{PhB}, W_g>$ be the final state in PhB that is reached upon zero or more applications of $\mathsf{eval}_{\mathsf{Inc}}$. We note that $\forall n \in \mathsf{sccNodes}_P(k), \mathsf{safe}(n, d_i)$.*

Lemma A.23 (PhC leads to consistent and safe nodes). *Consider an incremental-analysis instance $\mathcal{I} = (\mathcal{A}, \mathsf{seeds}, d_{\text{old}})$, as discussed above. Let $\mathrm{SI}_c = <k, \emptyset, d_c, \emptyset, \emptyset, \emptyset, O_k, \mathrm{PhC}, W_g>$ be the final state in PhC that is reached upon zero or more applications of $\mathsf{eval}_{\mathsf{Inc}}$. We note that $\forall n \in \mathsf{sccNodes}_P(k), \mathsf{safe}(n, d_c) \wedge \mathsf{consistent}(n, d_c)$.*

Theorem A.26. *Consider a dataflow analysis $\mathbb{A}$, and its analysis instance $\mathcal{A} = (P, \vartheta_e)$, where $P = (N, E, n_0)$ has been derived upon applying a sequence of one or more transformations on some program $P_{\text{old}} = (N_{\text{old}}, E_{\text{old}}, n_0)$. Let $\mathcal{A}_{\text{old}} = (P_{\text{old}}, \vartheta_e)$ be an analysis instance of $\mathbb{A}$ for P_{old}, and let d_{old} be its MFP solution. Let $\mathsf{seeds} = \mathsf{seeds}(P_{\text{old}}, P)$ represent the set of seed nodes for program transformations from P_{old} to P. Given the incremental-analysis instance $\mathcal{I} = (\mathcal{A}, \mathsf{seeds}, d_{\text{old}})$, if SI_{MFP} is the result of fixed-point application of the $\mathsf{eval}_{\mathsf{Inc}}$ for $\mathcal{I}$, and d_{mfp} is the dataflow state in SI_{MFP}, then*

$$\overset{k<|\mathsf{SDG}_P|}{\underset{k=0}{\forall}}, \mathsf{consistentSCC}(k, d_{\text{mfp}}) \wedge \mathsf{safeSCC}(k, d_{\text{mfp}})$$

Corollary A.27. *Consider a dataflow analysis $\mathbb{A}$, and its analysis instance $\mathcal{A} = (P, \vartheta_e)$, where $P = (N, E, n_0)$ has been derived upon applying a sequence of one or more transformations on some program $P_{\text{old}} = (N_{\text{old}}, E_{\text{old}}, n_0)$. Let $\mathcal{A}_{\text{old}} = (P_{\text{old}}, \vartheta_e)$ be an analysis instance of $\mathbb{A}$ for P_{old}, and let d_{old} be its MFP solution. Let* seeds = seeds(P_{old}, P) *represent the set of seed nodes for program transformations from P_{old} to P. Given the incremental-analysis instance $\mathcal{I} = (\mathcal{A}, \text{seeds}, d_{\text{old}})$, if $\text{SI}_{MFP}$is the result of fixed-point application of the* eval_{Inc} *for $\mathcal{I}$, and d_{mfp} is the dataflow state in SI_{MFP} then $d_{\text{mfp}} = \text{mfp}(\mathcal{A})$.*

5 Discussion

In this section, we first briefly discuss a heuristic, termed *accessed-keys heuristic*, used by IncIDFA to reduce the number of transfer-function applications. Then, we present a discussion on various other interesting aspects of our proposed techniques.

Accessed-Keys Heuristic. We note that a significant portion of time spent during IDFA-stabilization involves applying transfer functions. We propose a heuristic to reduce the frequency of transfer-function applications, based on two observations: (i) transfer functions often modify only a small part of the IN flowmap to produce the OUT flowmap, and (ii) in the context of incremental updates, changes to the IN flowmap may not always necessitate application of the transfer function.

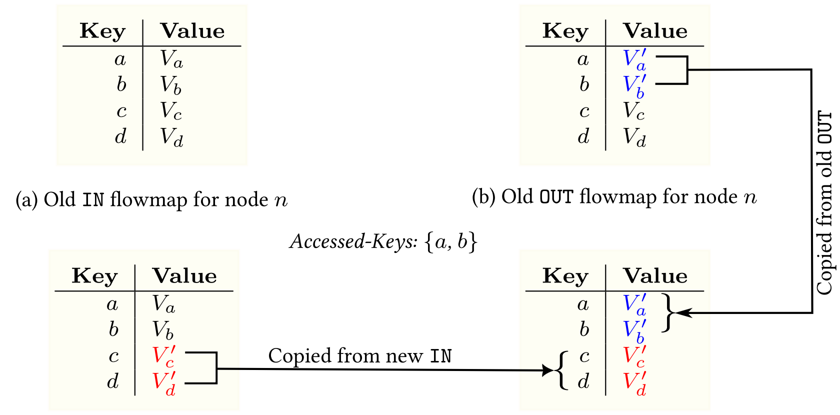
Domain of the flowmaps. A dataflow state, say $d = (\text{IN}, \text{OUT})$, contains two maps, namely IN and OUT. Given a program node n, these maps return the IN and OUT dataflow facts of the node n, respectively. Consider a dataflow analysis in which each dataflow fact itself is a map, termed as a *flowmap*, such as the constant-propagation flowmaps shown for each node in Fig. 2. The domain of the flowmaps, $\mathcal{D}$, is the set of elements over which this flowmap is defined. Examples include: program variables, abstract memory locations (on stack and heap), and so on. In such cases, the goal of the dataflow analysis is to calculate some meaningful information corresponding to each element of the domain $\mathcal{D}$ at every node of the program.

Accessed-Keys. Consider a node n. Let $\mathcal{D}$ be the domain of its flowmaps. For each application of transfer-function of n, say τ_n, we define the set of *accessed-keys*, denoted as $\mathcal{AK}(n)$, to contain all those elements of $\mathcal{D}$ corresponding to which the values in flowmaps of n may have been accessed during the application of τ_n. The accessed-keys set for a node can be easily captured by appropriately overriding the methods used to read and write from the flowmaps.

Let $\mathcal{AK}(n)$ be the set of accessed-keys for n during its last transfer-function application. If no element in $\mathcal{AK}(n)$ has changed between the old and new IN flowmaps, then: (i) for each $x \notin \mathcal{AK}(n)$, the new $\text{OUT}(x)$ equals the new $\text{IN}(x)$, and (ii) for each $x \in \mathcal{AK}(n)$, the new $\text{OUT}(x)$ equals the old $\text{OUT}(x)$. This heuristic eliminates unnecessry transfer-function applications. We illustrate this heuristic using an example.

Example A.1. In Fig. 11a, the IN map for node n is shown for an IDFA, where $\mathcal{D} = \{a, b, c, d\}$. Assume that the transfer function modifies the mappings of the domain-elements a and b to obtain the OUT flowmap as shown in Fig. 11b. Also assume that mappings for domain-elements c and d were not read or written. So, $\mathcal{AK}(n)$ = {a,b}. During incremental update, say the new IN map for n is as shown in Fig. 11c. Here, say the mappings for elements in $\mathcal{AK}(n)$ remain same as that in the old IN flowmap. Consequently, we can apply the accessed-keys heuristic to obtain the new OUT flowmap (shown in Fig. 11d) without applying its transfer-function, as follows: (i) For elements in $\mathcal{AK}(n)$, the mappings are obtained without changes from the old OUT flowmap. (ii) For the remaining elements, the mappings are copied from the new IN flowmap to the old OUT flowmap.

Order of Processing of Nodes. Iterative dataflow algorithms typically use a worklist-based approach, where the order in which nodes are extracted from the worklist can significantly influence



Key	Value
a	V_a
b	V_b
c	V_c
d	V_d

(a) Old IN flowmap for node n

Key	Value
a	V_a'
b	V_b'
c	V_c
d	V_d

(b) Old OUT flowmap for node n

Key	Value
a	V_a
b	V_b
c	V_c'
d	V_d'

(c) New IN flowmap for node n

Key	Value
a	V_a'
b	V_b'
c	V_c'
d	V_d'

(d) New OUT flowmap for node n

Fig. 11. Example demonstrating the accessed-key heuristics.

the number of times nodes are processed before reaching the maximum fixed-point solution. Accordingly, various node ordering strategies have been studied in the literature [3, 26, 50, 53, 62, 66, 70]. As shown by Horwitz et al. [53] and Cooper et al. [26], the *priority-SCC iteration* (PSI) approach, which uses a priority-queue for implementing the worklist, is the most efficient approach in general. The PSI approach processes each SCC at a time, in the topological-sort order of the SDG. Within each SCC, nodes are processed according to a reverse-postorder priority.

Given the efficiency of PSI, we use it for all the three schemes: CompIDFA, InitRestart, and IncIDFA. Note that there are two subtle differences between the PSI approaches for CompIDFA and IncIDFA: (i) in CompIDFA, the worklist is initialized with all nodes in an SCC, whereas in IncIDFA, only seed nodes are added, and (ii) IncIDFA includes a special first pass (initialization-pass, or PhA and PhB), where each node is processed at most once. Consequently, even when the program changes affect all nodes in the flowgraph, the order of processing of nodes (and hence the total number of times various nodes are processed) may differ between the PSI approaches used by CompIDFA and IncIDFA.

Instantiation of IncIDFA. In order to assess the generality of IncIDFA, we have implemented (i) a set of base classes in the IMOP compiler framework, each of which implements IncIDFA for some sub-category of iterative dataflow problems, and (ii) a set of 10 instantiations of these base classes to specific dataflow problems. Appendix C shows how our proposed IncIDFA works in the context of parallel programs. Fig. 12 summarizes the set of classes implemented.

In case of DataFlowAnalysis, we categorize the problems on the basis of whether their flow-facts are implemented as maps from the set of *cells* (that is, the set of abstract memory locations on stack and heap) to the set of some meaningful information. We term the type of such flow-facts as *cell-maps*. Note that in this hierarchy, our proposed accessed-keys heuristic is applicable only to those dataflow problems where the flow-facts are of type cell-maps (that is, those which are subclasses of CellularDataFlowAnalysis).

The description of various specializations of these subclasses is shown in Fig. 12. Note that all these instantiations automatically support incremental update of their dataflow solutions in response to any program-changes; the incrementalization logic is provided by the base classes.

Stabilization of the Helper Analyses. The incremental update of dataflow solutions for any iterative dataflow analysis may rely on the stabilization of analysis results of various other analyses, which we term as helper analyses. In our implementation of CompIDFA and IncIDFA in IMOP,

```
FlowAnalysis                                  superclass for all IDFAs
├ ControlFlowAnalysis                         class of IDFAs that do not depend on data-accesses
│ ├InterProceduralControlFlowAnalysis         intra-thread contex-insensitive inter-procedural IDFAs
│ │ └ DominanceAnalysis                       dominator information
│ └IntraProceduralControlFlowAnalysis         intra-thread intra-procedural IDFAs
│   └ PredicateAnalysis                       information about (subset of) branches taken
└ DataFlowAnalysis                            context-insensitive inter-procedural IDFAs based on r/w
  ├CellularDataFlowAnalysis                   class of IDFAs whose flow-facts are cell-maps
  │ ├InterThreadBackwardCellularAnalysis      inter-thread backward IDFAs with cell-maps
  │ └InterThreadForwardCellularAnalysis       inter-thread forward IDFAs with cell-maps
  │   ├ CopyPropagationAnalysis               information about variables that are copies of each other
  │   ├ PointsToAnalysis                      points-to information
  │   └ ReachingDefinitionAnalysis            reaching definitions
  └NonCellularDataFlowAnalysis                class of IDFAs whose cell-maps are not flow-facts
    ├InterThreadBackwardNonCellularAnalysis   inter-thread backward IDFAs without cell-maps
    │ ├ DataDependenceBackwardAnalysis        information about backward inter-thread data-dependences
    │ └ LivenessAnalysis                      liveness information of variables
    └InterThreadForwardNonCellularAnalysis    inter-thread forward IDFAs without cell-maps
      ├ DataDependenceForwardAnalysis         information about forward inter-thread data-dependences
      ├ HeapValidityAnalysis                  information about non-free'd pointers to heap
      └ LockSetAnalysis                       information about locks that must have been taken at a node
```

Fig. 12. Implementation and instantiations of IncIDFA in the IMOP compiler framework. The classes shown in bold are specific iterative dataflow problems.

one or more of the following analysis results must be stabilized before stabilizing the iterative dataflow solutions: (i) control-flow graphs, (ii) call graphs, (iii) phase information, (iv) inter-task and sibling-barrier edges, and (v) SCC Decomposition Graph. IMOP conforms to the self-stabilizing compiler design, *Homeostasis* [93], which automatically ensures that the stabilizations are triggered for required analysis results, at required compilation points, with required arguments. Consequently, stabilization of these helper analyses is guaranteed by IMOP.

Modes of IDFA-Stabilization. To assess the efficacy of the proposed incrementalization scheme, we compare IncIDFA with the following stabilization schemes: (i) CompIDFA, which corresponds to reinitialization of the dataflow solutions, followed by its exhaustive recomputation using the standard IDFA, and (ii) InitRestart, which corresponds to resetting the dataflow solution only at those nodes that are reachable from the seed nodes, followed by application of the restarting iterations approach (see Section 3.2). Note that InitRestart corresponds to one of the most straight-forward approaches to realize incremental update of dataflow solutions in response to program-changes, without losing on precision guarantees.

Further, for assessing the impact of accessed-keys heuristic, we study two variants for each of the aforementioned schemes: those with accessed-keys heuristic enabled (CompIDFA-AC, InitRestart-AC, IncIDFA-AC), and those without (CompIDFA-NAC, InitRestart-NAC, IncIDFA-NAC), leading to a total of six modes of IDFA-stabilization.

It is important to note that the primary objective of our approach is is to provide a *generic* incrementalization algorithm that can be applied to any arbitrary dataflow problem. Hence, we do not compare IncIDFA with techniques that rely on domain-specific knowledge about specific dataflow problems (such as [24, 45, 76, 79, 82, 101, 122, 127, 144, 149, 150]).

Generating Stabilization-Triggers and Program-Changes from a Real-World Client. In order to ensure that our evaluations do not rely on synthetically generated program-changes or stabilization-triggers, we have used BarrElim (see Nougrahiya and Nandivada [93]), which is a real-world set of optimization passes that is used to remove redundant barriers in OpenMP programs. The various components of BarrElim are (i) redundant barrier remover, (ii) OmpPar

Table 1. Benchmark characteristics, and baseline evaluation numbers for their compilation with `BarrElim`. All times are in seconds. Abbreviations: *#LOC*=lines of code, *#Node*/*#Edge*=number of nodes/edges in the CFG, *#SCC*=number of non-singleton SCCs, *#Barr*=number of static barriers, *#Trig*=number of stabilization triggers with `BarrElim`, *#N-Proc*=number of times nodes were processed during stabilization, *STB*=IDFA-stabilization time, and Tot=total compilation time.

1	2	3	4	5	6	7	8	9	10	11	12	13	14
Bench.	***#LOC***	***#Node***	***#Edge***	***#SCC***	***#Barr***	***#Trig***	***#N-Proc***	`CompIDFA-AC`				`IncIDFA-NAC`	
								Nanda		**K2**		**Nanda**	**K2**
								STB	**Tot**	**STB**	**Tot**	**STB**	**STB**
NPB													
1. BT	2615	4748	5016	11	47	21	115.0k	20.47	48.62	9.39	26.38	8.01	3.96
2. CG	642	1403	1485	15	31	70	83.8k	3.95	8.03	2.26	5.47	1.91	1.30
3. EP	352	775	813	9	4	2	1.9k	0.11	1.49	0.08	1.58	0.02	0.01
4. FT	899	2033	2151	10	14	45	96.3k	12.79	20.29	6.41	11.54	17.09	7.99
5. IS	333	711	762	5	4	2	1.6k	0.1	2.05	0.08	2.04	0.07	0.06
6. LU	2355	4687	4974	35	35	27	117.9k	14.63	27.71	7.53	16.59	12.10	6.30
7. MG	1278	2784	2918	6	19	205	645.2k	73.12	90.73	37.33	48.61	91.03	43.03
8. SP	2543	5364	5744	11	72	14	71.3k	7.88	24.93	4.03	14.25	6.62	3.45
SPEC													
9. quake	1489	3333	3491	9	22	30	90.5k	8.85	15.23	4.28	9	2.03	0.96
10. art-m	1691	1710	1791	13	4	9	44.5k	7.09	13.72	3.75	8.62	11.00	5.38
Sequoia													
11. amgmk	895	1867	1949	1	5	44	107.1k	10.37	17.15	5.58	10.31	11.92	6.49
12. clomp	1605	4162	4289	6	73	61	294.6k	24.91	92.57	13.24	44.52	26.30	13.32
13. stream	331	735	762	3	12	25	14.0k	0.62	2.6	0.5	2.5	0.40	0.32

expander, (iii) OmpPar merger, (iv) OmpPar-loop interchange, (v) OmpPar unswitching, (vi) variable privatization, (vii) function inliner, (viii) scope remover, and (ix) unused-elements remover. For a detailed discussion of these components, we refer the reader to the original paper.

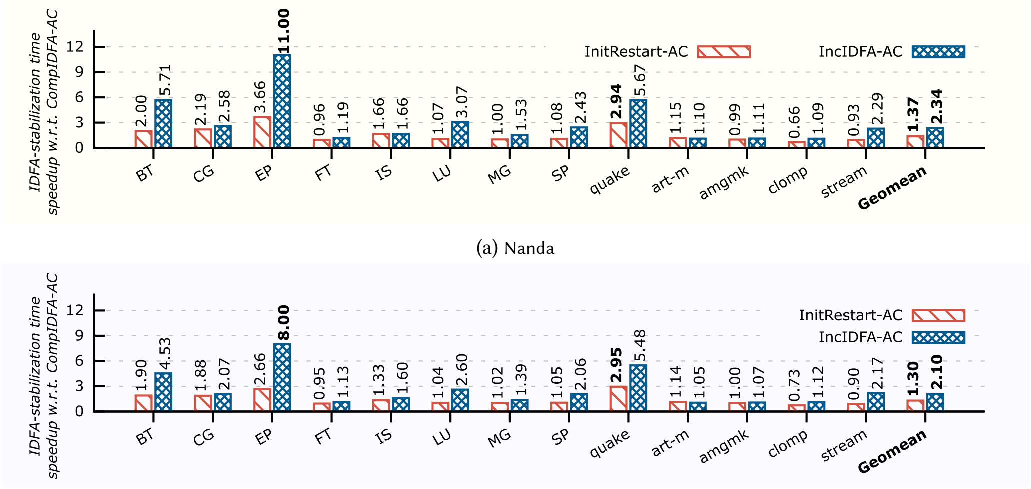
`BarrElim` has been implemented in IMOP. Using its self-stabilizing design Homeostasis [93], IMOP automatically keeps track of the program-changes in a centralized data structure efficiently, as and when they happen. When an attempt is made to read from a dataflow solution, the incremental analysis is triggered automatically, if there are pending program-changes to be handled.

6 Implementation and Evaluation

In order to evaluate the efficacy of our proposed approach, we have implemented our proposed scheme (`IncIDFA`) and the prior works (`CompIDFA`, and `InitRestart`) in the IMOP compiler framework [92], an open-source platform for parallel OpenMP C programs. IMOP spans over 170k lines of Java code, and has been effectively utilized in several published works [2, 73, 86, 93, 141–143]. For empirical evaluation, we have used all the six IDFA-stabilization modes, namely `CompIDFA-AC`, `CompIDFA-NAC`, `InitRestart-AC`, `InitRestart-NAC`, `IncIDFA-AC`, and `IncIDFA-NAC`, as discussed in Section 5. In total, the implementation of all these generic classes spans around 7kLOC in IMOP.

To assess the generality of our proposed approach, we implemented ten specific dataflow problems (listed in Section 5). As expected, no incrementalization-specific code was required in the implementation of any of these ten analyses. The complete artifact of this paper, including the source code, can be downloaded from Zenodo [94].

Experimental Setup. We empirically evaluated our proposed techniques using thirteen benchmark programs from three popular OpenMP benchmark suites (see Table 1). The benchmarks include: (i) all eight programs from NPB-OMP 3.0 suite [140], (ii) quake and art-m, the two (out of three)



(a) Nanda

(b) K2

Fig. 13. Speedup in IDFA-stabilization time using IncIDFA-AC when applying the client optimization, BarrElim, with respect to CompIDFA-AC. Higher is better.

OpenMP-C programs from SPEC OMP 2001 [8] that can be handled by IMOP, and (iii) all three OpenMP C programs – amgmk, clomp, and stream – from Sequoia benchmark suite [120]. These represent some of the largest open-source benchmarks for OpenMP C. Note that the remaining programs from SPEC OMP 2001 and Sequoia, which contain a mix of C/C++/MPI code, are not supported by IMOP and were excluded. Table 1 provides five key characteristics for each benchmark: lines of code (Column 2), number of program-nodes and edges in the flow graph (Columns 3 and 4, respectively), number of strongly connected components (SCCs) with more than one node (Column 5), and number of static barriers (Column 6).

In order to study the performance behaviour of our proposed approach across different hardware architectures, we used two platforms: (i) **Nanda**, a 64-thread 2.3 GHz Intel Xeon Gold 5218 system with 64 GB RAM; and (ii) **K2**, a 64-thread 2.3 GHz AMD Abu Dhabi system with 512 GB RAM. All reported compilation and execution times are geometric means over 30 runs (as recommended by Georges et al. [41]). We present our evaluations along the following three directions: (i) performance evaluation of IncIDFA-AC (in Section 6.1), (ii) performance impact of using the accessed-keys heuristic (in Section 6.2), and (iii) empirical correctness of IncIDFA-AC (in Section 6.3).

6.1 Performance Evaluation

We evaluate the efficacy of our proposed IncIDFA-AC algorithm along the two dimensions of IDFA-stabilization time and total compilation time, in the context of BarrElim, which includes a collection of real-world optimization passes and dataflow analysis passes (see Section 5). Towards that goal, we take CompIDFA-AC as our baseline, which corresponds to a naïve stabilization approach that entails exhaustive recomputation of the dataflow solutions from scratch, on any given program change. For fairness, the accessed-keys heuristic is enabled in CompIDFA-AC.

(A) IDFA-Stabilization Time. When compiling the benchmarks with BarrElim, the time spent in IDFA-stabilization using CompIDFA-AC is shown in Columns 9 and 11 of Table 1, for Nanda and K2, respectively. Fig. 13 shows the relative speedups in IDFA-stabilization time using IncIDFA-AC compared to CompIDFA-AC; for reference, we also include InitRestart-AC. As expected, IncIDFA-AC consistently outperforms both CompIDFA-AC and InitRestart-AC modes, across both the platforms

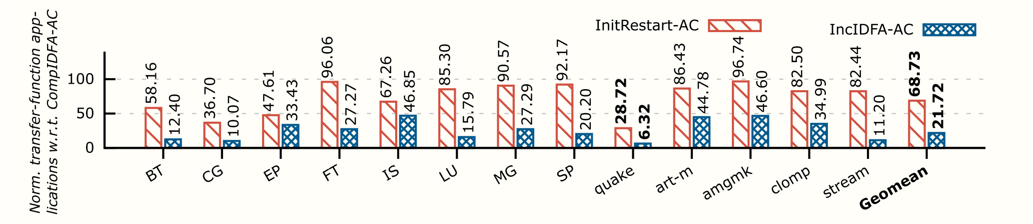


Fig. 14. Total number of applications of transfer-functions across all stabilization triggers when performing BarrElim optimization, for various configurations normalized with respect to the number of applications in case of CompIDFA-AC (set to 100). Lower is better.

– as compared to CompIDFA-AC, we observe the maximum speedup of 11.0× (geomean 2.34×) for Nanda, and that of 8.0× (geomean 2.10×) for K2.

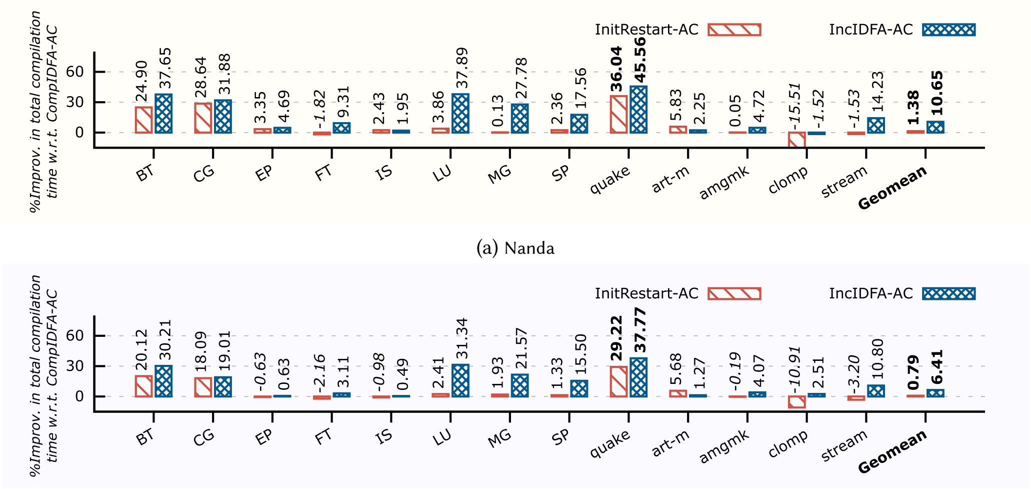
The gains in IDFA-stabilization time depend upon various factors specific to the stabilization-mode, such as (i) the number of times program nodes are processed, (ii) the number of skipped transfer function applications using the accessed-keys heuristics, (iii) cost of processing various types of nodes, and so on. In Table 1, Column 8 shows the number of times nodes are processed during the stabilization with CompIDFA-AC. The second factor (number of skipped transfer functions) for all three stabilization modes has been quantified in Section 6.2. In Fig. 14, we show the number of transfer function applications performed with IncIDFA-AC and InitRestart-AC, normalized to CompIDFA-AC. The speedups obtained for various benchmarks in Fig. 13 closely match the numbers shown in Fig. 14. We do not report the values for the third factor as it varies significantly across various nodes and benchmarks, and hence is difficult to summarize. We illustrate these observations through an inspection of the relative performance of each stabilization mode.

IncIDFA-AC vs. CompIDFA-AC. IncIDFA-AC results in significant improvements in IDFA-stabilization time as compared to CompIDFA-AC, as shown in Fig. 13. This is largely due to a significant reduction in the number of times nodes are processed using IncIDFA-AC compared to CompIDFA-AC. The maximum speedups are observed for EP (11× in Nanda, and 8× in K2), and quake (5.76× in Nanda, and 5.48× in K2), which is consequent upon the fact that the numbers of transfer-function applications with IncIDFA-AC (per 100 applications in CompIDFA-AC) are low for the case of EP (33.43 applications) and quake (6.32 applications). In contrast, we observe much lesser speedup gains for clomp (1.09× in Nanda, and 1.12× in K2), owing to a relatively smaller reduction in the number of transfer-function applications with IncIDFA-AC (34.99 applications per 100 applications with CompIDFA-AC).

IncIDFA-AC vs. InitRestart-AC. From Fig. 13, we note that IncIDFA-AC consistently outperforms InitRestart-AC across both the platforms (except for IS and art-m, where the performance is identical). The maximum gain is observed for EP (3.0× in both Nanda and K2), due to approximately 33.3% fewer transfer-function applications with IncIDFA-AC. Additionally, IncIDFA-AC avoids the overhead of traversing the flowgraph and reinitializing the flow facts, as required in InitRestart-AC. Fig. 14 shows that IncIDFA-AC consistently requires fewer transfer-function applications than InitRestart-AC.

Summary. Overall, we found that in terms of IDFA-stabilization time, IncIDFA-AC outperforms all the other stabilization modes. These gains improve the overall compilation time, as discussed next.

(B) Total Compilation Time. In Table 1, Columns 10 and 12 show the total compilation time with CompIDFA-AC, when applying BarrElim on the benchmarks under study, for Nanda and K2, respectively. This total compilation time spans from parsing the input file to applying the BarrElim optimizations and writing the final program to disk. Fig. 15 shows the benefits of using IncIDFA-AC compared to CompIDFA-AC, with InitRestart-AC included for reference. We consider



(a) Nanda

(b) K2

Fig. 15. Percentage improvement in total compilation time using IncIDFA-AC when applying the client optimization, BarrElim, with respect to CompIDFA-AC. Higher is better.

performance differences below 2% as negligible due to measurement errors. We see that with IncIDFA-AC, the maximum and geomean improvements with respect to CompIDFA-AC are 45.56% and 10.65% respectively in Nanda, and 37.77% and 6.41% respectively in K2. IncIDFA-AC consistently outperforms both CompIDFA-AC and InitRestart-AC (except for IS in Nanda, and art-m, where the difference is minor).

The gains in total compilation time depend on factors such as (i) the fraction of time spent in IDFA-stabilization (see Table 1, Columns 9 and 10 for Nanda, and Columns 11 and 12 for K2); (ii) overall speedup in the IDFA-stabilization time (shown in Fig. 13), and (iii) non-deterministic changes to the available heap size for the VM, and the associated GC overheads. This third factor is non-deterministic and infeasible to calculate empirically.

IncIDFA-AC vs. CompIDFA-AC. Fig. 15 shows that IncIDFA-AC consistently outperforms CompIDFA-AC in terms of the total compilation time (except for clomp in Nanda, where the performance degradation is negligible). The highest gains are for quake, where IDFA-stabilization time makes up a significant portion of the total time (58.10% in Nanda, and 47.55% in K2), with IncIDFA-AC resulting in significant gains in the IDFA-stabilization time (5.67× in Nanda, and 5.48× in K2). Similarly, MG sees 27.78% improvement in total time on Nanda, and 21.57% in K2, given that a significant amount of compilation time was spent in IDFA-stabilization (80.6% in Nanda, and 76.79% in K2), which, in turn, obtained a decent improvement with IncIDFA-AC (1.53× in Nanda, and 1.39× in K2). In contrast, despite the highest improvements in the IDFA-stabilization time (11× in Nanda, and 8× in K2), EP witnesses little (though significant) gains in the total compilation time (4.69% in Nanda, and 0.63% in K2). This is unsurprising, given that only a small fraction of the total time (7.38% in Nanda, and 5.06% in K2) is spent in IDFA-stabilization of EP.

IncIDFA-AC vs. InitRestart-AC. Except for minor degradation for IS and art-m, IncIDFA-AC consistently outperforms InitRestart-AC. The maximum gains are noted for MG (27.68% in Nanda, and 20.02% in K2), as a significantly large chunk of the total compilation time for MG is spent in its IDFA-stabilization, which in turn had a decent gain with IncIDFA-AC (1.53× in Nanda; 1.36× in K2).

Summary. IncIDFA-AC offers significant gains in overall compilation time, as compared to CompIDFA-AC and InitRestart-AC.

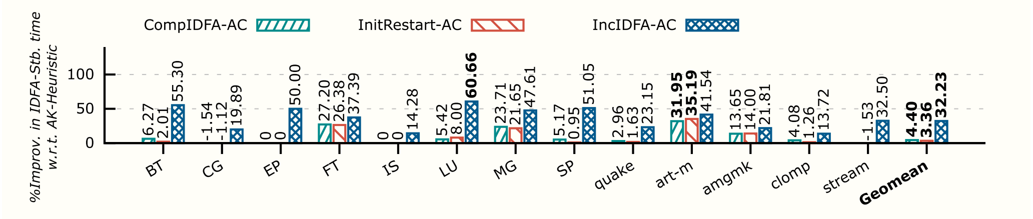


Fig. 16. Percentage improvement in IDFA-stabilization-time when applying the accessed-keys heuristic for all the three algorithms. The baseline in each case is the corresponding algorithm with accessed-keys heuristic disabled (such as CompIDFA-NAC for CompIDFA-AC). Higher is better.

6.2 Impact of Accessed-Keys Heuristic

We evaluate the impact of accessed-keys heuristic by comparing the heuristic-enabled versions of the three algorithms (CompIDFA-AC, InitRestart-AC, and IncIDFA-AC) with their heuristic-disabled counterparts (CompIDFA-NAC, InitRestart-NAC, and IncIDFA-NAC, respectively). Due to space constraints, we present the results for only one platform (Nanda) here; the results for K2 can be found in Appendix D. Fig. 16 shows the percentage improvement using the heuristic, for each of the three algorithms. We see that IncIDFA-AC consistently outperforms IncIDFA-NAC; improvements up to 60.66% (geomean 32.23%). Further, this graph demonstrates that accessed-keys heuristic has a significantly higher impact on IncIDFA algorithm, as compared to on CompIDFA and InitRestart – this is because the latter approaches reset the various flowmaps to their least informative values ($\top$), which decreases the likelihood of the heuristic being applicable.

The gains from the heuristic depends upon: (i) the number of transfer-function applications skipped, and (ii) the time spent in applying the transfer-functions when processing the nodes. For IncIDFA, we observed a maximum reduction of 84.57% in transfer-function applications, with a geomean of 64.89% (see Appendix D, Fig. 24). Note that quantifying the second factor is difficult, as it varies significantly across different kinds of nodes and dataflow analyses.

With the accessed-keys heuristic, we notice maximum gains in case of LU (60.66%), due to a total of 80.62% skipped transfer-function applications. In contrast, IS sees a gain of only 14.28%, given that only 48.89% of its transfer-function applications were skipped with the heuristic.

Summary. The accessed-keys heuristic significantly reduces the number of transfer-function applications, leading to substantial improvements in the IDFA-stabilization time.

6.3 Empirical Correctness

For each stabilization mode, we used IMOP to generate output text files with final dataflow facts annotated as comments on the nodes (statements/predicates). We verified that for all benchmarks, the files match verbatim across stabilization modes. Further, to ensure correct stabilization during BarrElim, we confirmed that the optimized code is identical across all stabilization modes.

Overall evaluation summary. Our evaluation demonstrates that (i) IncIDFA-AC improves IDFA-stabilization time and overall compilation time compared to naïve stabilization schemes, (ii) the accessed-keys heuristic is effective in further improving the compilation time, and (iii) our implementation ensures correct analysis and optimized code.

7 Related Work

In Section 1, we discussed key prior works related to the incrementalization of iterative approaches to dataflow analyses. Now we review the literature on various alternative approaches.

Elimination methods, such as structural analysis and interval analysis, work by analyzing the control-tree of the program [5, 7, 44, 135, 136]. By exploiting the hierarchical structure of the program, they are generally (i) faster than the iterative methods, and (ii) more amenable to efficient incremental updates [87]. Various incrementalization schemes have been given for such methods [15, 20–22, 58, 110–112, 126]. However, elimination methods are only applicable to reducible flowgraphs [51], limiting their use in various common scenarios, as noted by Cooper et al. [26].

Methods of attributes for dataflow analysis have been introduced by Babich and Jazayeri [10]. These methods use attribute grammars, where dataflow information is represented as attributes on parse-tree nodes, and the transfer functions are specified as part of the production rules. Several incremental update schemes in the context of attribute grammars have been proposed [4, 13, 14, 19, 31, 47, 56, 61, 64, 85, 102, 103, 106, 117, 145, 147, 148]. The key issue with attribute grammars is their inability to handle programs with cyclic or arbitrary control-flows.

Logic-programming-based methods offer an orthogonal approach to dataflow analysis, differing from the iterative techniques. These methods employ fixed-point application of logical rules and inference mechanisms on program facts to model and solve dataflow problems. Uwe Aßmann [9] introduced the idea of using Datalog for static analysis. Since then, many generic tools and extensions to Datalog, such as Soufflé [60], Flix [81], FlowSpec [124], Flan [1], and others [6, 71, 80, 91, 119] have been developed. In general, Datalog-based methods are amenable to fast incrementalization schemes. Consequently, various incrementalization frameworks like IncA [134], Ladder[133], and others [35, 38, 39, 39, 52, 99, 114–116, 121, 123, 131, 132, 155] have also been proposed. While the declarative nature of Datalog simplifies expression of complex dataflow analyses, it limits the analysis writer's control over the low-level computational details, as noted by Ceri et al. [23]. For instance, the order and method of execution of rules in the Datalog engines can typically not be customised by the analysis writer. Given that our proposed techniques depend heavily on the order in which various nodes are processed, it remains to be explored how they could be adapted to Datalog-based methods.

Demand-driven analyses [11, 33, 54, 104, 113] focus on inspecting only the relevant parts of a flowgraph to answer the given query. Following program edits, incremental updates are needed only for these relevant parts when the query is issued. In many demand-driven frameworks, queries are structured using the formalism of Context-Free Language (CFL) reachability [105], which models various program-properties (such as points-to relations) as realizable paths in a graph, such that the paths conform to the CFL. Various incrementalization approaches have been proposed for such demand-driven and CFL-reachability-based analyses [36, 79, 122, 129]. We believe that it is an interesting idea to extend our proposed ideas to demand driven analysis.

8 Conclusion

We presented IncIDFA, a novel algorithm that provides a generic solution for automatically generating precise incremental variants of any monotone iterative dataflow analysis. To validate the correctness and applicability of IncIDFA for a wide range of iterative dataflow problems, we provide proofs for the soundness and precision guarantees of IncIDFA. Alongside the core algorithm, we also presented a heuristic to reduce the number of transfer-function applications. To ensure applicability of IncIDFA to any arbitrary abstract domain and program edits, we also provided its formal model and correctness proofs. We implemented IncIDFA in the IMOP compiler for parallel OpenMP C programs, and instantiated it for ten specific dataflow problems, without requiring custom incremental-update code. Evaluations on a set of real-world optimization passes (BarrElim) and standard benchmark programs showed encouraging performance, demonstrating the capabilities of IncIDFA for efficiently and automatically handling incremental iterative algorithms for arbitrary monotone dataflow problems.

Data-Availability Statement

An artifact of this paper can be downloaded from Zenodo [94]. It contains the source code, benchmark programs, evaluation scripts, as well as the graph-generation scripts necessary to validate the results presented in this paper. In terms of reproducibility of results, the artifact has no known limitations to our knowledge.

Acknowledgments

This research was partially supported by IBM CAS project 1156, and SERB CRG project CRG/2022/006971.

References

[1] Supun Abeysinghe, Anxhelo Xhebraj, and Tiark Rompf. 2024. Flan: An Expressive and Efficient Datalog Compiler for Program Analysis. *Proc. ACM Program. Lang.* 8, POPL, Article 86 (Jan. 2024), 33 pages. https://doi.org/10.1145/3632928

[2] Aditya Agrawal and V. Krishna Nandivada. 2023. UWOmppro: UWOmp++ with Point-to-Point Synchronization, Reduction and Schedules. In *2023 32nd International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 27–38. https://doi.org/10.1109/PACT58117.2023.00011

[3] A. V. Aho and J. D. Ullman. 1975. Node Listings for Reducible Flow Graphs. In *Proceedings of the Seventh Annual ACM Symposium on Theory of Computing* (Albuquerque, New Mexico, USA) *(STOC '75)*. Association for Computing Machinery, New York, NY, USA, 177–185. https://doi.org/10.1145/800116.803767

[4] Henk Alblas. 1991. Incremental Attribute Evaluation. In *Attribute Grammars, Applications and Systems*, Henk Alblas and Boϑrivoj Melichar (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 215–233.

[5] F. E. Allen and J. Cocke. 1976. A Program Data Flow Analysis Procedure. *Commun. ACM* 19, 3 (March 1976), 137. https://doi.org/10.1145/360018.360025

[6] Mario Alvarez-Picallo, Alex Eyers-Taylor, Michael Peyton Jones, and C.-H. Luke Ong. 2019. Fixing Incremental Computation. In *Programming Languages and Systems*, Luís Caires (Ed.). Springer International Publishing, Cham, 525–552.

[7] 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* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 288–298. https://doi.org/10.1145/2568225.2568243

[8] Vishal Aslot, Max Domeika, Rudolf Eigenmann, Greg Gaertner, Wesley B. Jones, and Bodo Parady. 2001. SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance. In *Workshop on OpenMP Applications and Tools*, Rudolf Eigenmann and Michael J. Voss (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–10.

[9] Uwe Aßmann. 1996. On Edge Addition Rewrite Systems and Their Relevance to Program Analysis. In *Graph Grammars and Their Application to Computer Science*, Janice Cuny, Hartmut Ehrig, Gregor Engels, and Grzegorz Rozenberg (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 321–335.

[10] Wayne A. Babich and Mehdi Jazayeri. 1978. The Method of Attributes for Data Flow Analysis. Part I. Exhaustive Analysis. *Acta Informatica* 10, 3 (1978), 245–264.

[11] Wayne A. Babich and Mehdi Jazayeri. 1978. The Method of Attributes for Data Flow Analysis. Part II. Demand Analysis. *Acta Informatica* 10, 3 (1978), 265–272.

[12] Ayon Basumallik and Rudolf Eigenmann. 2008. Incorporation of OpenMP Memory Consistency into Conventional Dataflow Analysis. In *OpenMP in a New Era of Parallelism*, Rudolf Eigenmann and Bronis R. de Supinski (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 71–82.

[13] Jeroen Bransen, Atze Dijkstra, and S. Doaitse Swierstra. 2014. Lazy Stateless Incremental Evaluation Machinery for Attribute Grammars. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Partial Evaluation and Program Manipulation* (San Diego, California, USA) *(PEPM '14)*. Association for Computing Machinery, New York, NY, USA, 145–156. https://doi.org/10.1145/2543728.2543735

[14] Jeroen Bransen, Atze Dijkstra, and S. Doaitse Swierstra. 2015. Incremental Evaluation of Higher Order Attributes. In *Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation* (Mumbai, India) *(PEPM '15)*. Association for Computing Machinery, New York, NY, USA, 39–48. https://doi.org/10.1145/2678015.2682541

[15] Michael Burke. 1990. An Interval-Based Approach to Exhaustive and Incremental Interprocedural Data-Flow Analysis. *ACM Trans. Program. Lang. Syst.* 12, 3 (Jul 1990), 341–395. https://doi.org/10.1145/78969.78963

[16] M.G. Burke and B.G. Ryder. 1990. A Critical Analysis of Incremental Iterative Data Flow Analysis Algorithms. *IEEE Transactions on Software Engineering* 16, 7 (1990), 723–728. https://doi.org/10.1109/32.56098

[17] Haipeng Cai and John Jenkins. 2018. Leveraging Historical Versions of Android Apps for Efficient and Precise Taint

Analysis. In *Proceedings of the 15th International Conference on Mining Software Repositories* (Gothenburg, Sweden) *(MSR '18)*. Association for Computing Machinery, New York, NY, USA, 265–269. https://doi.org/10.1145/3196398.3196433

[18] David Callahan and Jaspal Sublok. 1988. Static Analysis of Low-Level Synchronization. *SIGPLAN Not.* 24, 1 (Nov 1988), 100–111. https://doi.org/10.1145/69215.69225

[19] Alan Carle and Lori Pollock. 1989. Modular Specification of Incremental Program Transformation Systems. In *Proceedings of the 11th International Conference on Software Engineering* (Pittsburgh, Pennsylvania, USA) *(ICSE '89)*. Association for Computing Machinery, New York, NY, USA, 178–187. https://doi.org/10.1145/74587.74612

[20] Martin Carroll and Barbara G Ryder. 1987. An Incremental Algorithm for Software Analysis. In *Proceedings of the Second ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (Palo Alto, California, USA) *(SDE 2)*. Association for Computing Machinery, New York, NY, USA, 171–179. https://doi.org/10.1145/24208.24228

[21] M. D. Carroll and B. G. Ryder. 1988. Incremental Data Flow Analysis via Dominator and Attribute Update. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '88)*. Association for Computing Machinery, New York, NY, USA, 274–284. https://doi.org/10.1145/73560.73584

[22] Steven Carroll and Constantine Polychronopoulos. 2004. A Framework for Incremental Extensible Compiler Construction. *International Journal of Parallel Programming* 32, 4 (2004), 289–316. https://doi.org/10.1023/B:IJPP.0000035816.93295.68

[23] S. Ceri, G. Gottlob, and L. Tanca. 1989. What You Always Wanted to Know About Datalog (And Never Dared to Ask). *IEEE Trans. on Knowl. and Data Eng.* 1, 1 (March 1989), 146–166. https://doi.org/10.1109/69.43410

[24] Yuting Chen, Qiuwei Shi, and Weikai Miao. 2015. Incremental Points-to Analysis for Java via Edit Propagation. In *Structured Object-Oriented Formal Language and Method*, Shaoying Liu and Zhenhua Duan (Eds.). Springer International Publishing, Cham, 164–178.

[25] Christopher L. Conway, Kedar S. Namjoshi, Dennis Dams, and Stephen A. Edwards. 2005. Incremental Algorithms for Inter-procedural Analysis of Safety Properties. In *Computer Aided Verification*, Kousha Etessami and Sriram K. Rajamani (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 449–461.

[26] Keith D. Cooper, Timothy J. Harvey, and Ken Kennedy. 2002. *Iterative Data-flow Analysis, Revisited.* Technical Report. Rice University.

[27] Keith D. Cooper and Ken Kennedy. 1984. Efficient Computation of Flow Insensitive Interprocedural Summary Information. In *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction* (Montreal, Canada) *(SIGPLAN '84)*. Association for Computing Machinery, New York, NY, USA, 247–258. https://doi.org/10.1145/502874.502898

[28] Patrick Cousot and Radhia Cousot. 2002. Modular Static Program Analysis. In *Compiler Construction*, R. Nigel Horspool (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 159–179.

[29] Leonardo Dagum and Ramesh Menon. 1998. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE* 5, 1 (1998), 46–55.

[30] Arnab De, Deepak D'Souza, and Rupesh Nasre. 2011. Dataflow Analysis for Datarace-Free Programs. In *Programming Languages and Systems*, Gilles Barthe (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 196–215.

[31] Alan Demers, Thomas Reps, and Tim Teitelbaum. 1981. Incremental Evaluation for Attribute Grammars with Application to Syntax-Directed Editors. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Williamsburg, Virginia) *(POPL '81)*. Association for Computing Machinery, New York, NY, USA, 105–116. https://doi.org/10.1145/567532.567544

[32] Lisa Nguyen Quang Do, Karim Ali, Benjamin Livshits, Eric Bodden, Justin Smith, and Emerson Murphy-Hill. 2017. Just-in-Time Static Analysis. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Santa Barbara, CA, USA) *(ISSTA 2017)*. Association for Computing Machinery, New York, NY, USA, 307–317. https://doi.org/10.1145/3092703.3092705

[33] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. 1995. Demand-Driven Computation of Interprocedural Data Flow. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '95)*. Association for Computing Machinery, New York, NY, USA, 37–48. https://doi.org/10.1145/199448.199461

[34] Evelyn Duesterwald and Mary Lou Soffa. 1991. Concurrency Analysis in the Presence of Procedures Using a Data-Flow Framework. In *Proceedings of the Symposium on Testing, Analysis, and Verification* (Victoria, British Columbia, Canada) *(TAV4)*. Association for Computing Machinery, New York, NY, USA, 36–48. https://doi.org/10.1145/120807.120811

[35] Michael Eichberg, Matthias Kahl, Diptikalyan Saha, Mira Mezini, and Klaus Ostermann. 2007. Automatic Incrementalization of Prolog Based Static Analyses. In *Proceedings of the 9th International Conference on Practical Aspects of Declarative Languages* (Nice, France) *(PADL'07)*. Springer-Verlag, Berlin, Heidelberg, 109–123. https://doi.org/10.1007/978-3-540-69611-7_7

[36] Julian Erhard, Simmo Saan, Sarah Tilscher, Michael Schwarz, Karoliine Holter, Vesal Vojdani, and Helmut Seidl. 2024. Interactive Abstract Interpretation: Reanalyzing Multithreaded C Programs for Cheap. *International Journal on*

Software Tools for Technology Transfer (2024). https://doi.org/10.1007/s10009-024-00768-9

[37] István Forgács. 1994. Double Iterative Framework for Flow-Sensitive Interprocedural Data Flow Analysis. *ACM Trans. Softw. Eng. Methodol.* 3, 1 (Jan 1994), 29–55. https://doi.org/10.1145/174634.174635

[38] Isabel Garcia-Contreras, Jose F. Morales, and Manuel V. Hermenegildo. 2019. Incremental Analysis of Logic Programs with Assertions and Open Predicates. In *Logic-Based Program Synthesis and Transformation: 29th International Symposium, LOPSTR 2019, Porto, Portugal, October 8-10, 2019, Revised Selected Papers* (Porto, Portugal). Springer-Verlag, Berlin, Heidelberg, 36–56. https://doi.org/10.1007/978-3-030-45260-5_3

[39] Isabel Garcia-Contreras, Jose F. Morales, and Manuel V. Hermenegildo. 2021. Incremental and Modular Context-Sensitive Analysis. *Theory and Practice of Logic Programming* 21, 2 (2021), 196–243. https://doi.org/10.1017/S1471068420000496

[40] GCC-Developer-Community. 2024. *GCC GitHub Repository.* Retrieved September 2, 2024 from https://github.com/gcc-mirror/gcc

[41] Andy Georges, Dries Buytaert, and Lieven Eeckhout. 2007. Statistically Rigorous Java Performance Evaluation. *SIGPLAN Not.* 42, 10 (Oct. 2007), 57–76. https://doi.org/10.1145/1297105.1297033

[42] Vida Ghoddsi. 1983. *Incremental Analysis of Programs.* Ph. D. Dissertation. University of Central Florida.

[43] Google. 2001. *Chrome V8.* https://github.com/v8/v8

[44] Susan L. Graham and Mark Wegman. 1976. A Fast and Usually Linear Algorithm for Global Flow Analysis. *J. ACM* 23, 1 (Jan 1976), 172–202. https://doi.org/10.1145/321921.321939

[45] William G. Griswold. 1993. Direct Update of Data Flow Representations for a Meaning-Preserving Program Restructuring Tool. In *Proceedings of the 1st ACM SIGSOFT Symposium on Foundations of Software Engineering* (Los Angeles, California, USA) *(SIGSOFT '93)*. Association for Computing Machinery, New York, NY, USA, 42–55. https://doi.org/10.1145/256428.167063

[46] Dirk Grunwald and Harini Srinivasan. 1993. Data Flow Equations for Explicitly Parallel Programs. *SIGPLAN Not.* 28, 7 (Jul 1993), 159–168. https://doi.org/10.1145/173284.155349

[47] Rajiv Gupta and Mary Lou Soffa. 1994. A Framework for Partial Data Flow Analysis. In *Proceedings of the International Conference on Software Maintenance (ICSM '94).* IEEE Computer Society, USA, 4–13.

[48] M.J. Harrold and G. Rothermel. 1996. Separate Computation of Alias Information for Reuse. *IEEE Transactions on Software Engineering* 22, 7 (1996), 442–460. https://doi.org/10.1109/32.538603

[49] Matthew S. Hecht. 1977. *Flow Analysis of Computer Programs.* Elsevier Science Inc., USA.

[50] Matthew S. Hecht and Jeffrey D. Ullman. 1973. Analysis of a Simple Algorithm for Global Data Flow Problems. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Boston, Massachusetts) *(POPL '73)*. Association for Computing Machinery, New York, NY, USA, 207–217. https://doi.org/10.1145/512927.512946

[51] M. S. Hecht and J. D. Ullman. 1974. Characterizations of Reducible Flow Graphs. *J. ACM* 21, 3 (jul 1974), 367–375. https://doi.org/10.1145/321832.321835

[52] Manuel Hermenegildo, German Puebla, Kim Marriott, and Peter J. Stuckey. 2000. Incremental Analysis of Constraint Logic Programs. *ACM Trans. Program. Lang. Syst.* 22, 2 (Mar 2000), 187–223. https://doi.org/10.1145/349214.349216

[53] Susan Horwitz, Alan Demers, and Tim Teitelbaum. 1987. An Efficient General Iterative Algorithm for Dataflow Analysis. *Acta Informatica* 24, 6 (1987), 679–694. https://doi.org/10.1007/BF00282621

[54] Susan Horwitz, Thomas Reps, and Mooly Sagiv. 1995. Demand Interprocedural Dataflow Analysis. *SIGSOFT Softw. Eng. Notes* 20, 4 (Oct. 1995), 104–115. https://doi.org/10.1145/222132.222146

[55] Lei Huang, Girija Sethuraman, and Barbara Chapman. 2008. Parallel Data Flow Analysis for OpenMP Programs. In *International Workshop on OpenMP*, Barbara Chapman, Weiming Zheng, Guang R. Gao, Mitsuhisa Sato, Eduard Ayguadé, and Dongsheng Wang (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 138–142.

[56] Scott E. Hudson. 1991. Incremental Attribute Evaluation: A Flexible Algorithm for Lazy Update. *ACM Trans. Program. Lang. Syst.* 13, 3 (Jul 1991), 315–341. https://doi.org/10.1145/117009.117012

[57] IBM. 2017. *Eclipse OpenJ9.* https://github.com/eclipse/openj9

[58] S. Jain and C. Thompson. 1988. An Efficient Approach to Data Flow Analysis in a Multiple Pass Global Optimizer. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation* (Atlanta, Georgia, USA) *(PLDI '88)*. Association for Computing Machinery, New York, NY, USA, 154–163. https://doi.org/10.1145/53990.54006

[59] Swati Jaiswal, Uday P. Khedker, and Alan Mycroft. 2021. A Unified Model for Context-Sensitive Program Analyses: The Blind Men and the Elephant. *ACM Comput. Surv.* 54, 6, Article 114 (July 2021), 37 pages. https://doi.org/10.1145/3456563

[60] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016. Soufflé: On Synthesis of Program Analyzers. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer International Publishing, Cham, 422–430.

[61] Gail E. Kaiser and Simon M. Kaplan. 1993. Parallel and Distributed Incremental Attribute Evaluation Algorithms for Multiuser Software Development Environments. *ACM Trans. Softw. Eng. Methodol.* 2, 1 (Jan 1993), 47–92.

https://doi.org/10.1145/151299.151312

[62] John B. Kam and Jeffrey D. Ullman. 1976. Global Data Flow Analysis and Iterative Algorithms. *J. ACM* 23, 1 (Jan 1976), 158–171. https://doi.org/10.1145/321921.321938

[63] John B. Kam and Jeffrey D. Ullman. 1977. Monotone Data Flow Analysis Frameworks. *Acta Informatica* 7, 3 (1977), 305–317. https://doi.org/10.1007/BF00290339

[64] Simon M Kaplan and Gail E Kaiser. 1986. Incremental Attribute Evaluation in Distributed Language-Based Environments. In *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing* (Calgary, Alberta, Canada) *(PODC '86)*. Association for Computing Machinery, New York, NY, USA, 121–130. https://doi.org/10.1145/10590.10601

[65] J. Keables, K. Roberson, and A. von Mayrhauser. 1988. Data Flow Analysis and its Application to Software Maintenance. In *Proceedings. Conference on Software Maintenance, 1988.* 335–347. https://doi.org/10.1109/ICSM.1988.10185

[66] K. W. Kennedy. 1975. Node Listings Applied to Data Flow Analysis. In *Proceedings of the 2nd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Palo Alto, California) *(POPL '75)*. Association for Computing Machinery, New York, NY, USA, 10–21. https://doi.org/10.1145/512976.512978

[67] Uday P. Khedker. 1995. *A Generalised Theory of Bit Vector Data Flow Analysis.* Ph. D. Dissertation.

[68] Uday P. Khedker and Dhananjay M. Dhamdhere. 1994. A Generalized Theory of Bit Vector Data Flow Analysis. *ACM Trans. Program. Lang. Syst.* 16, 5 (Sept. 1994), 1472–1511. https://doi.org/10.1145/186025.186043

[69] Uday P. Khedker and Dhananjay M. Dhamdhere. 1999. Bidirectional Data Flow Analysis: Myths and Reality. *SIGPLAN Not.* 34, 6 (June 1999), 47–57. https://doi.org/10.1145/606666.606676

[70] Gary A. Kildall. 1973. A Unified Approach to Global Program Optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Boston, Massachusetts) *(POPL '73)*. Association for Computing Machinery, New York, NY, USA, 194–206. https://doi.org/10.1145/512927.512945

[71] David Klopp, Sebastian Erdweg, and André Pacak. 2024. Object-Oriented Fixpoint Programming with Datalog. *Proc. ACM Program. Lang.* 8, OOPSLA2, Article 273 (Oct. 2024), 27 pages. https://doi.org/10.1145/3689713

[72] Jakob Krainz and Michael Philippsen. 2017. Diff Graphs for a Fast Incremental Pointer Analysis. In *Proceedings of the 12th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems* (Barcelona, Spain) *(ICOOOLPS'17)*. Association for Computing Machinery, New York, NY, USA, Article 4, 10 pages. https://doi.org/10.1145/3098572.3098578

[73] Gnanambikai Krishnakumar, Kommuru Alekhya Reddy, and Chester Rebeiro. 2019. ALEXIA: A Processor with Lightweight Extensions for Memory Safety. *ACM Trans. Embed. Comput. Syst.* 18, 6, Article 122 (Nov. 2019), 27 pages. https://doi.org/10.1145/3362064

[74] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization* (Palo Alto, California) *(CGO '04)*. IEEE Computer Society, Washington, DC, USA.

[75] Bozhen Liu and Jeff Huang. 2018. D4: Fast Concurrency Debugging with Parallel Differential Analysis. *SIGPLAN Not.* 53, 4 (Jun 2018), 359–373. https://doi.org/10.1145/3296979.3192390

[76] Bozhen Liu, Jeff Huang, and Lawrence Rauchwerger. 2019. Rethinking Incremental and Parallel Pointer Analysis. *ACM Trans. Program. Lang. Syst.* 41, 1, Article 6 (March 2019), 31 pages. https://doi.org/10.1145/3293606

[77] LLVM-Developer-Community. 2017. *LLVM GitHub Repository.* Retrieved August 26, 2024 from https://github.com/llvm/llvm-project/commit/b323f4f173710c60bcc76628d8155e476023c5b5

[78] LLVM-Developer-Community. 2024. *LLVM GitHub Repository.* Retrieved September 2, 2024 from https://github.com/llvm/llvm-project

[79] Yi Lu, Lei Shang, Xinwei Xie, and Jingling Xue. 2013. An Incremental Points-to Analysis with CFL-Reachability. In *Compiler Construction*, Ranjit Jhala and Koen De Bosschere (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 61–81.

[80] Magnus Madsen and Ondϑrej Lhoták. 2020. Fixpoints for the masses: programming with first-class Datalog constraints. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 125 (Nov. 2020), 28 pages. https://doi.org/10.1145/3428193

[81] Magnus Madsen, Ming-Ho Yee, and Ondϑrej Lhoták. 2016. From Datalog to Flix: a Declarative Language for Fixed Points on Lattices. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 194–208. https://doi.org/10.1145/2908080.2908096

[82] T.J. Marlowe and B.G. Ryder. 1991. Hybrid Incremental Alias Algorithms. In *Proceedings of the Twenty-Fourth Annual Hawaii International Conference on System Sciences*, Vol. ii. 428–437 vol.2. https://doi.org/10.1109/HICSS.1991.184005

[83] Thomas J. Marlowe and Barbara G. Ryder. 1989. An Efficient Hybrid Algorithm for Incremental Data Flow Analysis. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '90)*. Association for Computing Machinery, New York, NY, USA, 184–196. https://doi.org/10.1145/96709.96728

[84] Stephen P. Masticola and Barbara G. Ryder. 1993. Non-Concurrency Analysis. *SIGPLAN Not.* 28, 7 (Jul 1993), 129–138. https://doi.org/10.1145/173284.155346

[85] J. Micallef and G.E. Kaiser. 1993. Support Algorithms for Incremental Attribute Evaluation of Asynchronous Subtree Replacements. *IEEE Transactions on Software Engineering* 19, 3 (1993), 231–252. https://doi.org/10.1109/32.221136

[86] Prasoon Mishra and V. Krishna Nandivada. 2024. COWS for High Performance: Cost Aware Work Stealing for Irregular Parallel Loop. *ACM Transactions on Architecture and Code Optimization* 21, 1, Article 12 (Jan 2024), 26 pages. https://doi.org/10.1145/3633331

[87] Steven S. Muchnick. 1998. *Advanced Compiler Design and Implementation.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[88] Steven S. Muchnick and Neil D. Jones. 1981. *Program Flow Analysis: Theory and Application.* Prentice Hall Professional Technical Reference.

[89] Nomair A. Naeem, Ondϑrej Lhoták, and Jonathan Rodriguez. 2010. Practical Extensions to the IFDS Algorithm. In *Compiler Construction*, Rajiv Gupta (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 124–144.

[90] Lawton Nichols, Mehmet Emre, and Ben Hardekopf. 2019. Fixpoint Reuse for Incremental JavaScript Analysis. In *Proceedings of the 8th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis* (Phoenix, AZ, USA) *(SOAP 2019)*. Association for Computing Machinery, New York, NY, USA, 2–7. https://doi.org/10.1145/3315568.3329964

[91] Emma Nilsson-Nyman, Görel Hedin, Eva Magnusson, and Torbjörn Ekman. 2009. Declarative Intraprocedural Flow Analysis of Java Source Code. *Electronic Notes in Theoretical Computer Science* 238, 5 (2009), 155–171. Proceedings of the 8th Workshop on Language Descriptions, Tools and Applications (LDTA 2008).

[92] Aman Nougrahiya and V. Krishna Nandivada. 2019. *IMOP : IIT Madras OpenMP compiler framework.* https://github.com/amannougrahiya/imop-compiler

[93] Aman Nougrahiya and V. Krishna Nandivada. 2024. Homeostasis: Design and Implementation of a Self-Stabilizing Compiler. *ACM Transactions on Programming Languages and Systems* 46, 2, Article 6 (May 2024), 58 pages. https://doi.org/10.1145/3649308

[94] Aman Nougrahiya and V Krishna Nandivada. 2025. *Artifact for IncIDFA: an Efficient and Generic Algorithm for Incremental Iterative Dataflow Analysis.* https://doi.org/10.5281/zenodo.14598500

[95] Oracle. 1999. *HotSpot.* https://github.com/openjdk-mirror/jdk7u-hotspot

[96] Rohan Padhye and Uday P. Khedker. 2013. Interprocedural Data Flow Analysis in Soot using Value Contexts. In *Proceedings of the 2nd ACM SIGPLAN International Workshop on State Of the Art in Java Program Analysis* (Seattle, Washington) *(SOAP '13)*. Association for Computing Machinery, New York, NY, USA, 31–36. https://doi.org/10.1145/2487568.2487569

[97] Komal Pathade and Uday Khedker. 2023. Computing Maximum Fixed Point Solutions over Feasible Paths in Data Flow Analyses. *Science of Computer Programming* 228 (2023), 102944. https://doi.org/10.1016/j.scico.2023.102944

[98] L. L. Pollock and M. L. Soffa. 1989. An Incremental Version of Iterative Data Flow Analysis. *IEEE Transactions on Software Engineering* 15, 12 (1989), 1537–1549.

[99] Germán Puebla and Manuel Hermenegildo. 1996. Optimized Algorithms for Incremental Analysis of Logic Programs. In *Static Analysis*, Radhia Cousot and David A. Schmidt (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 270–284.

[100] Daniel Quinlan, Chunhua Liao, Thomas Panas, Robb Matzke, Markus Schordan, Rich Vuduc, and Qing Yi. 2013. *ROSE User Manual: A Tool for Building Source-to-Source Translators.* Technical Report. Lawrence Livermore National Laboratory.

[101] G. Ramalingam and Thomas Reps. 1994. An Incremental Algorithm for Maintaining the Dominator Tree of a Reducible Flowgraph. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon, USA) *(POPL '94)*. Association for Computing Machinery, New York, NY, USA, 287–296. https://doi.org/10.1145/174675.177905

[102] Thomas Reps. 1982. Optimal-Time Incremental Semantic Analysis for Syntax-Directed Editors. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Albuquerque, New Mexico) *(POPL '82)*. Association for Computing Machinery, New York, NY, USA, 169–176. https://doi.org/10.1145/582153.582172

[103] Thomas Reps. 1988. Incremental Evaluation for Attribute Grammars with Unrestricted Movement between Tree Modifications. *Acta Informatica* 25, 2 (1988), 155–178. https://doi.org/10.1007/BF00263583

[104] Thomas Reps. 1994. Solving Demand Versions of Interprocedural Analysis Problems. In *Compiler Construction*, Peter A. Fritzson (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 389–403.

[105] 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* (San Francisco, California, USA) *(POPL '95)*. Association for Computing Machinery, New York, NY, USA, 49–61. https://doi.org/10.1145/199448.199462

[106] Thomas Reps, Tim Teitelbaum, and Alan Demers. 1983. Incremental Context-Dependent Analysis for Language-Based Editors. *ACM Trans. Program. Lang. Syst.* 5, 3 (July 1983), 449–477. https://doi.org/10.1145/2166.357218

[107] Sara Royuela, Roger Ferrer, Diego Caballero, and Xavier Martorell. 2015. Compiler Analysis for OpenMP Tasks Correctness. In *Proceedings of the 12th ACM International Conference on Computing Frontiers* (Ischia, Italy) *(CF '15)*. Association for Computing Machinery, New York, NY, USA, Article 7, 8 pages. https://doi.org/10.1145/2742854.2742882

[108] Radu Rugina and Martin C. Rinard. 2003. Pointer Analysis for Structured Parallel Programs. *ACM Trans. Program. Lang. Syst.* 25, 1 (Jan 2003), 70–116. https://doi.org/10.1145/596980.596982

[109] B.G. Ryder, T.J. Marlowe, and M.C. Paull. 1988. Conditions for Incremental Iteration: Examples and Counterexamples. *Science of Computer Programming* 11, 1 (1988), 1–15.

[110] Barbara G. Ryder. 1983. Incremental Data Flow Analysis. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (Austin, Texas) *(POPL '83)*. Association for Computing Machinery, New York, NY, USA, 167–176. https://doi.org/10.1145/567067.567084

[111] Barbara G. Ryder and Marvin C. Paull. 1986. Elimination Algorithms for Data Flow Analysis. *ACM Comput. Surv.* 18, 3 (Sep 1986), 277–316. https://doi.org/10.1145/27632.27649

[112] Barbara G. Ryder and Marvin C. Paull. 1988. Incremental Data-Flow Analysis Algorithms. *ACM Trans. Program. Lang. Syst.* 10, 1 (Jan 1988), 1–50. https://doi.org/10.1145/42192.42193

[113] Mooly Sagiv, Thomas Reps, and Susan Horwitz. 1996. Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation. *Theoretical Computer Science* 167, 1 (1996), 131–170. https://doi.org/10.1016/0304-3975(96)00072-2

[114] Diptikalyan Saha and C. R. Ramakrishnan. 2005. Incremental and Demand-Driven Points-to Analysis Using Logic Programming. In *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming* (Lisbon, Portugal) *(PPDP '05)*. Association for Computing Machinery, New York, NY, USA, 117–128. https://doi.org/10.1145/1069774.1069785

[115] Diptikalyan Saha and C. R. Ramakrishnan. 2005. Symbolic Support Graph: A Space Efficient Data Structure for Incremental Tabled Evaluation. In *Logic Programming*, Maurizio Gabbrielli and Gopal Gupta (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 235–249.

[116] Diptikalyan Saha and C. R. Ramakrishnan. 2006. A Local Algorithm for Incremental Evaluation of Tabled Logic Programs. In *Logic Programming*, Sandro Etalle and Mirosław Truszczyński (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 56–71.

[117] João Saraiva, Doaitse Swierstra, and Matthijs Kuiper. 2000. Functional Incremental Attribute Evaluation. In *Compiler Construction*, David A. Watt (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 279–294.

[118] Shigehisa Satoh, Kazuhiro Kusano, and Mitsuhisa Sato. 2001. Compiler Optimization Techniques for OpenMP Programs. *Scientific Programming* 9, 2-3 (2001), 131–142.

[119] Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. 2016. On Fast Large-Scale Program Analysis in Datalog. In *Proceedings of the 25th International Conference on Compiler Construction* (Barcelona, Spain) *(CC '16)*. Association for Computing Machinery, New York, NY, USA, 196–206. https://doi.org/10.1145/2892208.2892226

[120] Seager, M. 2008. The ASC Sequoia Programming Model. (8 2008). https://doi.org/10.2172/945684

[121] Helmut Seidl, Julian Erhard, and Ralf Vogler. 2020. *Incremental Abstract Interpretation.* Springer International Publishing, Cham, 132–148. https://doi.org/10.1007/978-3-030-41103-9_5

[122] Lei Shang, Yi Lu, and Jingling Xue. 2012. Fast and Precise Points-to Analysis with Incremental CFL-Reachability Summarisation: Preliminary Experience. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. 270–273. https://doi.org/10.1145/2351676.2351720

[123] Shikha Singh, Sergey Madaminov, Michael A. Bender, Michael Ferdman, Ryan Johnson, Benjamin Moseley, Hung Ngo, Dung Nguyen, Soeren Olesen, Kurt Stirewalt, and Geoffrey Washburn. 2020. A Scheduling Approach to Incremental Maintenance of Datalog Programs. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 864–873. https://doi.org/10.1109/IPDPS47924.2020.00093

[124] Jeff Smits and Eelco Visser. 2017. FlowSpec: Declarative Dataflow Analysis Specification. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering* (Vancouver, BC, Canada) *(SLE 2017)*. Association for Computing Machinery, New York, NY, USA, 221–231. https://doi.org/10.1145/3136014.3136029

[125] Soot-Developer-Community. 2024. *Soot GitHub Repository.* Retrieved September 2, 2024 from https://github.com/soot-oss/soot

[126] Vugranam C. Sreedhar, Guang R. Gao, and Yong-Fong Lee. 1996. A New Framework for Exhaustive and Incremental Data Flow Analysis Using DJ Graphs. In *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation* (Philadelphia, Pennsylvania, USA) *(PLDI '96)*. Association for Computing Machinery, New York, NY, USA, 278–290. https://doi.org/10.1145/231379.231434

[127] Vugranam C. Sreedhar, Guang R. Gao, and Yong-Fong Lee. 1997. Incremental Computation of Dominator Trees. *ACM Trans. Program. Lang. Syst.* 19, 2 (Mar 1997), 239–252. https://doi.org/10.1145/244795.244799

[128] Richard M. Stallman and GCC-Developer-Community. 2009. *Using The GNU Compiler Collection: A GNU Manual For GCC Version 4.3.3.* CreateSpace, Paramount, CA.

[129] Benno Stein, Bor-Yuh Evan Chang, and Manu Sridharan. 2021. Demanded Abstract Interpretation. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 282–295. https://doi.org/10.1145/3453483.3454044

[130] Zewen Sun, Duanchen Xu, Yiyu Zhang, Yun Qi, Yueyang Wang, Zhiqiang Zuo, Zhaokang Wang, Yue Li, Xuandong Li, Qingda Lu, Wenwen Peng, and Shengjian Guo. 2023. BigDataflow: A Distributed Interprocedural Dataflow Analysis Framework. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1431–1443. https://doi.org/10.1145/3611643.3616348

[131] Tamás Szabó. 2023. Incrementalizing Production CodeQL Analyses. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1716–1726. https://doi.org/10.1145/3611643.3613860

[132] Tamás Szabó, Gábor Bergmann, Sebastian Erdweg, and Markus Voelter. 2018. Incrementalizing Lattice-Based Program Analyses in Datalog. *Proc. ACM Program. Lang.* 2, OOPSLA, Article 139 (oct 2018), 29 pages. https://doi.org/10.1145/3276509

[133] Tamás Szabó, Sebastian Erdweg, and Gábor Bergmann. 2021. Incremental Whole-Program Analysis in Datalog with Lattices. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 1–15. https://doi.org/10.1145/3453483.3454026

[134] Tamás Szabó, Sebastian Erdweg, and Markus Voelter. 2016. IncA: A DSL for the Definition of Incremental Program Analyses. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE 2016)*. Association for Computing Machinery, New York, NY, USA, 320–331. https://doi.org/10.1145/2970276.2970298

[135] Robert Tarjan. 1973. Testing Flow Graph Reducibility. In *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing* (Austin, Texas, USA) *(STOC '73)*. Association for Computing Machinery, New York, NY, USA, 96–107. https://doi.org/10.1145/800125.804040

[136] J. D. Ullman. 1972. A Fast Algorithm for the Elimination of Common Subexpressions. In *13th Annual Symposium on Switching and Automata Theory (swat 1972)*. 161–176. https://doi.org/10.1109/SWAT.1972.1

[137] 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* (Toronto, Ontario, Canada) *(CASCON '10)*. IBM Corp., USA, 214–224. https://doi.org/10.1145/1925805.1925818

[138] Jens Van der Plas, Quentin Stiévenart, and Coen De Roover. 2023. Result Invalidation for Incremental Modular Analyses. In *Verification, Model Checking, and Abstract Interpretation*, Cezara Dragoi, Michael Emmi, and Jingbo Wang (Eds.). Springer Nature Switzerland, Cham, 296–319.

[139] Jens Van der Plas, Quentin Stiévenart, Noah Van Es, and Coen De Roover. 2020. Incremental Flow Analysis through Computational Dependency Reification. In *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 25–36. https://doi.org/10.1109/SCAM51674.2020.00008

[140] Rob F Van der Wijngaart and Parkson Wong. 2002. *NAS parallel benchmarks version 3.0.* Technical Report. NAS technical report, NAS-02-007.

[141] Jyothi Krishna Viswakaran Sreelatha and Shankar Balachandran. 2016. Compiler Enhanced Scheduling for OpenMP for Heterogeneous Multiprocessors. In *Workshop on Energy Efficiency with Heterogeneous Computing (EEHCO '16)*. ACM, Prague, Czech Republic.

[142] Jyothi Krishna Viswakaran Sreelatha, Shankar Balachandran, and Rupesh Nasre. 2018. CHOAMP: Cost Based Hardware Optimization for Asymmetric Multicore Processors. *IEEE Trans. Multi-Scale Computing Systems* 4, 2 (2018), 163–176.

[143] Jyothi Krishna Viswakaran Sreelatha and Rupesh Nasre. 2018. Optimizing Graph Algorithms in Asymmetric Multicore Processors. *IEEE Trans. on CAD of Integrated Circuits and Systems* 37, 11 (2018), 2673–2684.

[144] Frédéric Vivien and Martin Rinard. 2001. Incrementalized Pointer and Escape Analysis. *SIGPLAN Not.* 36, 5 (May 2001), 35–46. https://doi.org/10.1145/381694.378804

[145] J. A. Walz and G. F. Johnson. 1988. Incremental Evaluation for a General Class of Circular Attribute Grammars. In *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation* (Atlanta, Georgia, USA) *(PLDI '88)*. Association for Computing Machinery, New York, NY, USA, 209–221. https://doi.org/10.1145/53990.54011

[146] Chengpeng Wang, Wuqi Zhang, Zian Su, Xiangzhe Xu, Xiaoheng Xie, and Xiangyu Zhang. 2024. When Dataflow Analysis Meets Large Language Models. *CoRR* abs/2402.10754 (2024). https://doi.org/10.48550/ARXIV.2402.10754 arXiv:2402.10754

[147] Dashing Yeh. 1983. On Incremental Evaluation of Ordered Attribute Grammars. *BIT Numerical Mathematics* 23, 3 (1983), 308–320. https://doi.org/10.1007/BF01934460

[148] D. Yeh and U. Kastens. 1988. Improvements of an Incremental Evaluation Algorithm for Ordered Attribute Grammars. *SIGPLAN Not.* 23, 12 (Dec 1988), 45–50. https://doi.org/10.1145/57669.57672

[149] Jyh-Shiarn Yur, Barbara G. Ryder, and William A. Landi. 1999. An Incremental Flow- and Context-Sensitive Pointer Aliasing Analysis. In *Proceedings of the 21st International Conference on Software Engineering* (Los Angeles, California, USA) *(ICSE '99)*. Association for Computing Machinery, New York, NY, USA, 442–451. https://doi.org/10.1145/302405.302676

[150] Jyh-Shiarn Yur, Barbara G. Ryder, William A. Landi, and Phil Stocks. 1997. Incremental Analysis of Side Effects for C Software System. In *Proceedings of the 19th International Conference on Software Engineering* (Boston, Massachusetts, USA) *(ICSE '97)*. Association for Computing Machinery, New York, NY, USA, 422–432. https://doi.org/10.1145/253228.253369

[151] Frank Kenneth Zadeck. 1984. Incremental Data Flow Analysis in a Structured Program Editor. In *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction* (Montreal, Canada) *(SIGPLAN '84)*. Association for Computing Machinery, New York, NY, USA, 132–143. https://doi.org/10.1145/502874.502888

[152] Sheng Zhan and Jeff Huang. 2016. ECHO: Instantaneous in Situ Race Detection in the IDE. 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, 775–786. https://doi.org/10.1145/2950290.2950332

[153] Yuan Zhang and Evelyn Duesterwald. 2007. Barrier Matching for Programs with Textually Unaligned Barriers. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Jose, California, USA) *(PPoPP '07)*. Association for Computing Machinery, New York, NY, USA, 194–204. https://doi.org/10.1145/1229428.1229472

[154] Yuan Zhang, Evelyn Duesterwald, and Guang R. Gao. 2008. Concurrency Analysis for Shared Memory Programs with Textually Unaligned Barriers. Springer-Verlag, Berlin, Heidelberg, Chapter Languages and Compilers for Parallel Computing, 95–109. https://doi.org/10.1007/978-3-540-85261-2_7

[155] David Zhao, Pavle Subotic, Mukund Raghothaman, and Bernhard Scholz. 2021. Towards Elastic Incrementalization for Datalog. In *23rd International Symposium on Principles and Practice of Declarative Programming* (Tallinn, Estonia) *(PPDP 2021)*. Association for Computing Machinery, New York, NY, USA, Article 20, 16 pages. https://doi.org/10.1145/3479394.3479415

A Formal Description and Correctness of IncIDFA

In Section A.1, we provide detailed proofs of correctness for precision and soundness guarantees of IncIDFA. Then, for sake of completeness, we also present the formal description of the traditional iterative dataflow analysis, CompIDFA in Section A.2.

A.1 Correctness Proofs for IncIDFA

Consider a dataflow analysis $\mathbb{A} = (\mathcal{L}, \mathcal{F}, \mathcal{M})$. Let $\mathcal{A} = (P, \vartheta_e)$ be an analysis instance of $\mathbb{A}$, where $P = (N, E, n_0)$ has been derived upon applying a sequence of one or more transformations on some program $P_{\text{old}} = (N_{\text{old}}, E_{\text{old}}, n_0)$. Let seeds = seeds$(P_{\text{old}}, P)$ represent the set of seed nodes for program transformations from P_{old} to P. Let d_{old} be the MFP solution of some analysis instance $\mathcal{A}_{\text{old}} = (P_{\text{old}}, \mathcal{M}_{\text{old}}, \vartheta_e)$ for $\mathbb{A}$. Given the analysis instance $\mathcal{A}$, and the incremental-analysis instance $\mathcal{I} = (\mathcal{A}, \text{seeds}, d_{\text{old}})$, in this section we prove the equivalence of CompIDFA and IncIDFA by showing that the fixed-point solutions obtained upon application of $\text{eval}_{\text{Comp}}$ on the initial CompIDFA-state of $\mathcal{A}$, and eval_{Inc} on the initial IncIDFA-state of $\mathcal{I}$, are the same.

Consistency and Safety.
In this section, we formally describe two fundamental ideas, namely *consistency* and *safety*, which form the basis for our correctness arguments for IncIDFA.

Definition A.1 (Consistent node). Consider a dataflow state $d_i = (\text{IN}_i, \text{OUT}_i)$ for some analysis instance $\mathcal{A} = (P, \vartheta_e)$, or for some incremental-analysis instance $\mathcal{I} = (\mathcal{A}, \text{seeds}, d_{\text{old}})$. For any node $n \in N$, we term n as a *consistent node* with respect to d_i, denoted as consistent(n, d_i), if and only if the following holds:

$$\mathtt{IN}_i(n) = \begin{cases} \vartheta e, & \text{if } n = n_0 \\ \displaystyle\bigsqcap_{p \in \mathrm{pred}_P(n)} \mathtt{OUT}_i(p), & \text{otherwise} \end{cases}$$

and

$$\mathtt{OUT}_i(n) = \mathcal{T}_n(\mathtt{IN}_i(n)), \text{where } \mathcal{T}_n = \mathcal{M}(n).$$

Note that when a node is consistent with respect to a dataflow state, its IN and OUT dataflow facts will not change upon recalculation. In contrast, if the OUT dataflow fact of a node changes upon its recalculation, then the successors of the node may become inconsistent with respect to the updated dataflow state.

Definition A.2 (Consistent SCC). Consider a dataflow state $d_i = (\mathtt{IN}_i, \mathtt{OUT}_i)$ for some analysis instance $\mathcal{A} = (P, \vartheta_e)$, or for some incremental-analysis instance $\mathcal{I} = (\mathcal{A}, \text{seeds}, d_{\text{old}})$. Given $\mathrm{scc}_P(k)$, the SCC with index k in the topological sort order of SDG_P, we term $\mathrm{scc}_P(k)$ as a *consistent SCC* with respect to d_i, denoted as $\mathrm{consistentSCC}(k, d_i)$, if and only if the following holds: $\forall n \in \mathrm{sccNodes}_P(k), \mathrm{consistent}(n, d_i)$.

Theorem A.3 (Fixed-point via consistency). *A dataflow state $d_i = (\mathtt{IN}_i, \mathtt{OUT}_i)$ is a fixed-point solution for the analysis instance $\mathcal{A} = (P, \vartheta_e)$, as well as for all the incremental-analysis instances of the form $\mathcal{I} = (\mathcal{A}, *, *)$, if and only if the following holds: $\forall n \in N, \mathrm{consistent}(n, d_i)$.*

Proof. Follows directly from Definitions 4.4 and A.1. □

Definition A.4 (Safe node). Consider a dataflow state $d_i = (\mathtt{IN}_i, \mathtt{OUT}_i)$ for some analysis instance $\mathcal{A} = (P, \vartheta_e)$, or for some incremental-analysis instance $\mathcal{I} = (\mathcal{A}, \text{seeds}, d_{\text{old}})$. Let $d_{\text{mfp}} = (\mathtt{IN}_{\text{mfp}}, \mathtt{OUT}_{\text{mfp}})$ be the MFP solution of $\mathcal{A}$. For any node $n \in N$, we term n as a *safe node* with respect to d_i, denoted as $\mathrm{safe}(n, d_i)$, if and only if the following holds:

$$\mathtt{IN}_{\text{mfp}}(n) \sqsubseteq \mathtt{IN}_i(n), \text{and } \mathtt{OUT}_{\text{mfp}}(n) \sqsubseteq \mathtt{OUT}_i(n)$$

Note that for a given node n, and a dataflow state $d_i = (\mathtt{IN}_i, \mathtt{OUT}_i)$ of analysis instance $\mathcal{A}$ (or $\mathcal{I}$), if $\mathtt{IN}_i(\mathrm{n}) = \top$ and $\mathtt{OUT}_i(n) = \top$, then $\mathrm{safe}(n, d_i)$ trivially holds. Similarly, if d_i is the MFP solution of $\mathcal{A}$ (or $\mathcal{I}$), then $\forall n \in N, \mathrm{safe}(n, d_i)$ trivially holds.

Definition A.5 (Safe SCC). Consider a dataflow state $d_i = (\mathtt{IN}_i, \mathtt{OUT}_i)$ for some analysis instance $\mathcal{A} = (P, \vartheta_e)$, or for some incremental-analysis instance $\mathcal{I} = (\mathcal{A}, \text{seeds}, d_{\text{old}})$. Let $d_{\text{mfp}} = (\mathtt{IN}_{\text{mfp}}, \mathtt{OUT}_{\text{mfp}})$ be the MFP solution of $\mathcal{A}$. Given $\mathrm{scc}_P(k)$, the SCC with index k in the topological sort order of SDG_P, we term $\mathrm{scc}_P(k)$ as a *safe SCC* with respect to d_i, denoted as $\mathrm{safeSCC}(k, d_i)$, if and only if the following holds: $\forall n \in \mathrm{sccNodes}_P(k), \mathrm{safe}(n, d_i)$.

Theorem A.6 (Maximum fixed-point as consistency and safety). *A dataflow state $d_i = (\mathtt{IN}_i, \mathtt{OUT}_i)$ is the maximum fixed-point solution for the analysis instance $\mathcal{A}$, as well as for all the incremental-analysis instances of the form $\mathcal{I} = (\mathcal{A}, *, *)$, if and only if the following holds: $\forall n \in N, \mathrm{consistent}(n, d_i) \land \mathrm{safe}(n, d_i)$.*

Proof. There are two parts to this proof.

Part A: Consistency and safety imply MFP: Assume $\forall n \in N, \mathrm{consistent}(n, d_i) \land \mathrm{safe}(n, d_i)$. Since $\forall n \in N, \mathrm{consistent}(n, d_i)$, from Theorem A.3 we conclude that d_i is a fixed-point solution of $\mathcal{A}$ (or $\mathcal{I}$). Let $d_{\text{mfp}} = (\mathtt{IN}_{\text{mfp}}, \mathtt{OUT}_{\text{mfp}})$ be the MFP solution for $\mathcal{A}$ (or $\mathcal{I}$). Since d_i is a fixed-point solution, from Definition 4.5 of the MFP solution, we know that $\forall n \in N$:

$$\mathtt{IN}_i(n) \sqsubseteq \mathtt{IN}_{\text{mfp}}(n) \land \mathtt{OUT}_i(n) \sqsubseteq \mathtt{OUT}_{\text{mfp}}(n) \tag{1}$$

Since $\forall n \in N, \mathsf{safe}(n, d_i)$, from Definition A.4 of safe nodes we conclude that $\forall n \in N$:

$$\mathtt{IN}_{\text{mfp}}(n) \sqsubseteq \mathtt{IN}_i(n) \land \mathtt{OUT}_{\text{mfp}}(n) \sqsubseteq \mathtt{OUT}_i(n) \tag{2}$$

From (1) and (2), we have, $\mathtt{IN}_i = \mathtt{IN}_{\text{mfp}} \land \mathtt{OUT}_i = \mathtt{OUT}_{\text{mfp}}$, that is, $d_i = d_{\text{mfp}}$, the MFP solution.

Part B: MFP implies consistency and safety: Assume that d_i is the MFP solution for $\mathcal{A}$ (or $\mathcal{I}$). Since an MFP solution is also a fixed-point solution, we conclude from Definition 4.4 that $\forall n \in N$:

$$\mathtt{IN}_i(n) = \begin{cases} \vartheta e, & \text{if } n = n_0 \\ \bigsqcap\limits_{p \in \mathsf{pred}_P(n)} \mathtt{OUT}_i(p), & \text{otherwise} \end{cases}$$

and

$$\mathtt{OUT}_i(n) = \mathcal{T}_n(\mathtt{IN}_i(n)), \text{where } \mathcal{T}_n = \mathcal{M}(n)$$

Hence, by Definition A.1, $\forall n \in N, \mathsf{consistent}(n, d_i)$ holds. Similarly, since d_i is the MFP solution of $\mathcal{A}$ (or $\mathcal{I}$), we conclude from Definition A.4 that $\forall n \in N, \mathsf{safe}(n, d_i)$ holds. Hence proved. □

Corollary A.7 (Maximum fixed-point as consistency and safety per SCC). *Consider an analysis instance $\mathcal{A} = (P, \vartheta_e)$, or some incremental-analysis instance of the form $\mathcal{I} = (\mathcal{A}, *, *)$. Let $|\mathsf{SDG}_P|$ denote the number of SCCs in SDG_P. A dataflow state $d_i = (\mathtt{IN}_i, \mathtt{OUT}_i)$ is the maximum fixed-point solution for $\mathcal{A}$ (or $\mathcal{I}$), if and only if the following holds:*

$$\overset{|\mathsf{SDG}_P|-1}{\underset{k=0}{\forall}} (\mathsf{consistentSCC}(k, d_i) \land \mathsf{safeSCC}(k, d_i))$$

Proof. Follows directly from Theorem A.6 and Definitions A.2 and A.5. □

Lemma A.8 (Consistency and safety with processNode). *Consider an invocation of the helper function* processNode*, with arguments $d_i = (\mathtt{IN}_i, \mathtt{OUT}_i)$, n, and Q to obtain the resulting dataflow state $d_j = (\mathtt{IN}_j, \mathtt{OUT}_j)$. We note that*

- *if $Q = \mathsf{pred}_P(n)$, then $\mathsf{consistent}(n, d_j)$ holds, and*
- *if $\forall q \in Q, \mathsf{safe}(q, d_i)$ is true, then $\mathsf{safe}(n, d_j)$ holds.*

Proof. The first part of the lemma, about consistency of n with respect to d_j when Q is the complete set of predecessors of n, follows directly from Definition A.1, and Fig. 8. We now prove the second part of the lemma.

Let $d_{\text{mfp}} = (\mathtt{IN}_{\text{mfp}}, \mathtt{OUT}_{\text{mfp}})$ be the MFP solution.

$$\begin{aligned} \mathtt{IN}_{\text{mfp}}(n) &= \sqcap_{q \in Q} \mathtt{OUT}_{\text{mfp}}(q) && \text{(from Definition 4.5)} \\ &\sqsubseteq \sqcap_{q \in Q} \mathtt{OUT}_i(q) && \text{(from Definition A.4)} \\ &= \mathtt{IN}_j(n) && \text{(from Fig. 8)} \end{aligned}$$

Hence, $\mathtt{IN}_{\text{mfp}}(n) \sqsubseteq \mathtt{IN}_j(n)$. From Definition A.4 of safe nodes, we conclude that $\mathsf{safe}(n, d_j)$ holds. □

Theorem A.9 (Consistency and safety with CompIDFA). *Consider a dataflow analysis $\mathbb{A} = (\mathcal{L}, \mathcal{F}, \mathcal{M})$, and its analysis instance $\mathcal{A} = (P, \vartheta_e)$, where $P = (N, E, n_0)$. Let $|\mathsf{SDG}_P|$ denote the number of SCCs in SDG_P. If $\mathsf{Sc}_i = <|\mathsf{SDG}_P| - 1, \emptyset, d_i>$ is the resulting* CompIDFA*-state from the fixed-point application of $\mathsf{eval}_{\mathtt{Comp}}$ on $\mathsf{initState}_{\mathtt{Comp}}(\mathcal{A})$, then, $\forall n \in N, \mathsf{consistent}(n, d_i) \land \mathsf{safe}(n, d_i)$.*

Proof. From Theorem A.31, d_i is the MFP solution for $\mathcal{A}$. Hence, from Theorem A.6, we conclude that $\forall n \in N, \mathsf{consistent}(n, d_i) \land \mathsf{safe}(n, d_i)$. □

Corollary A.10 (Consistency and safety per SCC with $\texttt{CompIDFA}$). *Consider a dataflow analysis* $\mathbb{A} = (\mathcal{L}, \mathcal{F}, \mathcal{M})$*, and its analysis instance* $\mathscr{A} = (P, \vartheta_e)$*, where* $P = (N, E, n_0)$*. Let* $|\mathrm{SDG}_P|$ *denote the number of SCCs in* SDG_P*. If* $\mathrm{Sc}_i = <|\mathrm{SDG}_P| - 1, \emptyset, d_i>$ *is the resulting* $\texttt{CompIDFA}$*-state from the fixed-point application of* $\mathrm{eval}_{\mathsf{Comp}}$ *on* $\mathsf{initState}_{\mathsf{Comp}}(\mathscr{A})$*, then,*

$$\overset{|\mathrm{SDG}_P|-1}{\underset{k=0}{\forall}} (\mathsf{consistentSCC}(k, d_i) \wedge \mathsf{safeSCC}(k, d_i))$$

Proof. Follows directly from Theorem A.31 and Corollary A.7. □

Consistency and safety upon program transformations.

Consider a dataflow analysis $\mathbb{A} = (\mathcal{L}, \mathcal{F}, \mathcal{M})$, and its analysis instance $\mathscr{A} = (P, \vartheta_e)$, where $P = (N, E, n_0)$ has been derived upon applying a sequence of one or more transformations on some program $P_{\mathrm{old}} = (N_{\mathrm{old}}, E_{\mathrm{old}}, n_0)$. Let $\mathscr{A}_{\mathrm{old}} = (P_{\mathrm{old}}, \mathcal{M}_{\mathrm{old}}, \vartheta_e)$ be the analysis instance of $\mathbb{A}$ for P_{old}, and $d_{\mathrm{old}} = (\mathtt{IN}_{\mathrm{old}}, \mathtt{OUT}_{\mathrm{old}})$ be the MFP solution of $\mathscr{A}_{\mathrm{old}}$. Let $\mathsf{seeds} = \mathsf{seeds}(P_{\mathrm{old}}, P)$ represent the set of seed nodes from P_{old} to P.

Lemma A.11 (Unaffected SCCs). *Consider an SCC, say* $\mathsf{scc}_P(k)$*, from the SDG of P, such that* $\mathsf{sccNodes}_P(k)$ *does not contain any seed nodes. Let* $d_i = (\mathtt{IN}_i, \mathtt{OUT}_i)$ *be a dataflow state for* $\mathscr{A}$*, such that,* $\forall n \in \mathsf{sccNodes}_P(k), \mathtt{IN}_i(n) = \mathtt{IN}_{\mathrm{old}}(n) \wedge \mathtt{OUT}_i(n) = \mathtt{OUT}_{\mathrm{old}}(n)$*. Let Q be the set of all those nodes that are predecessors of the nodes in* $\mathsf{sccEntryNodes}_P(k)$*, but do not belong to* $\mathsf{sccNodes}_P(k)$*. If each element of Q is consistent and safe with respect to* d_i*, and if the* $\mathtt{OUT}$ *dataflow facts for all these elements match their values in* d_{old}*, then all nodes in* $\mathsf{sccNodes}_P(k)$ *too are consistent and safe in* d_i*.*

Proof. *(Sketch.)* Consider that we reset and recompute the dataflow facts of all nodes in the SCC. Since the incoming dataflow facts to the SCC are consistent and safe, the recomputed fixed-point dataflow facts for the nodes in the SCC too will be consistent and safe. We now argue that the fixed-point results of recomputation will match the results stored in d_{old}.

Since $\mathsf{sccNodes}_P(k)$ does not contain any seed nodes, we note that the set of predecessors for all nodes in the SCC, including the entry-nodes, remain unchanged. Consequently, the dataflow equations for none of the nodes change. Since the incoming dataflow facts ($\mathtt{OUT}$ of the nodes in set Q, and the set Q itself) do not change, we note that recalculation of the dataflow facts for all nodes in the SCC will yield the same result as stored in d_{old}.

We leave a formal proof for this lemma as an exercise for the reader. □

Informally, in the context of program transformations, this lemma states that if there are no seed nodes in an SCC, if the incoming dataflow facts to the SCC have not changed from their fixed-point values since before the transformations, and if the incoming dataflow facts are consistent and safe with respect to some dataflow state after the transformation, then the dataflow facts for the nodes in the SCC too are consistent and safe with respect to that dataflow state. This implies that such an SCC need not be processed to obtain the MFP solution after the transformations.

Consistency and safety for SCCs unreachable from seed nodes.

Consider a dataflow analysis $\mathbb{A}$, and its analysis instance $\mathscr{A} = (P, \vartheta_e)$, where $P = (N, E, n_0)$ has been derived upon applying a sequence of one or more transformations on some program $P_{\mathrm{old}} = (N_{\mathrm{old}}, E_{\mathrm{old}}, n_0)$. Let $\mathsf{seeds} = \mathsf{seeds}(P_{\mathrm{old}}, P)$ represent the set of seed nodes for program transformations from P_{old} to P. Let d_{old} be the MFP solution of some analysis instance $\mathscr{A}_{\mathrm{old}} = (P_{\mathrm{old}}, \mathcal{M}_{\mathrm{old}}, \vartheta_e)$ for $\mathbb{A}$.

Lemma A.12 (Unreachable SCCs). *For an incremental-analysis instance* $\mathcal{I} = (\mathscr{A}, \mathsf{seeds}, d_{\mathrm{old}})$*, assume* $\mathrm{Si}_{MFP} = \mathrm{eval}^{\infty}_{\mathsf{Inc}}(\mathsf{initState}_{\mathtt{IncIDFA}}(\mathcal{I}))$*, is the result of the fixed-point application of* $\mathrm{eval}_{\mathsf{Inc}}$*. Let* d_{mfp} *be the dataflow state in* Si_{MFP}*. In the topological sort order of* SDG_P*, let k be the minimum index* *of any SCC that contains a seed node. We note that the SCCs that are unreachable from the seed nodes are consistent and safe with respect to* d_{mfp}*. Formally,*

$$k = \min_{\forall s \in \text{seeds}} (\mathsf{sccID}_P(s)) \quad \Rightarrow \quad \overset{l<k}{\underset{l=0}{\forall}}, \mathsf{consistentSCC}(l, d_{\text{mfp}}) \land \mathsf{safeSCC}(l, d_{\text{mfp}}) \tag{3}$$

Proof. Let SI_{INIT} be the initial IncIDFA-state of instance $\mathcal{I}$, and let d_{init} be the dataflow state in SI_{INIT}. For all SCCs that are unreachable from the seed node, it directly follows from Lemma A.11 (using strong induction on the index of SCCs), that the SCCs are consistent and safe with respect to d_{init}.

From Definition 4.11, we note that the first SCC to be processed by $\text{eval}_{\mathsf{Inc}}$ is that with index k. From Rule [PhC-FP], and definition of the method firstNodeWithLeastSCCId, we note that the SCCs are processed in the increasing order of their indices. Hence, this ensures that the dataflow facts in d_{mfp} (the fixed-point state) for program nodes in SCCs that are unreachable from the seed nodes, remain unchanged from their values in d_{init}. Since these SCCs are consistent and safe with respect to d_{init}, their safety and consistency hold for d_{mfp} as well. □

Reachability in $\text{eval}_{\mathsf{Inc}}$.
In the rest of the formalism, $\mathbb{N}_0$ denotes the set of all natural numbers and zero. For any $p \in \mathbb{N}_0$, we use ${\text{eval}_{\mathsf{Inc}}}^p(\text{SI}_i)$ to denote p applications of the function $\text{eval}_{\mathsf{Inc}}$ on some IncIDFA-state SI_i. Note that ${\text{eval}_{\mathsf{Inc}}}^0(\text{SI}_i) = \text{SI}_i$. As before, the fixed-point application of $\text{eval}_{\mathsf{Inc}}$ is denoted by ${\text{eval}_{\mathsf{Inc}}}^\infty(\text{SI}_i)$.

Definition A.13 (Reachable IncIDFA*-state).* Consider an incremental-analysis instance, $\mathcal{I}$, as discussed above. An IncIDFA-state, $\text{SI}_i \in \mathcal{S}_I$ is considered as a reachable IncIDFA-state, denoted by $\mathsf{reachable}(\text{SI}_i, \mathcal{I})$, if and only if there exists an interpretation of firstNode and firstNodeWithLeastSCCId, such that, $\exists p \in \mathbb{N}_0, \text{SI}_i = {\text{eval}_{\mathsf{Inc}}}^p(\mathsf{initState}_{\mathsf{IncIDFA}}(\mathcal{I}))$.

Informally, in the context of an incremental-analysis instance $\mathcal{I}$, an IncIDFA-state is considered reachable if and only if it can be obtained upon applying the $\text{eval}_{\mathsf{Inc}}$ function on the initial IncIDFA-state for $\mathcal{I}$ zero or more times.

*Definition A.14 (*SCC-init *state).* Consider an SCC at index k in the topological sort order of the SDG_P. We term an IncIDFA-state as the initial state of the SCC, denoted by $\mathsf{SCC\text{-}init}(k)$, if it is of the form $<k, \mathsf{sccEntryNodes}_P(k), d_i, \emptyset, \emptyset, \emptyset, O_k, \text{PhA}, *>$, where:

$$d_i = (\mathtt{IN}_i, \mathtt{OUT}_i)$$
$$\forall x \in \mathsf{sccExitNodes}_P(k), O_k(x) = \mathtt{OUT}_i(x)$$

Informally, if the SCC-init state for SCC at index k in the topological sort order of the SDG_P is reachable upon zero or more applications of $\text{eval}_{\mathsf{Inc}}$ on the initial IncIDFA-state of an incremental-analysis instance $\mathcal{I}$, then it implies that the SCC is processed by the IncIDFA algorithm. Note that the initial IncIDFA-state is an SCC-init state.

Termination of processing of an SCC by $\text{eval}_{\mathsf{Inc}}$.
We now argue the termination of processing of $\text{eval}_{\mathsf{Inc}}$ when it reaches a state which is SCC-init state for some SCC.

Definition A.15 (SCC-fixed-point states). Consider an SCC at some index k in the topological sort order of the SDG_P. We define three types of fixed-point IncIDFA-states for the SCC, as follows:

- PhA-fixed-point states, which are of the form $<k, \emptyset, *, *, *, *, O_k, \text{PhA}, *>$,
- PhB-fixed-point states, which are of the form $<k, \emptyset, *, *, *, \emptyset, O_k, \text{PhB}, *>$, and
- PhC-fixed-point states, which are of the form $<k, \emptyset, *, \emptyset, \emptyset, \emptyset, O_k, \text{PhC}, *>$.

If a IncIDFA-state, say Si_i is PhA-fixed-point, PhB-fixed-point, or PhC-fixed-point for some SCC with index k, then we denote these facts as isSCC-PhA-FP(Si_i, k), isSCC-PhB-FP(Si_i, k), or isSCC-PhC-FP(Si_i, k), respectively.

Informally, these IncIDFA-states denote the termination of the corresponding phases while processing any SCC.

Lemma A.16 (Termination of PhA). *Consider an incremental-analysis instance, $\mathcal{I}$, as discussed above. If the initial state of an SCC is reachable from the initial* IncIDFA*-state for $\mathcal{I}$, then some* PhA-fixed-point *state is reachable as well for the SCC. Formally,*

$$\text{reachable}(\text{SCC-init}(k), \mathcal{I}) \Rightarrow \exists \text{Si}_i \in \mathcal{S}_I, \text{isSCC-PhA-FP}(\text{Si}_i, k) \wedge \text{reachable}(\text{Si}_i, \mathcal{I})$$

Proof. At each step of application of eval_{Inc}, starting with the state SCC-init(k), either of the two rules [PhA-Mark] or [PhA-Proc] from Fig. 7 will be applicable, until a PhA-fixed-point state is reached. Each application of either of these rules non-deterministically removes one node from the worklist and processes it. The processed node is added to the set S_k. In both these rules, we note that if a node is present in the set S_k, then it is not added back to the worklist. This implies that a node can get added at most once to the worklist. Given the finite number of nodes in $\text{sccNodes}_P(k)$, the worklist will be empty after at most $|\text{sccNodes}_P(k)|$ applications of the eval_{Inc} function, leading to an IncIDFA-state which is a PhA-fixed-point state. Hence proved. □

Lemma A.17 (Termination of PhB). *Consider an incremental-analysis instance, $\mathcal{I}$, as discussed above. If the initial state of an SCC is reachable from the initial* IncIDFA*-state for $\mathcal{I}$, then some* PhB-fixed-point *state is reachable as well for the SCC. Formally,*

$$\text{reachable}(\text{SCC-init}(k), \mathcal{I}) \Rightarrow \exists \text{Si}_i \in \mathcal{S}_I, \text{isSCC-PhB-FP}(\text{Si}_i, k) \wedge \text{reachable}(\text{Si}_i, \mathcal{I})$$

Proof. From Lemma A.16, we note that an IncIDFA-state which is a PhA-fixed-point state is reachable for $\mathcal{I}$. In the next application of eval_{Inc}, rule [PhA-FP] from Fig. 7 is applicable, resulting in some IncIDFA-state of the form $<k, *, *, *, *, \emptyset, *, \text{PhB}, *>$. Starting with that state, the argument for reachability of a PhB-fixed-point state is similar as that used in Lemma A.16, relying on the rule [PhB-Proc] from Fig. 9. □

Lemma A.18 (Termination of PhC). *Consider an incremental-analysis instance, $\mathcal{I}$, as discussed above. If the initial state of an SCC is reachable from the initial* IncIDFA*-state for $\mathcal{I}$, then some* PhC-fixed-point *state is reachable as well for the SCC. Formally,*

$$\text{reachable}(\text{SCC-init}(k), \mathcal{I}) \Rightarrow \exists \text{Si}_i \in \mathcal{S}_I, \text{isSCC-PhC-FP}(\text{Si}_i, k) \wedge \text{reachable}(\text{Si}_i, \mathcal{I})$$

Proof. From Lemma A.17, we note that an IncIDFA-state which is a PhB-fixed-point state is reachable for $\mathcal{I}$. In the next application of eval_{Inc}, rule [PhB-FP] from Fig. 9 is applicable, resulting in an IncIDFA-state of the form $<k, W_k, *, \emptyset, \emptyset, \emptyset, *, \text{PhC}, *>$. If W_k is empty, then a PhC-fixed-point state has been reached. Otherwise, rule [PhC-Proc] from Fig. 10 is applicable. Note that a node may get added to the worklist at most as many times as is the height of the lattice [87, 88]. Hence the worklist will get empty after a finite number of applications of the rule [PhC-Proc], resulting in a PhC-fixed-point state. Hence proved. □

From Lemmas A.16, A.17, and A.18, we infer that if an SCC is processed by eval_{Inc}, then the processing eventually terminates. We now prove that upon termination of processing of an SCC, the SCC is rendered consistent and safe with the current dataflow state.

Consistency and safety of SCCs processed by eval_{Inc}.
Now, we gradually discuss the guarantees of consistency and safety for nodes in the SCCs that are processed by eval_{Inc}.

Lemma A.19 (PhB leads to consistent or underapproximated nodes). *Consider an incremental-analysis instance $\mathcal{I} = (\mathcal{A}, \text{seeds}, d_{\text{old}})$, as discussed above. Assume that* $\mathsf{SCC\text{-}init}(k)$*, the initial state for SCC at index k in the topological sort order of* SDG_P*, is a reachable* IncIDFA*-state for the incremental-analysis instance* $\mathcal{I}$*. Let* $\text{Si}_i = <k, \emptyset, d_i, S_k, U_k, \emptyset, O_k, \text{PhB}, W_g>$ *be the* PhB-fixed-point *state that is reached upon zero or more applications of* eval_{Inc} *on* $\mathsf{SCC\text{-}init}(k)$*. We note that in* Si_i*,* $\forall n \in \text{sccNodes}_P(k), \mathsf{consistent}(n, d_i) \lor n \in U_k$.

Proof. Consider a node, say n, in the set $\text{sccNodes}_P(k)$. Two cases arise on the basis of whether node n was present in the intra-SCC worklist, W_k, in any IncIDFA-state between $\mathsf{SCC\text{-}init}(k)$ and Si_i:

Case A: Node n was present in W_k in some IncIDFA-state. Three sub-cases arise on the basis of the evaluation rule that was used for processing n in PhA or PhB:

Case A.i: [PhA-Mark] In this case, all predecessors of node n are present in the set M_k. As per Rules [PhA-Mark] and [PhA-Proc], a node gets added to the set M_k only if its OUT dataflow fact does not change upon its processing. Hence, the OUT of none of the predecessors of node n changed. Since n is not a seed node, its set of predecessors ($\text{pred}_P(n)$) too remains unchanged. Hence, from Definition A.1, $\mathsf{consistent}(n, d_i)$ holds.

Case A.ii: [PhA-Proc] Assume that d' is the dataflow state in the IncIDFA-state obtained upon application of this rule. In this rule, method processNode gets invoked on n. If the set Q_p is same as $\text{pred}_P(n)$, then $\mathsf{consistent}(n, d')$ holds, as per Lemma A.8. Otherwise, as per this rule, node n gets added to the set $\mathsf{U}_k{}'$ of the resulting IncIDFA-state.

Case A.iii: [PhB-Proc] This case is similar to *Case A.iii* above.

Case B: Node n was never added to W_k. Clearly, node n is not a seed node. In this case, for each predecessor of node n in the same SCC, say $p \in \text{sameSccPred}_P(n)$, there are two possibilities: (i) Node p was not processed in any IncIDFA-state between $\mathsf{SCC\text{-}init}(k)$ and Si_i. (ii) Node p was processed as per Rule [PhB-Proc], but its processing resulted in no change in its OUT dataflow fact. In both these cases, since the OUT dataflow fact does not change for any predecessor of n, and since n is clearly not a seed node, from Definition A.1, $\mathsf{consistent}(n, d_i)$ holds.

Hence proved. □

Lemma A.20 (Set M_k is a subset of S_k). *Consider an incremental-analysis instance* $\mathcal{I}$*, as discussed above. Assume that* $\mathsf{SCC\text{-}init}(k)$ *is a reachable* IncIDFA*-state for* $\mathcal{I}$*. Let* Si_a *be the* PhA-fixed-point *state that is reached upon zero or more applications of* eval_{Inc} *on* $\mathsf{SCC\text{-}init}(k)$*. We note that for all* IncIDFA*-state* $\text{Si}_i = <k, \emptyset, d_i, S_k, U_k, M_k, O_k, \text{PhA}, W_g>$*, that lie between* $\mathsf{SCC\text{-}init}(k)$ *and* Si_a *(both inclusive), the following holds:* $\mathsf{M}_k \subseteq S_k$.

Proof. Elements in set M_k are added only in case of Rules [PhA-Mark] and [PhA-Proc]. In both these cases, the element gets added to set S_k as well. Hence, $\mathsf{M}_k \subseteq \mathsf{S}_k$. □

Lemma A.21 (Set S_k is safe). *Consider an incremental-analysis instance* $\mathcal{I}$*, as discussed above. Assume that* $\mathsf{SCC\text{-}init}(k)$ *is a reachable* IncIDFA*-state for* $\mathcal{I}$*. Further, assume that all SCCs with index less than k are consistent and safe. Let* Si_b *be the* PhB-fixed-point *state that is reached upon zero or more applications of* eval_{Inc} *on* $\mathsf{SCC\text{-}init}(k)$*. We note that for all* IncIDFA*-state* $\text{Si}_i = <k, \emptyset, d_i, S_k, U_k, M_k, O_k, *, W_g>$*, that lie between* $\mathsf{SCC\text{-}init}(k)$ *and* Si_b *(both inclusive), the following holds:* $\forall n \in S_k, \mathsf{safe}(n, d_i)$.

Proof. Consider an integer $p \in \mathbb{N}_0$, such that $\text{eval}_{\text{Inc}}^{p}(\mathsf{SCC\text{-}init}(k)) = \text{Si}_b$. We prove that the lemma holds by using induction on the value of p.

Base case: Assume $p = 0$. In this case, the IncIDFA-state obtained is SCC-init(k). In this state, the set S_k is empty. Hence the lemma trivially holds.

Inductive hypothesis: Consider an IncIDFA-state, say $\mathrm{S}\textsc{i}_q$, such that $\mathrm{S}\textsc{i}_q = \langle k, \emptyset, d_q, \mathsf{S}_q, \mathsf{U}_q, \mathsf{M}_q, \mathcal{O}_k, \textsc{PhB}, W_g \rangle = \mathrm{eval}^{q}_{\mathsf{Inc}}(\text{SCC-init}(k))$. For all nodes $n \in \mathsf{S}_q$, $\mathsf{safe}(n, d_q)$ holds.

Inductive step: Consider an IncIDFA-state, say $\mathrm{S}\textsc{i}_{q+1}$, such that $\mathrm{S}\textsc{i}_{q+1} = \langle k, \emptyset, d_{q+1}, \mathsf{S}_{q+1}, \mathsf{U}_{q+1}, \mathsf{M}_{q+1}, \mathcal{O}_k, \textsc{PhB}, W_g \rangle = \mathrm{eval}^{(q+1)}_{\mathsf{Inc}}(\text{SCC-init}(k))$. Now, we need to prove that for all nodes $n \in \mathsf{S}_{q+1}$, $\mathsf{safe}(n, d_{q+1})$ holds. Four cases arise, on the basis of the evaluation rule that was used to take this step:

[PhA-Mark] Let n be the node returned by the firstNode method upon application of this rule. In this case, all predecessors of node n belong to the set M_k, which implies that they are also present in the set S_q (from Lemma A.20). Furthermore, the OUT dataflow facts did not change for any of these predecessors. Since the dataflow fact of n are not recalculated, it is clear that they are safe with respect to the current dataflow state, d_{q+1}. Since n is the only new element in S_{q+1}, it proves that all elements of S_{q+1} are safe with respect to d_{q+1}.

[PhA-Proc] Let n be the node returned by the firstNode method upon application of this rule. The set Q_p passed to the method processNode contains two parts: (i) predecessors of n that belong to previous SCCs (which are given to be safe), and (ii) predecessors of n that belong to this SCC and are present in S_q (that is, are safe). Since all elements of Q_p are safe with respect to d_q, from Lemma A.8, we note that node n is safe with respect to d_{q+1}.

[PhA-FP] In this rule, no changes are made to S_q to obtain S_{q+1}. Hence, all nodes in S_{q+1} are safe with respect to d_{q+1}.

[PhB-Proc] The reasoning for this case is similar to that for Rule [PhA-Proc].

Hence proved.

□

Lemma A.22 (PhB leads to safe nodes). *Consider an incremental-analysis instance $\mathcal{I}$, as discussed above. Assume that* SCC-init(k) *is a reachable* IncIDFA-*state for $\mathcal{I}$. Further, assume that all SCCs with index less than k are consistent and safe. Let $\mathrm{S}\textsc{i}_b = \langle k, \emptyset, d_b, \mathsf{S}_k, \mathsf{U}_k, \emptyset, \mathcal{O}_k, \textsc{PhB}, W_g \rangle$ be the* PhB-fixed-point *state that is reached upon zero or more applications of* $\mathrm{eval}_{\mathsf{Inc}}$ *on* SCC-init(k)*. We note that* $\forall n \in \mathsf{sccNodes}_{\mathcal{P}}(k), \mathsf{safe}(n, d_b)$.

Proof. Consider a node, say n, in the set $\mathsf{sccNodes}_{\mathcal{P}}(k)$. Two cases arise on the basis of whether node n was present in the intra-SCC worklist, W_k, in any IncIDFA-state between SCC-init(k) and $\mathrm{S}\textsc{i}_b$:

Case A: Node n was present in W_k in some state. The evaluation rules that could be used for processing n in PhA or PhB are Rules [PhA-Mark], [PhA-Proc], or [PhB-Proc]. In all these rules, node n gets added to the set S_k. From Lemma A.21, we conclude, $\mathsf{safe}(n, d_b)$.

Case B: Node n was never added to W_k. Clearly, node n is not a seed node. In this case, for each predecessor of node n in the same SCC, say $p \in \mathsf{sameSccPred}_{\mathcal{P}}(n)$, there are two possibilities: (i) Node p was not processed in any IncIDFA-state between SCC-init(k) and $\mathrm{S}\textsc{i}_b$. (ii) Node p was processed as per Rule [PhB-Proc], but its processing resulted in no change in its OUT dataflow fact. In both these cases, since the OUT dataflow fact does not change for any predecessor of n, and since n is clearly not a seed node, from Definition A.4, $\mathsf{safe}(n, d_b)$ holds.

□

Lemma A.23 (PhC leads to consistent and safe nodes). *Consider an incremental-analysis instance $\mathcal{I}$, as discussed above. Assume that* SCC-init(k) *is a reachable* IncIDFA-*state for*

$\mathcal{I}$. Further, assume that all SCCs with index less than k are consistent and safe. Let $\text{Si}_c = {<}k, \emptyset, d_c, \emptyset, \emptyset, \emptyset, O_k, \text{PhC}, W_g{>}$ *be the* PhC-fixed-point *state that is reached upon zero or more applications on the* PhB-fixed-point *state, say* Si_b*, for the SCC. We note that* $\forall n \in \text{sccNodes}_P(k), \text{safe}(n, d_c) \wedge \text{consistent}(n, d_c)$.

Proof. Note that in the IncIDFA-state Si_b, all the nodes are safe, as per Lemma A.22. For all IncIDFA-states between Si_b and Si_c, the only rule applicable is Rule [PhC-Proc]. Consider any such IncIDFA-state, say Si_i. Let n be the node that was returned by method firstNode. Note that since the helper method processNode is invoked with only the safe predecessors of n (which, in this case, are all the predecessors of n), from Lemma A.8, node n is safe in $\text{eval}_{\text{Inc}}(\text{Si}_i)$. Hence, for all IncIDFA-state between Si_b and Si_c (both inclusive), we conclude that $\forall n \in \text{sccNodes}_P(k), \text{safe}(n, d_c)$.

From Lemma A.19, in IncIDFA-state Si_b, we note that all nodes in $\text{sccNodes}_P(k)$ are either consistent, or are present in the set U_b. From Rule [PhB-FP], note that the intra-SCC worklist, W_k, for PhC is populated with U_b. Hence, in IncIDFA-state Si_b, all nodes in $\text{sccNodes}_P(k)$ are present in W_k, or are consistent. It is easy to see that this property holds for each IncIDFA-state between Si_b and Si_c: At each step, only Rule [PhC-Proc] is applicable. With this rule, the node, say n, that is taken from the worklist, is processed using processNode with the complete set of predecessors of n (that is, $\text{pred}_P(n)$). Hence, from Lemma A.8, we know that node n is consistent in the resulting IncIDFA-state. If the OUT dataflow facts of node n changes, its successors may become inconsistent with the resulting IncIDFA-state, and hence are added back to W_k. Hence, we conclude that since W_k is empty in the IncIDFA-stateSi_c, $\forall n \in \text{sccNodes}_P(k), \text{consistent}(n, d_c)$.

Hence proved. □

Corollary A.24 (eval_{Inc} leads to consistent and safe SCC). *Consider an incremental-analysis instance $\mathcal{I}$, as discussed above. Assume that* SCC-init(k) *is a reachable* IncIDFA-*state for $\mathcal{I}$. Further, assume that all SCCs with index less than k are consistent and safe. Let* $\text{Si}_c = {<}k, \emptyset, d_c, \emptyset, \emptyset, \emptyset, O_k, \text{PhC}, W_g{>}$ *be the* PhC-fixed-point *state that is reached upon zero or more applications on the* PhB-fixed-point *state, say* Si_b*, for the SCC. We note that* $\text{safeSCC}(k, d_i) \wedge \text{consistentSCC}(k, d_i)$ *hold.*

Proof. Follows directly from Lemma A.23 and Definitions A.5 and A.2. □

Consistency and safety of SCCs unprocessed by eval_{Inc}.

Lemma A.25. *Consider an incremental-analysis instance $\mathcal{I}$, as discussed above. Assume* $\text{Si}_{MFP} = \text{eval}^{\infty}_{\text{Inc}}(\text{initState}_{\text{IncIDFA}}(\mathcal{I}))$*, is the result of the fixed-point application of* eval_{Inc}*. Let d_{mfp} be the dataflow state in* Si_{MFP}*. In the topological sort order of* SDG_P*, let k be the minimum index of any SCC that contains a seed node. Assume an SCC, with index k, such that* SCC-init(k) *is* not *reachable from $\mathcal{I}$. We note that* $\text{safeSCC}(k, d_{\text{mfp}})$ *and* $\text{consistentSCC}(k, d_{\text{mfp}})$ *hold.*

Proof. Since SCC with index k has not been processed by eval_{Inc}, it is clear that it does not contain any seed nodes. Furthermore, there must not have been any applications of Rule [PhC-FP], where the OUT of any of the predecessors of any entry node of $\text{scc}_P(k)$ changed (else $\text{scc}_P(k)$ would have been processed). Consequently, from Lemma A.11, we conclude that $\text{safeSCC}(k, d_{\text{mfp}})$ and $\text{consistentSCC}(k, d_{\text{mfp}})$ hold. □

Consistency and safety of all SCCs with eval_{Inc}.

Theorem A.26. *Consider a dataflow analysis $\mathbb{A}$, and its analysis instance $\mathcal{A} = (P, \vartheta_e)$, where $P = (N, E, n_0)$ has been derived upon applying a sequence of one or more transformations on some program $P_{\text{old}} = (N_{\text{old}}, E_{\text{old}}, n_0)$. Let $\mathcal{A}_{\text{old}} = (P_{\text{old}}, \mathcal{M}_{\text{old}}, \vartheta_e)$ be an analysis instance of $\mathbb{A}$ for P_{old}, and let d_{old} be its MFP solution. Let* seeds = seeds(P_{old}, P) *represent the set of seed nodes for program transformations from P_{old} to P. Given the incremental-analysis instance $\mathcal{I} = (\mathcal{A}, \text{seeds}, d_{\text{old}})$, let* $\text{Si}_{MFP} = \text{eval}^{\infty}_{\text{Inc}}(\text{initState}_{\texttt{IncIDFA}}(\mathcal{I}))$ *be the result of fixed-point application of the* eval_{Inc} *for $\mathcal{I}$. Let d_{mfp} be the dataflow state in Si_{MFP}. The following holds:*

$$\overset{k<|\text{SDG}_P|}{\underset{k=0}{\forall}}, \text{consistentSCC}(k, d_{\text{mfp}}) \wedge \text{safeSCC}(k, d_{\text{mfp}})$$

Proof. In the topological sort order of SDG_P, let m be the minimum index of any SCC that contains a seed node. For all SCCs with index less than m, the proof follows directly from Lemma A.12. For other SCCs, we prove the theorem by strong induction on their indices.

Base case: For $\text{scc}_P(m)$. The initial IncIDFA-state, as per Definition 4.11, is clearly SCC-init(m). From Lemma A.18, we note that the PhC-fixed-point of $\text{scc}_P(m)$ is reachable. From Corollary A.24, we conclude that consistentSCC(m, d_{mfp}) and safeSCC(m, d_{mfp}) hold.

Induction hypothesis: Assume that $\forall l, m < l < (k-1)$ consistentSCC(l, d_{mfp}) and safeSCC(l, d_{mfp}) holds.

Inductive step: To prove that consistentSCC(k, d_{mfp}) and safeSCC(k, d_{mfp}) holds. On the basis of whether $\text{scc}_P(k)$ has been processed by eval_{Inc}, two cases arise:

Case A: $\text{scc}_P(k)$ has been processed by eval_{Inc}. In this case, the proof follows directly from Corollary A.24.

Case B: $\text{scc}_P(k)$ has not been processed by eval_{Inc}. In this case, the proof follows directly from Lemma A.25.

Hence proved.

□

Corollary A.27. *Consider a dataflow analysis $\mathbb{A}$, and its analysis instance $\mathcal{A} = (P, \vartheta_e)$, where $P = (N, E, n_0)$ has been derived upon applying a sequence of one or more transformations on some program $P_{\text{old}} = (N_{\text{old}}, E_{\text{old}}, n_0)$. Let $\mathcal{A}_{\text{old}} = (P_{\text{old}}, \mathcal{M}_{\text{old}}, \vartheta_e)$ be an analysis instance of $\mathbb{A}$ for P_{old}, and let d_{old} be its MFP solution. Let* seeds = seeds(P_{old}, P) *represent the set of seed nodes for program transformations from P_{old} to P. Given the incremental-analysis instance $\mathcal{I} = (\mathcal{A}, \text{seeds}, d_{\text{old}})$, let* $\text{Si}_{MFP} = \text{eval}^{\infty}_{\text{Inc}}(\text{initState}_{\texttt{IncIDFA}}(\mathcal{I}))$ *be the result of fixed-point application of the* eval_{Inc} *for $\mathcal{I}$. Let d_{mfp} be the dataflow state in Si_{MFP}. The following holds: $d_{\text{mfp}} = \text{mfp}(\mathcal{A})$.*

Proof. Follows directly from Theorem A.26, and Corollary A.7. □

A.2 Formal Model for CompIDFA

In this section, we formalize the traditional exhaustive iterative dataflow analysis algorithm CompIDFA.

*Definition A.28 (*CompIDFA*-state).* In the context of CompIDFA, at any point during the application of the algorithm, we define the analysis state, termed CompIDFA-*state*, as a 3-tuple of the form $\text{Sc}_i = {<}k, W_k, d{>}$, where:

- k is the index of the current SCC being processed by the analysis, in the topological sort order (0-indexed) of SDG_P, the SDG of the program being analyzed.
- W_k denotes an intra-SCC worklist, containing those nodes of the current SCC for which the dataflow facts may require recalculation.
- d represents the current dataflow state.

$$\frac{\begin{array}{c}\mathrm{Sc} = \texttt{<}k, W_k, d\texttt{>} \qquad W_k \neq \texttt{<>} \qquad n = \mathrm{firstNode}(W_k) \\ d = (\mathtt{IN}, \mathtt{OUT}) \quad d' = (\mathtt{IN}', \mathtt{OUT}') = \mathrm{processNode}(d, n, \mathrm{pred}_P(n)) \\ W_k' = \begin{cases} (W_k \setminus \{n\}) \cup \mathrm{sameSccSucc}_P(n), & \text{if } \mathtt{OUT}'(n) \neq \mathtt{OUT}(n) \\ (W_k \setminus \{n\}), & \text{otherwise} \end{cases}\end{array}}{\mathbf{eval_{Comp}}(\mathrm{Sc}) \Rightarrow \texttt{<}k, W_k', d'\texttt{>}}\;[\text{Comp-Proc}]$$

$$\frac{\mathrm{Sc} = \texttt{<}k, \texttt{<>}, d\texttt{>} \quad k \leq (|\mathrm{SDG}_P| - 2)}{\mathbf{eval_{Comp}}(\mathrm{Sc}) \Rightarrow \texttt{<}k+1, \mathrm{sccNodes}_P(k+1), d\texttt{>}}\;[\text{Comp-SCC-FP}]$$

$$\frac{\mathrm{Sc} = \texttt{<}k, \texttt{<>}, d\texttt{>} \quad k = (|\mathrm{SDG}_P| - 1)}{\mathbf{eval_{Comp}}(\mathrm{Sc}) \Rightarrow \mathrm{Sc}}\;[\text{Comp-Global-FP}]$$

Fig. 17. Evaluation rules for the $\mathrm{eval_{Comp}}$ function.

We denote the set of all CompIDFA-states by $\mathcal{S}_C$.

Definition A.29 (Initial CompIDFA*-state).* Consider a dataflow analysis $\mathbb{A} = (\mathcal{L}, \mathcal{F}, \mathcal{M})$, and its analysis instance $\mathcal{A} = (P, \vartheta_e)$, where $P = (N, E, n_0)$. In the context of analysis instance $\mathcal{A}$, we define the initial CompIDFA-state as $\mathrm{initState_{Comp}}(\mathcal{A}) = \texttt{<}0, \mathrm{sccNodes}_P(0), d_{\mathrm{init}}\texttt{>}$, where

$$d_{\mathrm{init}} = (\mathtt{IN}_{\mathrm{init}}, \mathtt{OUT}_{\mathrm{init}})$$
$$\forall n \in N, \mathtt{IN}_{\mathrm{init}}(n) = \begin{cases} \vartheta_e, & \text{if } n = n_0 \\ \top, & \text{otherwise} \end{cases}$$
$$\forall n \in N, \mathtt{OUT}_{\mathrm{init}}(n) = \top$$

Definition A.30 ($\mathrm{eval_{Comp}}$ *function).* We formally represent the CompIDFA algorithm using a function $\mathrm{eval_{Comp}} : \mathcal{S}_C \longrightarrow \mathcal{S}_C$, which takes the current CompIDFA-state, and performs one step of the CompIDFA algorithm to generate the next CompIDFA-state.
For a given analysis instance $\mathcal{A}$, the application of CompIDFA algorithm can be seen as a fixed-point application of the $\mathrm{eval_{Comp}}$ function, denoted by $\mathrm{eval_{Comp}}^{\infty}$, on the initial CompIDFA-state, $\mathrm{initState_{Comp}}(\mathcal{A})$.

Fig. 17 defines the evaluation rules for the $\mathrm{eval_{Comp}}$ function, discussed next.

[Comp-Proc] When the intra-SCC worklist, W_k, is not empty in the current CompIDFA-state, a node is removed (non-deterministically, using the operation firstNode) from the worklist, and its dataflow facts are recalculated by invoking the helper method processNode. If recalculation resulted in a change in the OUT dataflow fact for n, then those successors of n that belong to the same SCC as that of n are added to the intra-SCC worklist, W_k. Note that the order in which nodes are taken from an intra-SCC worklist does not change the final fixed-point solution obtained for that SCC.

[Comp-SCC-FP] When the intra-SCC worklist, W_k, is empty in the current CompIDFA-state, then the state represents a local fixed-point for the nodes in the current SCC. If the current SCC is not the last SCC in the topological sort order of SDG_P, then the index information for the SCC being processed is incremented by one. Further, all nodes that belong to the immediately succeeding SCC in the topological sort, are added to the intra-SCC worklist.

[Comp-Global-FP] If the intra-SCC worklist, W_k, is empty in the current CompIDFA-state, and if the current SCC is the last SCC in the topological sort order of SDG_P, then a global fixed-point has been reached for the application of $\text{eval}_{\text{Comp}}$.

Theorem A.31 (Correctness of $\text{eval}_{\text{Comp}}$). *For a given analysis instance $\mathcal{A} = (P, \vartheta_e)$, a fixed-point application of the* $\text{eval}_{\text{Comp}}$ *function on the initial* `CompIDFA`*-state for $\mathcal{A}$, converges to a* `CompIDFA`*-state that contains the maximum fixed-point solution for $\mathcal{A}$. Formally,*

$$\text{eval}_{\text{Comp}}{}^{\infty}(\text{initState}_{\text{Comp}}(\mathcal{A})) = <|\text{SDG}_P| - 1, <>, d_{\text{mfp}}> \quad \Leftrightarrow \quad d_{\text{mfp}} = \text{mfp}(\mathcal{A})$$

Proof. The proofs for convergence and correctness of standard iterative dataflow analysis algorithm, employed by the function $\text{eval}_{\text{Comp}}$, are standard [87, 88], and hence omitted. □

B Generic Incremental IDFA

In this section, we present `IncIDFA`, a generic, incremental, flow-sensitive, interprocedural dataflow algorithm, which can be used to derive an incremental version of any arbitrary monotone iterative dataflow analysis. `IncIDFA` can be instantiated to write any IDFA-based analysis (for example, reaching-definitions analysis, liveness analysis, and so on), including those analyses that are non-distributive (such as points-to analysis and constant-propagation analysis). `IncIDFA` does not require the dataflow analysis writer to provide any details that are specific to incrementalization of the analysis.

B.1 Design of `IncIDFA`

Given a program P, for any monotone dataflow analysis X, the traditional generic iterative dataflow analysis algorithm (`CompIDFA`) computes the *dataflow solution* (the `IN` and `OUT` maps) of X for P, by taking as input (i) the lattice $\mathcal{L}$ of X, and (ii) the flow-function map $\mathcal{F}$ (that returns the flow function $\mathcal{F}_n$ for every program node n) for X. Without loss of generality, in this manuscript, we assume that each program is represented as a graph (for example, control-flow graph, or super-graph; see Section 2).

Say P_i be a program under compilation whose dataflow solution ($\texttt{IN}_i$ and $\texttt{OUT}_i$ maps) has been computed for a monotone dataflow analysis X. Consider a modified program P_j, obtained by performing a series of compiler transformations on P_i. For each such analysis X, the goal of our proposed algorithm is to compute the modified dataflow solution of X for P_j by incrementally updating the prior computed dataflow solution ($\texttt{IN}_i$ and $\texttt{OUT}_i$). Such incremental updates would require, in addition to other arguments, the information about the overall differences from P_i to P_j. For simplicity, one can assume that these differences are given in terms of sets of (i) nodes added (addedNodes), (ii) nodes removed (removedNodes), (iii) edges added (addedEdges), and (iv) edges removed (removedEdges) – collectively called as the *change-set.*

We take these requirements into consideration and propose a generic incremental iterative dataflow analysis algorithm, termed `IncIDFA`. For obtaining the dataflow solution of X for P_j, in addition to the arguments that are required by `CompIDFA` (namely, $\mathcal{L}$, $\mathcal{F}$ and P_j), `IncIDFA` needs only the following additional arguments: the old dataflow solution (`IN` and `OUT` for P_i) and the change-set (from P_i to P_j).

B.2 `IncIDFA` Algorithm

In this section, we describe our proposed worklist-based algorithm (`IncIDFA`) that works on the SCC decomposition of the program's super-graph (see Section 2); hereafter, we call this graph as the SCC-graph. Without loss of generality, we discuss this algorithm for forward IDFAs; the algorithm for backward IDFAs can be derived similarly. The key intuition of this algorithm is derived from Proposal 3 (Optimized Init-Restart-SCC) discussed in Section 3.4.

The `IncIDFA` algorithm is shown in Fig. 18. It starts by invoking the function `obtainSeedNodes` (see Section 2) to process the change-set (chSet) to obtain the set of *seed* program-nodes in the

```
1  Function IncIDFA($\mathcal{L}$, $\mathcal{F}$, $P$, chSet, IN, OUT) /* Modifies IN and OUT */
2      // $\mathcal{L}$: lattice for the analysis; $\mathcal{F}$: flow function map for the analysis
3      // $P$: modified program as per which the analysis results have to be updated
4      // chSet: change-set denoting the differences from $P_{old}$ to $P$
5      // IN and OUT: dataflow solution of the analysis for some program $P_{old}$
6      OrderedSet globalWL = obtainSeedNodes(chSet, $P$);
7      while globalWL $\neq \emptyset$ do processOneSCC($\mathcal{L}$, $\mathcal{F}$, IN, OUT, globalWL) ;
8  Function processOneSCC($\mathcal{L}$, $\mathcal{F}$, IN, OUT, globalWL) /* Modifies IN, OUT, and globalWL */
9      sccID = globalWL.peekFirst().getSCCId(); // SCC ID of the first node in globalWL
10     /* Step I: Extract all seed nodes belonging to this SCC. */
11     currSCCSeeds = globalWL.removeAllWithId(sccID);
12     /* Step II: Store the stale OUT information of exit nodes of this SCC. */
13     foreach $n \in$ getExits(sccID) do oldOUT($n$) = OUT($n$).clone();
14     /* Step III: (Initialization-step): under-approximation for the current SCC. */
15     safeNodes = $\emptyset$; underApprox = $\emptyset$;
16     if currSCCSeeds $\not\subseteq$ getEntries(sccID) then
17         underApproximate($\mathcal{L}$, $\mathcal{F}$, IN, OUT, safeNodes, underApprox, currSCCSeeds, getEntries(sccID), true);
18     underApproximate($\mathcal{L}$, $\mathcal{F}$, IN, OUT, safeNodes, underApprox, currSCCSeeds, currSCCSeeds.clone(), false);
19     /* Step IV: (Stabilization-step): get MFP solution for the current SCC. */
20     stabilize($\mathcal{L}$, $\mathcal{F}$, $P$, IN, OUT, underApprox);
21     /* Step V: Add required successors from other SCCs to globalWL */
22     foreach $n \in$ getExits(sccID) do
23         if oldOUT($n$) ! = OUT($n$) then
24             globalWL = globalWL $\bigcup$ (succ($n$) $\setminus$ getSCCNodes(sccID));
```

Fig. 18. Algorithm for incremental update of flow-facts, for a forward analysis on serial programs.

current program P. The ordered-set globalWL is kept sorted by the SCC ids of the program-nodes, to ensure that the affected SCCs are processed as per the topological sort order of the SCCs in the SCC-graph. Besides the standard operations like peekFirst, the ordered-set globalWL supports an additional method removeAllWithId, which given an SCC id as an argument, removes (and returns) the set of all the elements with the given SCC id.

The method IncIDFA processes one SCC at a time by invoking the method processOneSCC, which in turn processes each SCC in five steps, in order to implement the initialization-step and stabilization-step discussed in Proposal 3:

Step I: For the SCC being processed, we first extract (in currSCCSeeds) all those program-nodes from globalWL that belong to this SCC. These program-nodes are the directly impacted nodes as a result of either (i) any program-changes performed within this SCC, or (ii) any changes in the incoming flowmaps from one or more of the predecessor SCCs.

Step II: Note that during stabilization of dataflow solution within this SCC, the OUT flowmaps of any of the program-nodes may temporarily be under-approximated before they stabilize back to their fixed-point values. In many cases, we observe that the initial values (before the initialization-step) of the flowmaps for certain program-nodes match the fixed-point values at the end of the stabilization-step. When such program-nodes are also the exit nodes of the SCC, the standard IDFA approach may unnecessarily mark their successor program-nodes (from successor SCCs) to be processed, when the OUT flowmaps of the exit nodes change temporarily. This may result in redundant processing of successor SCCs. To avoid this issue, when we process a program-node, we do not add the inter-SCC successor program-nodes to the worklist. Instead, in Step II, we take a snapshot of the OUT flowmaps of all the exit nodes (obtained through getExits, Line 13), and compare it with the final fixed-point values (in Step V), to decide if any inter-SCC successor program-nodes need to be added to globalWL.

```
1  Function underApproximate($\mathcal{L}$, $\mathcal{F}$, IN, OUT, safeNodes, underApprox, currSCCSeeds, intraSCCWL,
     unconditionallyAdd) /* Modifies IN, OUT, safeNodes, underApprox, and intraSCCWL */
2      // safeNodes: set of nodes that have been processed during the initialization-step
3      // underApprox: set of nodes for which at least one predecessor was ignored during the initialization-step
4      // unconditionallyAdd: true, when the successors need to be added unconditionally to the worklist
5      unchangedOUTNodes = $\emptyset$;
6      while intraSCCWL is not empty do
7          $n$ = intraSCCWL.removeFirst();
8          if $n \in$ safeNodes then continue;
9          boolean hasOUTChanged = false;
10         if pred($n$) $\subseteq$ unchangedOUTNodes then unchangedOUTNodes $\bigcup$ = {n};
11         else
12             safePreds = $\{p : p \in$ pred($n$) && ($p$.getSCCId() $\neq$ $n$.getSCCId() || $p \in$ safeNodes)$\}$;
13             if safePreds ! = pred($n$) then underApprox $\cup = \{n\}$ ;
14             hasOUTChanged = processNode($n$, $\mathcal{L}$, $\mathcal{F}$, IN, OUT, safePreds);
15             if hasOUTChanged == false then unchangedOUTNodes $\bigcup$ = {n};
16         safeNodes $\bigcup = \{n\}$;
17         skipSuccs = safeNodes $\bigcup$ currSCCSeeds ;
18         if unconditionallyAdd || hasOUTChanged then intraSCCWL $\bigcup$ = (intraSCCSucc($n$) \ skipSuccs);

19 Function stabilize($\mathcal{L}$, $\mathcal{F}$, IN, OUT, intraSCCWL) /* Modifies IN, OUT, and intraSCCWL */
20     while intraSCCWL is not empty do
21         $n$ = intraSCCWL.removeFirst();
22         boolean hasOUTChanged = processNode($n$, $\mathcal{L}$, $\mathcal{F}$, IN, OUT, pred($n$));
23         if hasOUTChanged then intraSCCWL $\bigcup$ = intraSCCSucc($n$);
```

Fig. 19. Definitions for underApproximate and stabilize methods used while processing an SCC.

```
1  Function processNode($n$, $\mathcal{L}$, $\mathcal{F}$, IN, OUT, safePreds): boolean /* Modifies IN and OUT */
2      // Returns true if the OUT flowmap of $n$ has changed
3      // safePreds: set of predecessors whose OUT is a safe initial estimate
4      if safePreds == $\emptyset$ then IN($n$) = $\top$; else IN($n$) = $\bigsqcap_{p \in \text{safePreds}}$ OUT($p$);
5      OUT($n$) = $\mathcal{F}_n$(IN($n$));
6      if OUT($n$) has changed then return true;
7      else return false;
```

Fig. 20. Algorithm to recalculate IN and OUT of a program node.

Step III: (Initialization-step) We define a program-node to be *safe* (as discussed in Proposal 3), if (i) the program-node is not reachable from any seed node, or (ii) both its IN and OUT flowmaps are safe (i.e., are *safe-initial-estimates*). We define the IN flowmap of a node as safe, if (i) the flowmap maps all domain-elements to $\top$, or (ii) it has been computed using the OUT flowmaps of its predecessors that are safe. Similarly, we define the OUT flowmaps of a node as safe, if (i) the flowmap maps all domain-elements to $\top$, or (ii) it has been computed using the safe IN flowmap of its node. From Section 3, recall that if all the program-nodes of an SCC are safe, then no ghost-mappings can be present in the fixed-point solution. In this step, we implement the initialization-step (and its optimizations discussed in Proposal 3) which ensures that all the program-nodes are safe.

Recall from Optimization I/II of Proposal 3, that while calculating the IN flowmap of a program-node, we ignore the OUT flowmap of those predecessors that are not safe. Hence, we maintain a set safeNodes, of the nodes that have been marked as safe during the initialization-step. If any of the predecessors were ignored for a program-node during its processing, then the program-node is added to the set underApprox, to be processed in the stabilization-step (see Step IV). Both these sets are initially empty (see Line 15).

The initialization-step is performed in two parts, as per Optimization II/II of Proposal 3, by invoking the method underApproximate twice with different initial worklists (entry-nodes vs seed nodes). Note that the first invocation of underApproximate is not required if every seed node is also an entry node – in that case, the seed node will always have at least one safe predecessor. The method underApproximate employs a worklist-based algorithm to process the program-nodes in the given worklist. Note that since a seed node could also be an entry-node, it may get added twice to the worklist; to avoid processing any node more than once in the initialization-step, the set safeNodes is used to ignore the already-processed nodes. The method maintains a set named unchangedOUTNodes, which contains those nodes whose OUT flowmap do not change upon their processing during the initialization-step. If all predecessors of a node n belong to the unchangedOUTNodes set, then processing node n will not change its OUT flowmap; in this case, node n is added to the unchangedOUTNodes set (Line 10). Otherwise, if there exists even a single predecessor of n which is not present in the unchangedOUTNodes set, then the node n is processed (Lines 11-15), as discussed next.

Before processing any program-node n, the set of its safe predecessors is calculated first. A predecessor is added to safePreds (Line 12), if it belongs to a previous SCC (in the topological-sort order of the SCCs in the SDG), or it has already been processed in the initialization-step (as maintained in safeNodes). This subset of predecessors is used to calculate the new IN flowmap for n. If some predecessors of n are not in safePreds (and hence the flow information computed for n is incomplete), we add n to the set underApprox (Line 13), so as to complete its processing in the stabilization-step. Using the set safePreds, the new IN and OUT flowmaps of node n are calculated by invoking the method processNode.

The method processNode shown in Fig. 20 is similar to the one used in standard IDFA, with minor changes. We compute IN(n), by only considering the set of safePreds. Then, we invoke the appropriate analysis specific flow function ($\mathcal{F}_n$) to compute OUT(n). If the OUT flowmap of the node changes as a result of this invocation, then the method processNode returns true; otherwise it returns false. If the method returns false, then node n is added to the unchangedOUTNodes set (Fig. 19, Line 15).

Once the processing of node n is complete, it is marked as safe (Line 16). We add to intraSCCWL all the successors of n that are present in the current SCC, except the nodes that are already marked as safe, and the seed-nodes. We add to intraSCCWL only if (i) the OUT flowmap of node n changed during invocation of processNode, or (ii) the flag unconditionallyAdd was set during invocation of underApproximate.

As discussed in Optimization II/II, before the second invocation of underApproximate, we want to mark all the reachable nodes between entry-nodes and up until the seed-nodes as part of safeNodes. This is achieved by setting the flag unconditionallyAdd during the first invocation. As a result of this approach, it is ensured that at least one of the predecessors of each seed-node is safe, before the second invocation of underApproximate.

Step IV: (Stabilization-step) At the end of the initialization-step, all the program-nodes are safe. However, there may be program-nodes that contain incomplete flowmaps as one or more of their predecessors were ignored during the calculation of their IN flowmaps. Such nodes were saved in the set underApprox. Hence, in the stabilization-step, shown in the method stabilize, the worklist is initialized with the set underApprox (Line 20, Fig. 18), and the standard worklist-based algorithm is applied until the worklist is empty. Note that none of the predecessors in this step are skipped during the calculation of IN for a program-node. Further, none of the successor program-nodes from within the same SCC are skipped from addition back to the worklist if/when the OUT flowmap of the program-node changes. This is achieved by sending the argument pred(n) for safePreds, and argument $\emptyset$ for skipSuccs (Line 22), when invoking the method processNode.

Step V: Once the flowmaps of all the program-nodes of the current SCC have reached their (maximum) fixed-point values at the end of Step IV, we compare the states of OUT flowmaps of

```
1  Function processNodeUsingAccessedKeys($n$, $\mathcal{L}$, $\mathcal{F}$, IN, OUT, safePreds): boolean /* Modifies IN and OUT */
2      // Returns true if the OUT flowmap of $n$ has changed
3      // safePreds: set of predecessors whose OUT is a safe initial estimate
4      if safePreds == $\emptyset$ then IN($n$) = $\top$; else IN($n$) = $\bigsqcap_{p \in \text{safePreds}}$ OUT($p$);
5      accessedKeys = $\mathcal{AK}(n)$; // $\mathcal{AK}$ map is used to track the keys accessed in application of $\mathcal{F}_n$.
6      if IN($n$) == NULL || $\exists c \in$ accessedKeys, such that the mapping for c changed in the new IN($n$) then
7          OUT($n$) = $\mathcal{F}_n$(IN($n$));
8      else
9          OUT($n$) = IN($n$).clone();
10         foreach $c \in$ accessedKeys do
11             if oldOUT($n$) contains $c$ then OUT($n$)($c$) = oldOUT($n$)($c$);
12     if OUT($n$) has changed then return true;
13     else return false;
```

Fig. 21. Algorithm to recalculate IN and OUT of a program node, while using the accessed-keys heuristic.

exit-nodes with the values stored in the snapshot saved in Step II. If the values for any exit-node changes, then all its successors from successor SCCs are added to the set globalWL (Line 24).

B.3 Accessed-Key Heuristic

To use the accessed-key heuristic, in Fig. 19, instead of calling the function `processNode`, we call a modified version of it called `processNodeUsingAccessedKeys` (shown in Fig. 21). In this heuristic we invoke the transfer function, only (i) if the node is being processed for the first time, or (ii) if upon calculation of the new IN flowmap, there exists a domain-element whose mapping has changed from its value in the old IN flowmap (Line 7). Otherwise, the heuristic reconstructs the new OUT flowmap using the old OUT flowmap and the new IN flowmaps as discussed above.

For this heuristic to be applicable the program-nodes have to be immutable, even under program change, so that the set of accessed-keys do not change. In case of updates to the contents of a program node, the condition of immutability can be ensured by treating the change as two operations – deletion of the old program-node from the super-graph, and addition of a new program-node with the modified contents.

C Using IncIDFA for Dataflow Analysis of Parallel Programs

As discussed in Section 1, iterative dataflow analysis of explicitly parallel programs often requires adding extra edges to the flowgraph to model inter-thread communication that occurs through shared memory accesses. These edges are typically introduced between various synchronization primitives (such as post/wait operations, barrier directives, and so on) depending on the language semantics [12, 18, 30, 34, 46, 55, 84, 107, 108, 118].

C.1 Background

To demonstrate the applicability of the proposed algorithm for incremental iterative dataflow analysis of explicitly parallel programs, we use OpenMP [29], an industry-standard API for shared-memory parallel programming. We now provide a brief introduction to some key synchronization constructs of OpenMP (in Section C.1.1), followed by a discussion of phase analysis (in Section C.1.2), which is used to add synchronization edges (in Section C.1.3), and finally an overview of how standard iterative dataflow analysis can be performed on OpenMP programs using such synchronization edges (in Section C.1.4). It is important to note that the proposed algorithm, as discussed in Sections 3-4, is agnostic to how inter-thread communication is modeled in the flowgraphs.

C.1.1 OpenMP Constructs. Following are the key OpenMP constructs of our concern :

`#pragma omp parallel S`, denotes a parallel region that generates a team of threads, where each thread executes a copy of the code S in parallel.

`#pragma omp barrier`, specifies a program point where each thread of the team must reach before any thread is allowed to move forward. We use `ompBarrier` to abbreviate this directive. In the flowgraph representation of an OpenMP program, we represent each `ompBarrier` node with two separate nodes – a `preBarrier` node and a `postBarrier` node, with a control-flow edge from the former to the latter.

`#pragma omp flush`, is used to flush a thread's temporary view of the shared memory to make it consistent with the shared memory. We use `ompFlush` as a shorthand for this directive.

OpenMP adds implicit flushes at various synchronization points in the program, such as during lock acquisition and release, at barrier directives, at the entry and exit of critical constructs, and so on. Similarly, implicit barriers are added by default at the end of parallel loops, parallel regions, and so on. We assume that all such implicit flushes/barriers have been made explicit.

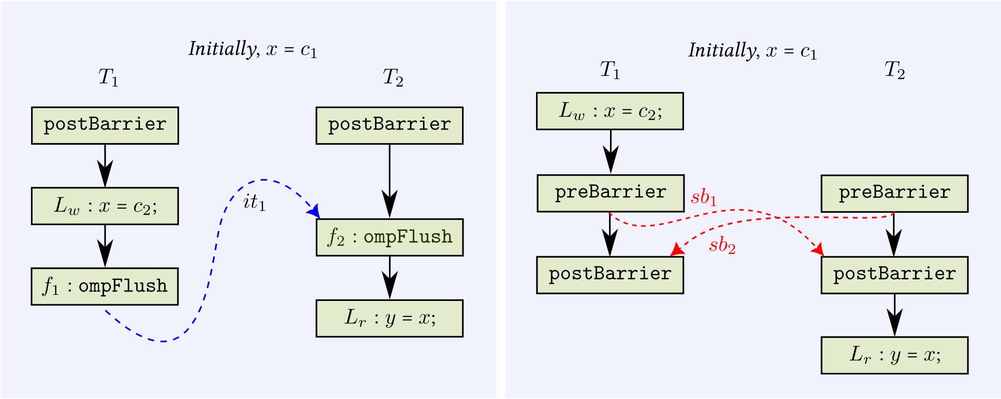
C.1.2 Phase Analysis. In a parallel region in OpenMP, all threads start their execution from the implicit barrier at the start of the region, until they encounter a barrier on their execution paths. Once each thread has encountered a barrier (same or different), all the threads will start executing the code after the barrier, until they encounter the next set of barriers. All such barriers that synchronize with each other form a *synchronization set*. Statically, a barrier may belong to more than one synchronization set. All statements that exist between two consecutive synchronization sets form a static *phase*. Each statement may belong to more than one phase. Two statements in two different phases that do not share any common phase may not run in parallel with each other. Precise identification of such phases can greatly improve the precision of various static analyses of parallel programs, as they limit the pair of statements that may communicate with each other through shared memory. For the purpose of our discussion, any phase analysis should suffice; we use the phase analysis derived from the concurrency analysis provided by [153, 154].

C.1.3 Synchronization Edges. In dataflow analysis of explicitly-parallel programs, special edges (for example, *synchronization* or *interference* edges) are added to the flow graphs to model inter-thread communication through shared-memory accesses [12, 18, 30, 34, 46, 55, 84, 107, 108, 118].

In order to model inter-thread communication in OpenMP programs, two kinds of synchronization edges are added in a flowgraph: (i) inter-task edges between `ompFlush` operations that may share a phase, and (ii) sibling-barrier edges, from `preBarrier` nodes to `postBarrier` nodes corresponding to barriers that share a synchronization set.

Inter-task edges. In OpenMP parallel regions, different threads (or tasks) may communicate with each other with the help of shared memory. As per OpenMP standards, a valid communication can take place between threads T_1 and T_2 through a shared variable, say v, only if the following order is strictly maintained: (1) T_1 writes to v, (2) T_1 flushes v, (3) T_2 flushes v, and (4) T_2 reads from v. Furthermore, T_2 must not have written to v since its last flush of the variable. We model such perceived communications with the help of `inter-task edges`, as shown in Fig. 22a. An inter-task edge is an edge that originates at an `ompflush`, say f_1 and terminates at (same or different) `ompFlush`, say f_2, such that (i) f_1 and f_2 share at least one common static phase, and (ii) there must exist some shared variable which is written on an `ompFlush`-free path before f_1, and read on an `ompFlush`-free path after f_2.

Sibling-barrier edges. As per OpenMP semantics, at the end of each barrier, all temporary views of different threads are made consistent with each other and with the shared memory. Consider a maximal set S of barriers that synchronize with each other at the end of a phase p; this set is termed as the *synchronization set* of p. For each pair of barrier $(b1, b2) \in S \times S$, in order to model the consistency of temporary views at the end of phase p, we add a special kind of synchronization-



(a) Inter-task edge

(b) Sibling-barrier edge

Fig. 22. Examples of two types of synchronization-edges in OpenMP, denoted by dashed lines. These edges model inter-thread communications. Solid lines denote one or more paths between two nodes in the flowgraph.

edge, termed as *sibling-barrier edge*, from the `preBarrier` of $b1$ to `postBarrier` of $b2$, as shown in Fig. 22b.

The inter-task edges and sibling-barrier edges can be used to port any standard iterative dataflow analysis to explicitly parallel programs, as discussed next.
We obtain a **super-graph** by adding these inter-task edges to CFGs and call graphs.

C.1.4 Parallel Iterative Dataflow Analysis. We now discuss an extension to the standard iterative dataflow analysis of serial programs, to make it suitable for parallel programs; we use OpenMP as the target parallel language. In OpenMP (and other such shared-memory parallelism models), there are two key portions of data environment for each thread – *private* and *shared*. Under the relaxed-consistency model of OpenMP, each thread is allowed to maintain its own temporary view of the shared memory, which can be made consistent with the global view of shared memory with the help of `ompFlush` directives. The communication between multiple threads resulting from these directives can be represented as inter-task edges, and sibling-barrier edges, as shown in Fig. 22.

We distinguish the private and shared portions of data-environment, such that the domain of each flow map consists of two mutually exclusive and exhaustive subsets. Consequently the standard flow maps can be seen as, $\mathtt{IN}(n) := \mathtt{IN}_{\text{priv}}(n) \cup \mathtt{IN}_{\text{shared}}(n)$, and $\mathtt{OUT}(n) := \mathtt{OUT}_{\text{priv}}(n) \cup \mathtt{OUT}_{\text{shared}}(n)$, where the subscript to a map indicates the type (memory locations) of its domain. Now, we discuss the key extension to serial IDFA to obtain IDFA$_p$.

Extension 1. In order to model the effects of an inter-task edge (f_1, f_2) between two `ompFlush` directives f_1 and f_2, for each shared variable x that can be communicated over the edge (f_1, f_2), we add the following dataflow equation to the serial IDFA; this equation uses the meet operator ($\sqcap$) of the underlying serial IDFA.

$$(\mathtt{IN}_{\text{shared}}(f_2))(x) := (\mathtt{IN}_{\text{shared}}(f_2))(x) \sqcap (\mathtt{OUT}_{\text{shared}}(f_1))(x) \qquad \textbf{(Ext-1)}$$

Correctness argument. We note that a sound IDFA for serial programs, extended with **Ext-1** remains sound in the context of OpenMP. OpenMP API specification [29, Section 1.4.4] specifies the sequence of events that *must* be followed for communication to happen between any two threads. Extension **Ext-1** conservatively models all such possible communications that may happen over any of the inter-task edges. Thus, since the underlying serial IDFA is sound, the extension remains sound, as no valid flow of shared data may happen at runtime except over the edges which we

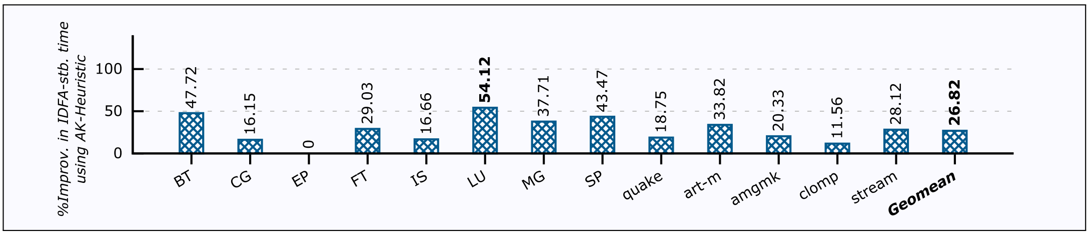


Fig. 23. Percentage improvement in IDFA stabilization-time using `IncIDFA` when applying the client optimization, `BarrElim`, with respect to `IncIDFA-NAC`, which is a version of `IncIDFA` with accessed-keys heuristic disabled. Higher is better.

model.

Complexity and Precision. For any given IDFA, the worst-case *complexity* of its corresponding IDFA$_p$ (upon adding the above proposed two kinds of sycnchronization-edges) remains unchanged. The *precision* of IDFA$_p$ depends on that of the underlying phase analysis, as imprecision in phase-analysis may result in modeling extra inter-task edges than needed.

C.2 Analysing Parallel Programs with `IncIDFA`

In this section, we discuss the applicability of `IncIDFA` for analyses of explicitly parallel programs. Recall that in order to model flow of data between threads through accesses to shared memory location, the flowgraph representation used for iterative dataflow analysis of parallel programs comprises of special edges, also termed as synchronization or interference edges [12, 18, 30, 34, 46, 55, 84, 107, 108, 118]. As discussed in Section C, we focus our discussion on two kinds of synchronization edges used by IMOP – (i) inter-task edges, and (ii) sibling-barrier edges. Together, these two kinds of edges represent most types of synchronization edges that have been discussed in the literature.

We notice that `IncIDFA` is applicable as is for the case of iterative dataflow analyses on such flowgraph representations. The special edges, as well as the special nodes (such as `ompFlush`, `preOmpBarrier`, and `postOmpBarrier`), are treated just like any other edge or node in the flowgraph. Note that the definition of meet operation remains to be the same in `IncIDFA`, as in the case of `CompIDFA` – only the dataflow information corresponding to shared memory locations (if any) are considered during the meet operation for the `OUT` flowmaps of the sources of incoming sibling-barrier edges at any `preOmpBarrier`.

D Miscellaneous Data

Performance impact of accessed-keys heuristic on K2. In Section 6.2, we have provided evaluation numbers for Nanda, to assess the impact of accessed-keys heuristic by comparing `IncIDFA` with `IncIDFA-NAC`. Now, we present the numbers for our evaluations on K2. Fig. 23 shows the percentage improvement using the heuristic, for K2. The baseline numbers, for `IncIDFA-NAC`, are present in Column 14 of Table 1. We see that `IncIDFA` consistently outperforms `IncIDFA-NAC`; improvements up to 54.12% (geomean 26.92%). Since the reasoning and observations with K2 remain same as that for Nanda, we skip the discussion for this graph.

In Fig. 24, we show the total number of transfer-function applications that were skipped as a result of the accessed-keys heuristic, normalized with respect to the number of applications in the baseline `IncIDFA-NAC`. We notice that there is a significant reduction in transfer-function applications with this heuristic, with maximum for art-m at 84.41%, and geomean of 59.91%. Consequently, this leads to improved compilation time, as discussed in Section 6.2.

Fig. 25 shows the normalized count of the number of applications of transfer-functions in `IncIDFA`-

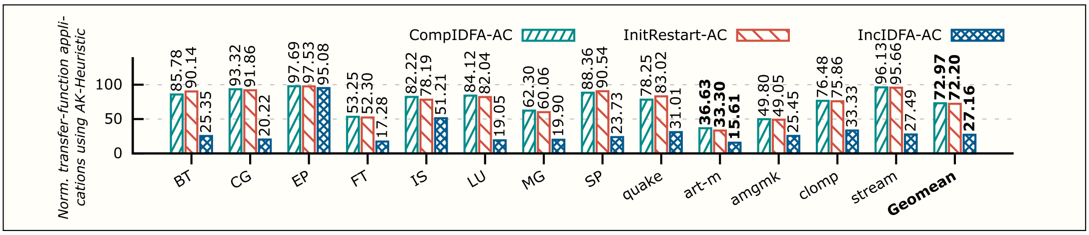


Fig. 24. Normalized count of the number of transfer-function applications when applying the accessed-keys heuristic over all three algorithms, as compared to per 100 applications in the respective algorithm with accessed-keys heuristic disabled. Lower is better.

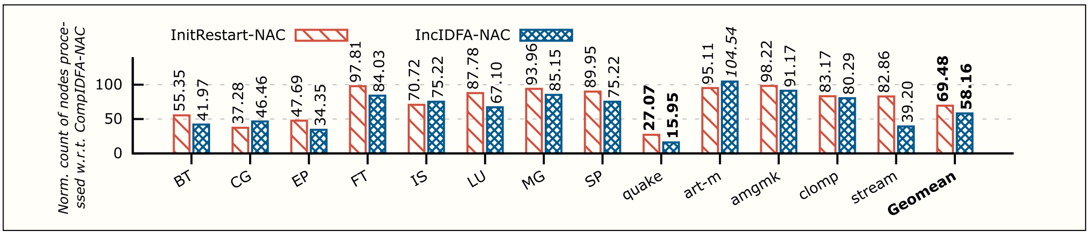


Fig. 25. Normalized count of the number of applications of transfer-functions in IncIDFA-NAC and InitRestart-NAC, per 100 applications of the transfer-functions in CompIDFA-NAC. Lower is better.

Optimization Pass in IMOP		Description
1.	Redundant barrier remover	removes barriers in absence of inter-thread dependences
2.	OmpPar expander	expands the scope of parallel-regions upwards and downwards
3.	OmpPar merger	merges two consecutive parallel-regions, if safe
4.	OmpPar-loop interchange	interchanges a parallel-region with its enclosing loop, if safe
5.	OmpPar unswitching	interchanges a parallel-region with its enclosing if-stmt, if safe
6.	Variable privatization	privatizes OpenMP shared variables, if safe
7.	Function inliner	selectively inlines monomorphic calls containing barriers
8.	Scope remover	removes redundant encapsulations of blocks in the given node
9.	Unused-elements remover	removes unused functions, types, and symbol declarations

Fig. 26. List of key passes in IMOP that belong to the `BarrElim` set of optimization passes. This figure has been reproduced (and summarized) from the paper by Nougrahiya and Nandivada [93].

`NAC` and `InitRestart-NAC`, per 100 applications of the transfer-functions in `CompIDFA-NAC`. Note that accessed-keys heuristic has been disabled for all three algorithms in this graph. Consequently, note that the number of applications of transfer-functions is same as the number of times nodes are processed during stabilization. This graph shows that `IncIDFA` results in a significant reduction in the total number of times nodes are processed, even in the absence of accessed-keys heuristic.

The `BarrElim` set of optimizations. In Fig. 26, we list key optimization passes that form the set `BarrElim`. The details of this pass are given in the paper by Nougrahiya and Nandivada [93]. Hence we skip the details here. As noted by the authors, `BarrElim` is a powerful set of optimization passes for OpenMP programs, achieving performance improvements of up to 5% on top of the already-optimized code generated by gcc with the -O3 switch. Therefore, in our evaluations we use the `BarrElim` set of optimization passes to generate the stabilization-triggers (using *Homeostasis*), which are then handled by our proposed incremental-update algorithm in order to stabilize the dataflow solutions in response to the program-changes (as provided by *Homeostasis*).

Received 2024-10-16; accepted 2025-02-18