Scaling Interprocedural Static Data-Flow Analysis to Large C/C++ Applications

An Experience Report

Fabian Schiebel
Fraunhofer Institute for Mechatronic Systems Design IEM, Paderborn, Germany

Florian Sattler
Saarland University, Saarland Informatics Campus, Saarbrücken, Germany

Philipp Dominik Schubert
Heinz Nixdorf Institute, Paderborn, Germany

Sven Apel
Saarland University, Saarland Informatics Campus, Saarbrücken, Germany

Eric Bodden
Paderborn University, Department of Computer Science, Heinz Nixdorf Institute, Germany
Fraunhofer IEM, Paderborn, Germany

Abstract

Interprocedural data-flow analysis is important for computing precise information on whole programs. In theory, the popular algorithmic framework *interprocedural distributive environments* (IDE) provides a tool to solve distributive interprocedural data-flow problems efficiently. Yet, unfortunately, available state-of-the-art implementations of the IDE framework start to run into scalability issues for programs with several thousands of lines of code, depending on the static analysis domain. Since the IDE framework is a basic building block for many static program analyses, this presents a serious limitation. In this paper, we report on our experience with making the IDE algorithm scale to C/C++ applications with up to 500 000 lines of code. We analyze the IDE algorithm and its state-of-the-art implementations to identify their weaknesses related to scalability at both a conceptual *and* implementation level. Based on this analysis, we propose several optimizations to overcome these weaknesses, aiming at a sweet spot between reducing running time and memory consumption. As a result, we provide an improved IDE solver that implements our optimizations within the PhASAR static analysis framework. Our evaluation on real-world C/C++ applications shows that applying the optimizations speeds up the analysis on average by up to 7×, while also reducing memory consumption by 7× on average as well. For the first time, these optimizations allow us to analyze programs with several hundreds of thousands of lines of LLVM-IR code in reasonable time and space.

2012 ACM Subject Classification Theory of computation → Program analysis

Keywords and phrases Interprocedural data-flow analysis, IDE, LLVM, C/C++

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2024.36

Supplementary Material *Software (Source Code + Experiment Results)*: `https://doi.org/10.5281/zenodo.13137082`

Funding This work was partially supported by the Fraunhofer Internal Programs under Grant No. PREPARE 840 231, and by the German Research Foundation under Grant No. AP 206/11-2, and within the Collaborative Research Center TRR 248 under Grant No. 389792660.

1 Introduction

Over the recent years static program analysis has become an important tool for finding bugs and security vulnerabilities [7, 11, 16, 26–28, 30]. To produce results that actually help developers in these tasks, static analyses are ideally both sound (or at least soundy [14]) and precise, i.e., they report only true findings without missing any real bugs and vulnerabilities. The analyses need to obtain a complete picture about the program under analysis and therefore have to be interprocedural, i.e., following procedure calls. But it is a major challenge to develop sound and precise inter-procedural analyses that scale well with large real-world target programs [6, 19, 31, 32].

The *interprocedural distributive environments* (IDE) framework [20] operates on data-flow problems whose flow functions distribute over the analysis' merge operator. Following the functional approach to interprocedural analysis [24], for such distributive data-flow problems IDE constructs fine-grained, per-fact, procedure summaries that can be reapplied in each subsequent calling context of a given procedure. This allows IDE to scale to larger programs relatively well even though its time complexity is $\mathcal{O}(|N| \cdot |D|^3)$, where N is the set of nodes of the target program's interprocedural control-flow graph and D is the symbol domain of the data-flow analysis.

Common static analysis frameworks such as Heros [5] and PhASAR [22] provide generic and parameterizable IDE solver implementations; they even implement the simpler IFDS [17] algorithm in terms of IDE. For an analysis problem on the desired target program to be solved in an automated manner, users of these frameworks merely have to specify its flow (and edge) functions and provide this specification to the IDE implementation. Current IDE implementations, also known as solvers, aim at analyzing real-world target programs in a fully flow and context-sensitive manner, computing precise and informative results depending on the quality of the flow (and edge) functions' specification. Nonetheless, the authors of this paper can tell from many years of experience in program analysis that all publicly available IDE implementations run into severe scalability issues for larger target programs – a major problem. This effectively impedes or even prevents the analysis of many real-world programs, or forces analysis developers to resort to simpler analysis domains, which reduces the precision and usefulness of the analysis results. Sattler et al., for instance, present a novel concept to combine program analysis and repository mining that addresses numerous relevant software engineering problems [21]. This approach, however, requires one to run an exhaustive IDE-based taint analysis that needs to generate and propagate *all* program variables, which, in turn, produces millions of data flows. In this vein, we use PhASAR's current IDE implementation to demonstrate that sound and precise analyses that produce more than 100 million data flow edges cannot be completed using ordinary consumer hardware. Such a huge number of data flows can easily arise already when analyzing programs that comprise fewer than 100 000 instructions in LLVM's [13] intermediate representation (IR). The number of IR instructions is relevant, since PhASAR performs its analyses on the LLVM-IR level, and even seemingly small C/C++ programs can lead to a large number of IR instructions. Still, using an IR enables analysis writers to develop analyses for programs originating from complex languages, such as C++, that would otherwise add drastic implementation overhead. Further, we can support analyzing programs from multiple different source languages (in our case C and C++) with just one analysis implementation, whereas a source-level analysis would need different implementations per language. Therefore, we prefer analyzing LLVM IR and handle the program size from within the solver.

In this work, we report on our experiences analyzing real-world programs with the IDE framework, identifying two critical optimization levers when implementing a generic state-of-the art IDE solver. Specifically, using 31 real-world C and C++ target programs, we evaluate PhASAR's state-of-the-art IDE solver implementation with regard to runtime and memory consumption. Based on insights gained from these experiments, we propose and evaluate two optimizations that we have devised to improve the performance of the IDE implementation. One optimization chooses an optimized data layout for storing required data, while the other one extends the garbage collection procedure from Arzt [1].

The improved IDE solver, which incorporates the abovementioned optimizations and insights, reduces analysis running times as well as memory consumption by up to 7× on average, depending on the client-analysis problem that should be solved. The experiments show that this allows one to conduct sound and precise inter-procedural data-flow analyses on interesting target programs such as FastDownward, a domain-independent planning system, in reasonable time and space.

In summary, we make the following contributions:

- We analyze the IDE algorithm as described in the literature and its state-of-the-art, openly-available implementations with regard to runtime and memory consumption.
- Based on the analysis, we propose optimizations that overcome these weaknesses.
- We report on an empirical study on our optimized IDE solver, showing that it improves runtime and memory usage of IDE-based analysis by up to 7× on average.
- We provide an open-source implementation of the IDE algorithm that incorporates our optimizations within PhASAR [22] and make it available as supplementary material[1].

The remainder of this paper is structured as follows: Section 2 gives an introduction to the IDE algorithm and Section 3 analyzes the state-of-the-art in IDE-based analysis and describes the problems that we identify. Section 4 presents our optimizations to IDE to mitigate these problems and Section 5 describes the highlights of our implementation. In Section 6, we detail on our empirical evaluation on real-world C/C++ programs and Section 8 concludes this paper.

2 Background on IDE

In this section, we introduce the conceptual *Interprocedural Distributive Environments* (IDE) [20] algorithm. IDE solves a data-flow problem by constructing an *exploded supergraph* (ESG). By construction, a data-flow fact d holds at instruction n, if a node (n, d) in the ESG is reachable from a special, tautological node (n_0, Λ) for an entry point statement n_0. The ESG is constructed by replacing each node in the target program's *interprocedural control-flow graph* (ICFG) with a bipartite graph representation of the respective flow functions. IDE requires all flow-functions to distribute over the merge operator (usually set union). Such distributive flow functions can be represented as bipartite graphs without loss of precision. The common flow functions *identity*, *gen* (generate), and *kill* (remove) are distributive and thus, all *gen/kill* data-flow problems can be encoded in IDE.

To enable a context-sensitive, interprocedural analysis, IDE follows the summary-based approach [24] to inter-procedural static data-flow analysis: It constructs per-fact summaries for sequences of instructions by composing their flow functions. The composition $h = g \circ f$ of two flow functions f and g, called *jump function*, can be produced by merging the nodes

[1] Supplementary Material: `https://zenodo.org/doi/10.5281/zenodo.13137081`

of g with the corresponding nodes of the domain of f. A jump function ranging from a given procedure p's starting point to its exit point builds up a summary ψ of p. Once summary ψ has been constructed for procedure p, it can be re-applied in any other context in which the procedure p is called. The runtime complexity of IDE is $\mathcal{O}(|N| \cdot |D|^3)$, where N is the set of nodes of the target program's ICFG and D is the data-flow domain of the analysis.

In addition, IDE allows to annotate the ESG's edges with lambda functions – so-called *edge functions* $f \in J$ – which operate on a separate value domain V and encode an additional value-computation problem. The value-computation problem specified using the ESG edges is solved when performing a reachability check. This way, IDE is able to effectively encode problems with infinite domains such as linear-constant propagation with $D = \mathcal{V}$, where $\mathcal{V}$ is the set of program variables and $V = \mathbb{Z}_{\perp}^{\top}$. In this setup, IDE would propagate constant variables through the program and compute their constant values using the edge functions. An exemplary ESG for a linear-constant propagation encoded in the aforementioned manner is shown in Figure 1. The ESG nodes are visualized in a matrix structure where the rows represent the program statements $n_1, \ldots, n_4$ and the columns represent the data-flow facts a, b, p and the special Λ fact. This way, Figure 1 also shows the bipartite nature of the encoded flow functions.

The jump functions constructed by the IDE algorithm describe data flows (and corresponding value computations). They comprise quadruples $\langle d_1, n, d_2, f \rangle$, where $d_1 \in D$ is the data-flow fact that holds at the source instruction (or node in the ICFG) $s_p \in N$, $n \in N$ is the target instruction, $d_2 \in D$ is the data-flow fact at the target instruction, and $f \in J$ is a function that describes the respective value computation. The source instruction s_p is implicit – it is the first instruction of the procedure that is being analyzed. In Figure 1, the jump function that describes that the data-flow fact a holds at ICFG node (n_4) in the program shown thus is: $\langle \Lambda, n_4, a, \lambda\ell.\ell \circ \lambda\ell.\ell + 2 \circ \lambda\ell.1 \rangle \equiv \langle \Lambda, n_4, a, \lambda\ell.3 \rangle$. Its evaluation yields that variable a carries the constant value 3 at ICFG node (n_4).

If an ESG node (n, d) is reachable along multiple program paths, the edge functions associated with the respective jump functions are combined using a *join* operation. Similar to flow functions, edge functions must distribute over the *join* operation. Hence, edge functions must be evaluable functions supporting regular function composition as well as the binary *join* operation and an *equality* relation. These operations – and the implementations for the flow and edge functions – need to be specified by analysis writers for the specific data-flow problem at hand.

The number of edges in an ESG is in $\mathcal{O}(|N| \cdot |D|^2)$. Even though D must be finite, D can be very large. Constructing the full ESG can easily lead to a graph containing millions of nodes and edges even for moderately-sized programs. Nearly all open-source state-of-the-art IDE implementations therefore construct only the valid paths reachable from the entry point (s_{main}, Λ) in an on-the-fly manner, as proposed by Naeem et al. [15].

Naeem's on-the-fly algorithm requires the following essential structures to solve an analysis problem:

- *JumpFn* ($D \times N \times D \to J$): Jump functions $\langle d_1, n, d_2, f \rangle$ tabulated by the IDE algorithm that describe the data-flow facts reachable from (s_{main}, Λ).
- *Incoming* ($N \times D \to N \times D$): A set that records nodes $\langle s_p, d \rangle$ that the analysis has observed to be reachable and predecessors of $\langle s_p, d \rangle$, where $s_p \in N$ a start point of procedure p. Using this set avoids the need to compute inverse flow functions, which might not be possible for all analysis problems.
- *EndSummary* ($N \times D \to N \times D \times J$): A table that stores jump functions that summarize the effect of a complete procedure p: $\langle s_p, d_1, e_p, d_2, f \rangle$, where $e_p \in N$ an exit point of p.

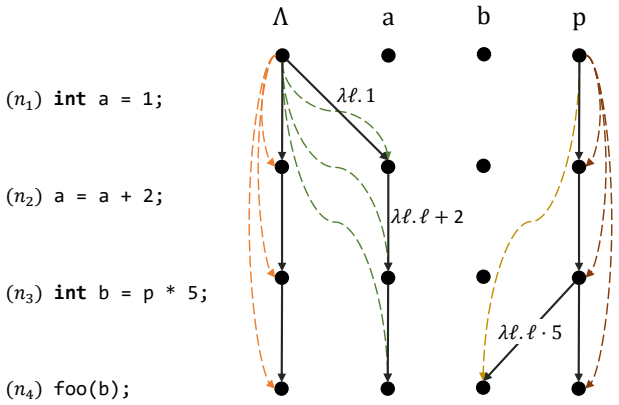

Figure 1 An example exploded supergraph for a linear constant analysis encoded in IDE [17]. The solid edges represent the individual flow functions, whereas the jump functions are denoted by the colored dashed edges. All (solid) flow edges are annotated with their edge functions; identity edge functions have been omitted to avoid cluttering. By following the flow edges in backwards direction, we can see that at (n_4) variable a is reachable from Λ and thus holds as data-flow fact. This information is also encoded as green dashed jump function from (n_1, Λ) to (n_4, a). Composing the annotated edge functions, we can see that at (n_4), variable a has the constant value 3.

These per-fact procedure summaries are reapplied in each subsequent context p is called.

2.1 IDE Algorithm Overview

The IDE algorithm works in two phases: (I) Constructing the relevant part of the ESG and (II) computing the values associated to the node-data-flow-fact pairs (n, d) by evaluating all edge functions f annotated to the jump functions in the ESG. We provide a copy of the original IDE algorithm as part of our supplementary website for this paper[2].

Phase I works as fixed point iteration starting from initial ESG nodes, called *seeds*. Based on the ICFG and the set of flow- and edge functions, the procedure ForwardComputeJump-FunctionsSLRPs (see algorithm Phase I) incrementally extends the ESG by adding new edges or updating the annotated edge functions of existing edges. This extending and updating of the ESG is performed by the Propagate (see algorithm Propagate) procedure, which gets iteratively called by the solver until a fixed point is reached. The final ESG for the example code snippet in Figure 1 is shown in the same figure (excluding the content of function `foo`).

Phase II (see algorithm Phase II) works in two steps: *value propagation* and *value computation*. First, in the value propagation phase, the initial edge values are propagated iteratively through the ESG from the seeds to the beginning of all analyzed procedures. After that, in the value computation phase, the edge functions of all remaining jump functions are evaluated with the values previously aggregated at the beginning of the respective procedure.

For example, consider the code snippet in Figure 1. Assuming that it is part of a function that gets called with $p = 4$, the value propagation will create the relation $(n_1, p) \mapsto 4$. If the code snippet is called with multiple different values for p, the relation gets updated using the lattice join of the value domain. Further, to aggregate the starting values for all procedures, the value propagation computes the relevant edge values for the call-site, in this case for b at n_4. It computes $b = (\lambda\ell.\ell \cdot 5)(4) = 20$ and iteratively propagates it into `foo`. After the value-propagation phase has finished, all remaining result relations can be computed, which leads to $(n_2, a) \mapsto 1, (n_2, p) \mapsto 4, (n_3, a) \mapsto 3$, etc.

[2] Supplementary website: `https://secure-software-engineering.github.io/paper-idesolverxx/`

3 The State of the Art

In many years of developing static data-flow analyses, we have found that state-of-the art analysis implementations, many of them implementing IDE (or a subset of it), do not scale to large programs comprising several hundreds of thousands to millions of lines of code. In the following, we report on the problems with current IDE implementations, with the example of PhASAR, that has lead us to define the optimizations to IDE that we present in Section 4.

To show the performance of a current state-of-the-art IDE implementation, we use the current `IDESolver` from PhASAR[3] in version v2403, which is the most recent stable version of the open-source framework at the time. To assess the state-of-the-art, we have applied the `IDESolver` to 31 real-world C and C++ programs[4] denoted in Table 1 and solved a typestate analysis (TSA), a linear constant analysis (LCA), and an instruction-interaction analysis (IIA) [21]. In Table 1 the columns with the analysis problems are sorted in ascending order by analysis complexity.

Measuring runtime and memory usage of the analysis runs, as Table 1 shows, we observed that, with increased analysis complexity, the number of recorded timeout (t/o) and out-of-memory (OOM) events grows. While the `IDESolver` was able to complete the LCA and TSA on almost all target programs, the solver performed worse on the IIA: In fact, we observed that six out of 31 could not be run on an ordinary developer machine, seven others ran out-of-memory while four others timed out.

The current situation, as illustrated by Table 1, that many interesting data-flow analyses cannot be solved on medium-sized to large target programs is inacceptable. While long runtimes can be tackled by running the analysis less often (e.g., in a CI/CD pipeline) or by increasing the time budget, the high memory requirements are often impossible to solve due to hardware limits; more memory might be integrated which then—depending on the system—would incur high procurement- and operating costs.

As some state-of-the-art IDE implementations, such as PhASAR and Heros, are open-source, we are able to analyze them to gain insights where the performance bottlenecks are and propose optimizations (cf. Section 4) for lowering the time- and memory requirements of IDE.

4 Optimizations

To mitigate the scalability issues of IDE identified in Section 3, we reviewed state-of-the art literature regarding IDE implementations, profiled the IDE solver implementation within the PhASAR framework, and identified two aspects that suggest to offer potential for effective optimizations in terms of both runtime and memory consumption. Although the IDE algorithm works in two phases (see Subsection 2.1), we can tell from our experience that IDE spends the majority of its time during phase I—the part that IFDS and IDE have in common. Thus, we aim to optimize phase I.

First, while computing the target analysis' fixed point, an IDE implementation must efficiently store the set of jump functions. This corresponds to the *JumpFn* map [20] in the original algorithm. The jump-functions table stores all ESG edges that are computed by the IDE solver. That is, it stores quadruples drawn from $(D \times N \times D) \to J$. The size of the jump-functions table is therefore bound by $\mathcal{O}(|N| \cdot |D|^2)$. As it is unlikely to

[3] PhASAR: `https://github.com/secure-software-engineering/phasar/tree/v2403`
[4] Subsection 6.2 provides details on how the results were obtained and how the analyses were configured.

Table 1 On the left, we see all evaluation targets with additional information, such as the revision we analyzed and the amount of LLVM-IR code. The IR code size is important because PhASAR's IDE solver works at the IR level. In addition, we report the number of procedures (Proc), the number of globals (Glob), and the number of call-sites (Calls) in the IR, which may influence the performance of the analysis. The three rightmost columns show time [s] and memory consumption [MiB] of the benchmarked analyses utilizing the `IDESolver` from PhASAR. Orange cells indicate that the memory of a common consumer machine (32 GiB) was exceeded. Dark orange cells indicate that even a compute cluster with 128 GiB would be insufficient. Red cells indicate the analysis ran out-of-memory with a memory limit of 250 GiB, and blue cells represent timeout (t/o) events exceeding four hours of analysis time.

							Typestate		LCA		IIA	
	Revision	Domain	LOC	Proc	Global	Calls	Time	Mem	Time	Mem	Time	Mem
FastDownward	641d70b3	Planning	849k	35k	5k	176k	20	1 407	81	7 709	-	OOM
asterisk	a0946200	Signal processing	626k	8k	15k	85k	72	4 131	t/o	-	-	OOM
bison	849ba01b	Parser	123k	1k	1k	13k	38	1 974	82	8 885	-	OOM
bitlbee	fb774da0	Chat client	91k	1k	2k	12k	1	203	17	2 126	-	OOM
brotli	9801a2c5	Compression	103k	978	173	10k	2	315	9	1 640	505	43 220
bzip2	1ea1ac18	Compression	29k	154	182	1k	3	166	20	1 829	842	34 006
cat	1913bfcf	UNIX utils	6k	223	139	736	<1	45	1	243	40	1 986
cp	1913bfcf	UNIX utils	23k	524	373	3k	<1	86	4	577	288	12 398
dd	1913bfcf	UNIX utils	19k	319	287	2k	<1	69	11	1 214	497	16 014
file	e94d5264	UNIX utils	1k	66	170	314	<1	39	<1	53	3	413
fold	1913bfcf	UNIX utils	6k	210	130	715	<1	52	2	245	41	1 943
grep	cb15dfa4	UNIX utils	79k	808	424	6k	1	207	25	3 208	545	44 827
gzip	23a870d1	Compression	17k	251	351	1k	<1	67	7	1 049	91	9 364
htop	bc22bee6	UNIX utils	58k	917	1k	7k	19	290	12	1 647	1 680	102 431
hypre	f69f8ef4	Solver	713k	3k	3k	71k	86	6 461	1 259	77 313	t/o	-
join	1913bfcf	UNIX utils	10k	267	184	1k	<1	66	2	324	55	3 098
kill	1913bfcf	UNIX utils	5k	196	135	663	<1	43	1	215	39	1 689
lepton	429fe880	Compression	139k	3k	889	24k	3	331	35	4 062	2 902	87 637
libjpeg_turbo	2cad2169	File format	142k	582	184	7k	1	242	161	9 172	-	OOM
libsigrok	68321f73	Signal processing	148k	1k	4k	16k	2	338	8	1 257	t/o	-
libzmq	ec6f3b1d	C++ Library	162k	5k	1k	26k	29	2 120	9	901	t/o	-
ls	1913bfcf	UNIX utils	31k	646	515	3k	<1	111	14	1 642	301	21 901
lz4	4a555363	Compression	35k	445	424	4k	12	221	5	847	749	23 597
openvpn	cec4353b	Security	187k	3k	4k	24k	10	540	74	8 135	t/o	-
opus	bce1f392	Codec	131k	851	472	10k	1	233	38	5 160	3 851	143 264
poppler	315ab300	Rendering	546k	15k	15k	87k	207	3 573	125	11 788	-	OOM
uniq	1913bfcf	UNIX utils	7k	242	181	939	<1	54	2	260	44	2 316
wc	1913bfcf	UNIX utils	10k	272	187	1k	<1	61	2	338	52	3 056
whoami	1913bfcf	UNIX utils	5k	180	113	539	<1	42	1	209	36	1 489
x264	e067ab0b	Codec	500k	2k	2k	33k	48	3 151	203	19 605	-	OOM
xz	e7da44d5	Compression	10k	252	455	1k	<1	57	2	327	31	4 740

reduce this worst case bound, we propose in Subsection 4.1 to lower the constant factors of these bounds by optimizing the memory layout of the jump-functions table, which enables practical performance gains. Second, most jump functions computed by IDE are just needed temporarily to craft the procedure summaries ψ. Once a summary has been created, the corresponding intermediate jump functions are no longer needed. Hence, to reduce IDE's memory footprint, we propose in Subsection 4.2 to remove such intermediate entries from the jump-functions table. In fact, we extend the work from Arzt [1] by designing a garbage collector for jump functions that—in contrast to the one proposed by Arzt—is applicable to arbitrary IDE problems.

It is important to note that our optimizations do not target just one particular implementation; our optimizations are generally applicable.

4.1 Data Structures for the Exploded Supergraph

While solving an IDE data-flow analysis problem, the solver incrementally creates jump functions (see Section 2) that need to be stored in memory. To solve the analysis problem efficiently, the jump functions need to be stored efficiently, allowing for short lookup and insertion times as well as for a small memory footprint.

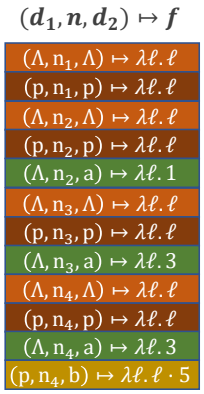

Figure 2a. The jump-functions table similar to the `FastSolver` of FlowDroid. Without nesting, the whole jump functions $\langle d_1, n, d_2, f\rangle$ of the ESG for Figure 1 are stored in one level which may lead some of d_1, d_2, and n being stored redundantly.

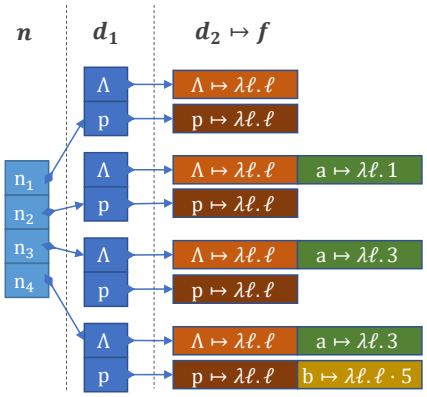

Figure 2b. The main jump-functions table from PhASAR and Heros. For each jump function $\langle d_1, n, d_2, f\rangle$, it maps the nodes n to inner maps, which map the source data-flow facts d_1 to the respective target facts d_2 and edge functions f. This avoids some nodes n and source facts d_1 to be stored multiple times, as they would be in Figure 2a, but adds extra cost for the inner mappings.

Figure 2 Different jump-functions table layouts currently used by open-source IDE implementations.

4.1.1 Jump Functions Table Analysis

Existing IDE solver implementations such as Heros [5], PhASAR [22] and FlowDroid [4] use different representations to store jump functions, each of which comes with different performance properties. PhASAR and Heros use nested mappings $N \to (D \to (D \to J))$ that map a target node $n \in N$ to a map of source data-flow fact $d_1 \in D$ to a map of target fact $d_2 \in D$ to the associated edge function $f \in J$. Yet, to speed up algorithm-specific lookup and insert tasks, Heros and PhASAR store each jump function redundantly in two additional maps, effectively modeling a multi-index table. In what follows, when referring to the jump-functions table structure used by PhASAR and Heros, we focus on the nested mapping described above, but keep in mind that the multi-index may have a drastic impact on the overall memory consumption of the solving process.

FlowDroid uses a flat $(N \times D \times D) \to D$ representation to map a full jump function $(n, d_1, d_2) \in N \times D \times D$ to the same target fact d_2. As FlowDroid only implements IFDS, which is a subset of IDE where all edge functions are implicity the identity function $\lambda x.x$, it does not need to store edge functions $f \in J$. It stores the target fact twice for implementation-specific support for path-tracking. As path tracking is out of scope for this work, we concentrate on the $(N \times D \times D)$ part of the data structure.

Both data structures (nested and flat) have their advantages and drawbacks. Consider the example in Figure 1. Having no nested mappings, as shown in Figure 2a, makes lookup and insertion fast, since they only consist of a single hash-map operation. In contrast, the nested approach, as shown in Figure 2b, requires three hash-map operations for each lookup or insert as for each of n, d_1 and d_2 in a jump-function entry a separate hash-map lookup or insertion is required.

In both designs, the noticeable duplication of the edge functions f could be solved by storing them in a separate cache. PhASAR, in fact, supports such a cache already. However, even with caching edge functions, nodes n and source facts d_1 may be stored redundantly in memory. This is, because it is likely that there are multiple jump functions

that lead to the same target node, which corresponds to the existence of the jump functions $(d_{1,1}, n, d_{2,1}), \ldots, (d_{1,k}, n, d_{2,m})$ for $n \in N$, $\{d_{1,1}, \ldots, d_{1,k}, d_{2,1}, \ldots d_{2,m}\} \subseteq D$ and $k, m \in \mathbb{N}$. Such jump functions may store the target node n multiple times in a flat structure, such as Figure 2a, but store n only once in a nested representation such as Figure 2b.

In the same vein, when generating data-flow facts, it is also likely that there are multiple target facts for the same source-fact and target node, for example, jump functions of the form $(d_1, n, d_{2,1}), \ldots, (d_1, n, d_{2,m})$ for $n \in N, \{d_1, d_{2,1}, \ldots, d_{2,m}\} \subseteq D$ and $m \in \mathbb{N}$. For instance, the jump functions $(\Lambda, n_2, \Lambda, \lambda\ell.\ell)$ and $(\Lambda, n_2, a, \lambda\ell.1)$ from Figure 1 fall in that category. In a flat representation such as of Figure 2a, jump functions store both source fact d_1 and target node n redundantly, but avoid the redundant storage in a nested representation as shown in Figure 2b.

In summary, nested mappings store less data from the jump functions redundantly and therefore are likely to expose a lower memory usage than a shallow representation. Conversely, common operations such as lookup and insertion of jump functions in the table are likely to be faster in the flat representation as there are fewer indirections and fewer hashing operations. Furthermore, map data structures themselves have implementation-specific memory overhead. Therefore, a nested representation is more memory efficient than a flat one only if the additionally introduced maps grow beyond an implementation-specific threshold to compensate the overhead of these maps.

4.1.2 Optimized Jump Functions Table

Given the analysis in Subsubsection 4.1.1, we propose a compromise between nested and flat data structure representations that harnesses the advantages of both to drastically improve both the memory usage as well as the runtime of the IDE algorithm. We acknowledge that a nested mapping is necessary to avoid duplicate storage of nodes and data-flow facts. However, to keep lookup times low and to keep the individual maps sufficiently large, we aim at reducing the nesting depth as well. Specifically, we propose a two-level nested map as a compromise between fast lookup times and low memory usage. For a design with two levels of nesting, there are six possible mappings to store jump functions:

1. $(n, d_1) \mapsto (d_2 \mapsto f)$
2. $(n, d_2) \mapsto (d_1 \mapsto f)$
3. $(d_1, d_2) \mapsto (n \mapsto f)$
4. $n \mapsto (d_1, d_2) \mapsto f$
5. $d_1 \mapsto (n, d_2) \mapsto f$
6. $d_2 \mapsto (n, d_1) \mapsto f$

To reduce the number of candidate representations, we consider one more optimization: As we limit ourselves to two-level nested maps, each jump functions access requires at least two indirections. However, with intelligent batch-processing, the effective number of indirections can be reduced. We observe that during ESG construction in the IDE algorithm(cf. Subsection 2.1), the only direct access to the jump-functions table is inside the Propagate function depicted on the left side of Algorithm 1. Here, the expression $JumpFn(e)$ performs the jump-functions table access where e represents a complete jump edge consisting of the target node n and the source- and target data-flow facts d_1 and d_2. We further observe that in the original algorithm Propagate is always called from within a loop where parts of n, d_1, or d_2 are loop-invariant.

So, if we design the jump-functions table accordingly, we can optimize the Propagate procedure (shown on the right side of Algorithm 1), by batching the access to the outer map for multiple jump functions accesses together. Here, Propagate receives an additional parameter j that denotes a view into the jump-functions table where the loop-invariant parts

Algorithm 1 The modifications in the Propagate procedure that support batch processing. An exemplary use of Propagate for the case in which the target node n is loop-invariant is shown in Lines 8-11. To highlight changes compared to the original algorithm from Sagiv et al. [20], additions are shown in ***green*** and removals are shown in ***red***.

```
1  Procedure Propagate(e, f)
2  |  let f' = f ⊓ JumpFn(e);
3  |  if f' ≠ JumpFn(e) then
4  |  |  JumpFn(e) = f';
5  |  |  Insert e into PathWorkList;
6  |  end
7  end
   // Example use:
8
9  for ... do
10 |  Propagate(⟨s_p, d1⟩ → ⟨n, d2⟩, f);
11 end
```

```
Procedure Propagate(j, e, f)
|  let f' = f ⊓ j(e);
|  if f' ≠ j(e) then
|  |  j(e) = f';
|  |  Insert e into PathWorkList;
|  end
end
// Example use:
j = JumpFn(⟨∗, ∗⟩ → ⟨n, ∗⟩);
for ... do
|  Propagate(j, ⟨s_p, d1⟩ → ⟨n, d2⟩, f);
end
```

Table 2 Access patterns of the jump-functions table with their number of occurrences within the original IDE algorithm [20] (cf. Subsection 2.1).

Invariant parts	# Occurrences
n	1 (call-flow)
n, d_1	2 (call-to-return-flow, summary-flow)
n, d_2	1 (return-flow)
d_1	1 (normal-flow)

are already fixed. In the example, j only contains jump functions where the target node is a previously fixed n. It is important that the extraction of j happens outside of the loop that calls Propagate. Using the smaller map j for accessing the jump functions instead of the complete table $JumpFn$ may improve the performance of Propagate. In fact, if j is one of the inner maps of our two-level nested jump-functions representation, using j effectively reduces the nesting depth of the table within Propagate, which in turn reduces the runtime cost of accessing individual jump functions.

Efficiently extracting the view j from the jump-functions table requires that the jump-functions table is laid out in a way that supports this operation. This can be achieved by placing the loop-invariant parts as keys into the outer map and the loop-variant parts into the inner maps. To decide which view j is best suited to achieve maximum performance improvement, we have to analyze which parts, n, d_1, or d_2, of a jump function are most frequently loop-invariant.

Based on careful analysis of the original algorithm [20], we identify four different access patterns, as depicted in Table 2. Although n is not strictly invariant in the normal-flow case, it may still be beneficial to consider n as invariant for the purpose of selecting a jump-functions representation, as most intraprocedural control-flow nodes mostly have only one (statement-sequence) or two (conditional branch) successors. Furthermore, to propagate all normal flows, the algorithm needs to iterate over all relevant n, d_2 pairs which is usually implemented as nested loop, effectively making n or d_2 temporarily loop-invariant. This consideration has no influence on the algorithmic correctness, but on the effectiveness of batch-processing jump functions accesses in the table.

Based on these observations, we conclude that it is beneficial to store the target fact d_2 in the inner map and n in the outer map. This enables us to filter out most of the six possible mappings presented above, leaving only

1. $(n, d_1) \mapsto (d_2 \mapsto f)$
4. $n \mapsto (d_1, d_2) \mapsto f$

as possible candidates, which we call JF_{ND} and JF_{N}, respectively, denoting the domain used in the outer map.

Furthermore, we also conjecture that a multi-index representation of the jump-functions table is not necessary. With any of JF_{ND} or JF_{N} we can efficiently model all access patterns that occur in the IDE algorithm. Hence, we introduce a third jump-functions representation, JF_{old}, that uses the deep nesting from PhASAR and Heros ($n \mapsto d_1 \mapsto d_2 \mapsto f$), but avoids the multi-index.

Our theoretical analysis also yields that, with JF_{ND}, we already have efficient access to the procedure summaries, eliminating the need for an extra *EndSummary* table that was proposed by Naeem et al. [15]. To access a summary[5] of procedure p, we can directly lookup the necessary jump functions at p's exit statements. With JF_{N}, to find matching summaries without the *EndSummary* table, one requires a linear search over the inner maps at p's exit statements. Depending on the size of these inner maps, this linear search may still be fast, so we split JF_{N} into two candidates: JF_{N} and JF_{NE} where JF_{NE} uses the explicit *EndSummary* table while JF_{N} omits it.

4.1.3 Discussion

From the observations in Subsubsection 4.1.2, one could conclude that JF_{ND} is superior to JF_{N} because, in three out of the five `Propagate` calls, d_1 is loop-invariant. However, in JF_{ND} (depicted in Figure 3a) the outer map is larger than in JF_{N} (depicted in Figure 3b) as its key space is larger: $|N| \leq |N \times D|$. Therefore, JF_{ND} needs to store more inner maps than JF_{N} although, in the end, both store the exact same number of jump functions. Furthermore, the inner maps in JF_{ND} are smaller than the inner maps in JF_{N}, as there are more of them and depending on the concrete implementation-specific overhead of a single inner map, the memory cost of the inner maps might outweigh their potential benefit. Hence, from a sole theoretical analysis, we cannot conclude which jump-functions representation performs better in practice; we need to perform an empirical evaluation to draw a final conclusion (Section 6).

4.2 Garbage Collection of Jump Functions

As discussed in Subsection 4.1, the jump-functions table has a great influence on the overall memory consumption of the IDE algorithm. Arzt [1] has shown that it is possible to remove entries in the jump-functions table without preventing the algorithm from reaching a fixed point. They present a garbage collector (GC) that runs concurrently to the actual IDE implementation, improving both memory usage and runtime of the underlying analysis. The GC removes jump functions when they are no longer needed. This applies when the complete data flow represented by a jump function has already been composed to a summary.

One limitation of the approach of Arzt [1] is that it only applies to an IFDS analysis and therefore does not need to deal with edge functions. In IDE, the value computation problem on data-flow edges can only be performed if the corresponding jump functions are

[5] Processing summaries as described in line 15.2 by Naeem et al. [15].

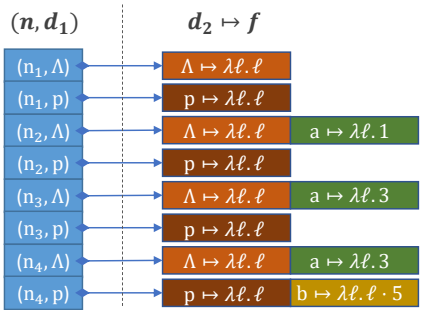

Figure 3a. jump-functions representation JF_{ND} for the example shown in Figure 1. The outer map has a two-dimensional key space consisting of the target node n and the source fact d_1, which reduces the size of the inner maps, containing only the target fact d_2 and the edge function f.

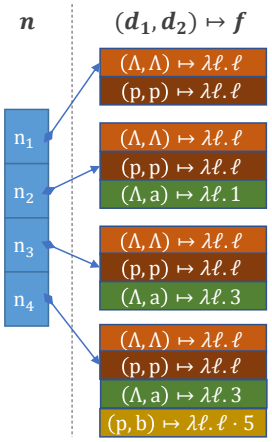

Figure 3b. jump-functions representation JF_N for the example shown in Figure 1. The outer map has a one-dimensional key space only consisting of the target node n, whereas the inner maps have a two dimensional key space containing the source- and target facts d_1 and d_2 as well as the associated edge functions f. Compared to JF_{ND}, JF_N contains fewer inner maps which in turn grow larger.

Figure 3 Exemplary jump-functions tables using the proposed representations JF_{ND} and JF_N.

present. This makes garbage collecting jump functions more complicated in a general IDE setting with associated edge functions. Although Arzt describes a possible extension of the GC to IDE as trivial, we recognize that the correct handling of corner cases makes it less obvious than it seems on the first glance. Especially, we need to ensure that subsequent result queries can still evaluate the edge-functions correctly that are annotated to the jump functions. Secondly, the garbage collection by Arzt [1] exploits multithreading at the level of the data-flow analysis solver. This requires the complete analysis toolchain to be thread safe. While some IDE implementations do satisfy this requirement and make use of multiple cores to speedup the solving process, other implementations are only single-threaded and do not provide thread-safe data structures. Specifically, PhASAR's analyses are not thread-safe and even LLVM—which PhASAR builds upon—is not generally thread-safe. Additionally, since we conduct a comprehensive study evaluating the runtime and memory consumption of IDE, we need to ensure that external factors, such as OS scheduling do not influence our evaluation results. Hence, we prefer using only a single thread, which eliminates many of these issues by removing non-determinism from the implementation.

In the following, we describe how we mitigate both limitations, the restriction to a subset of IDE and the enforced multi-threading.

4.2.1 Single-Threaded Garbage Collection

To keep the GC scalable, Arzt designed it to work on a procedure-level. That is, all jump functions corresponding to procedure p can be erased once there is no longer any worklist item that contains a node from inside p or from any procedure that can be transitively called by p [1]. We call this the *GC Condition*. Unfortunately, the order in which the ESG is constructed is not specified by the underlying algorithm [20], which is why one cannot precisely predict these points. If the garbage collector runs concurrently to the

actual analysis-solving thread, it can be invoked periodically based on a timer. Additional computations that the GC needs to perform to determine for which procedures the jump functions can be erased do not necessarily pause the analysis. However, as explained above, we decided to aim for a single-threaded solution here. The GC thus needs to be called explicitly at suitable points within the IDE algorithm and will pause the data-flow analysis for the garbage collection.

We observe that a procedure p can only become a candidate for garbage collection once the analysis within p has reached an exit statement. In theory, it is possible to invoke the GC after exiting any procedure, yet this has a non-negligible overhead that would render the analysis unscalable. Hence, we aim for finding a point in the IDE algorithm to place the GC, such that it gets called frequently enough to keep it effective, but not too frequent to keep it scalable. This means, that the GC should be invoked, once a sufficient amount of procedures have computed their summary.

There are several ways of deciding when the GC should be invoked, each with different characteristics and implications. One approach is to increment a counter, whenever a procedure has computed a new summary, and invoke the GC when the counter reaches a certain threshold. This approach has the advantage that it is easy to implement. On the downside, it does not decide to invoke the GC based on concrete information on the internal solver state, such as the content of the worklist or the jump-functions table. Therefore, many candidate procedures may actually fail the *GC Condition* and are not eligible for garbage collection yet. Hence, its performance may not be predictable and requires a decent amount of tuning. An alternative is to take the contents of the solver's worklist into account when deciding on when to invoke the GC. Since the *GC Condition* is based on the content of the worklist, we can invoke the GC when it is guaranteed that the candidate procedures will pass the *GC Condition*. In our implementation, we opted for this more informed procedure.

For deciding, when to invoke the GC, we split IDE's worklist into two separate worklists: One *PathWorkList* for top-down propagations, which stores jump functions in $D \times N \times D \times J$ to be processed, and another worklist, *RetWorklist*, for bottom-up summary applications that stores entries of the form $(d_1, p) \in D \times P$, where P is the domain of callable procedures in the target program. On a high level, the fixed-point iteration uses the *PathWorkList*, but also fills the *RetWorklist* on-the-fly when a procedure has reached its exit point. Once the *PathWorkList* becomes empty, the algorithm handles the work-items from the *RetWorklist*, which may fill the *PathWorkList* again. Although the data-flow propagations have stayed the same, using two worklists we now have structured the fixed-point iteration into stages (a stage ends, whenever the *PathWorkList* becomes empty) that allow placing a call to the GC.

For the two worklists to function properly, we modify the IDE algorithm as sketched in Algorithm 2. The pseudo code for handling procedure exit points that we removed in Line 9 of Algorithm 2 has moved to a new outer loop depicted in Algorithm 3. As applying procedure summaries may lead to new intra-procedural propagations at their return sites, the whole process runs in a loop until *both* worklists are empty, as shown in Algorithm 3.

Note that in subsequent iterations, the ForwardComputeJumpFunctionsSLRPs procedure must skip its initialization phase to not over-write the already computed results. Apart from that, we did not change the original IDE algorithm, as we describe in Paragraph 4.2.1.1.

Using two worklists, the garbage collection condition now slightly changes. The jump functions of a procedure p can only be collected if none of the *PathWorkList* and the *RetWorklist* contain a node from inside p or its transitive callees. This is, because when processing the worklist items (d_1, p) from the *RetWorklist*, the callers of p may be added to the *PathWorkList* again preventing garbage collection for p. Whenever the *PathWorkList* is

Algorithm 2 Modification in the ForwardComputeJumpFunctionsSLRPs procedure from the original IDE algorithm [20].

```
1  Procedure ForwardComputeJumpFunctionsSLRPs(...)
2      ...;
3      while PathWorkList ≠ ∅ do
4          Select and remove an item ⟨s_p, d_1⟩ → ⟨n, d_2⟩ from PathWorkList;
5          ...;
6          switch n do
7              ...;
8              case n is the exit node of p do
9                  Insert (d_1, p) into RetWorklist;
10             end
11             ...;
12         end
13     end
14 end
```

Algorithm 3 High-level overview of the two-step fixed point computation with garbage collection. The foreach loop in Line 5 denotes the content from ForwardComputeJumpFunctionsSLRPs [20] that we have removed from Algorithm 2. The function RunGarbageCollector behaves exactly as described by Arzt [1].

```
1  while PathWorkList ≠ ∅ do
2      ForwardComputeJumpFunctionsSLRPs(...);
3      while RetWorklist ≠ ∅ do
4          Remove (d_1, p) from RetWorklist;
5          foreach call node c that calls p with corresponding return-site r do
6              ...;
7          end
8      end
9      RunGarbageCollector();
10 end
```

empty, we have the guarantee that for all currently analyzed procedures (and their transitive callees), the analysis has reached their exit points, making them candidates for garbage collection. Hence, we now have a structure that precisely defines points for placing the GC.

In particular, we now have two candidate locations to place the garbage collection in Algorithm 3: Line 3: Right after the returning from ForwardComputeJumpFunctionsSLRPs (i.e., when the *PathWorkList* becomes empty) or Line 9: After the *RetWorklist* becomes empty. In Line 3, the *RetWorklist* is potentially non-empty as it may contain procedures p that have computed a new summary for the propagation of a source data-flow fact d_1 that needs to be propagated back to all callers of p. In Line 9, though, the *RetWorklist* is empty, whereas the *PathWorkList* may be filled with return flows again.

Both insertion points at Line 3 and Line 9 are very similar, however, Line 9 has one small benefit: Having a jump function from a procedure p in the *RetWorklist* prevents all caller procedures of p from being garbage collected. After processing the *RetWorklist* items, only those callers of p have jump functions in the *PathWorkList* for which the new information

from p requires further propagation. All other caller procedures can still be garbage collected (unless there are other callees that prevent the collection). This leads to our preference to place the garbage collection at Line 9. Note, although the worklists are processed until completion in one iteration of the outer loop, there are still potentially many iterations such that the garbage collector is run many times as well.

4.2.1.1 Correctness

Our modifications to the IDE algorithm and the integration of the garbage collection do not violate the correctness and complexity of the IDE algorithm. Splitting the worklist into two smaller worklists, as we have done in Algorithm 2 and Algorithm 3, does not create new worklist items that would not be created in the original, and also does not drop worklist items that would be processed in the original. Only the order, in which the worklist items are processed, may change. This is, because (1) the processing of exit nodes (cf. Line 9) gets delayed through the *RetWorklist* to Algorithm 3 without modifying the corresponding worklist items, and (2) since the processing order of the worklist items is not defined in the algorithm [20], any modification on the processing order has no influence on the correctness or complexity of the algorithm.

In addition, we use the same RunGarbageCollector function from Arzt without modification. Only the garbage collection condition, has slightly changed: Whereas in the original GC, a procedure p's jump functions can be erased, if the worklist does not contain a node from inside p or its transitive callees, in our extension, this requirement holds for both the *PathWorkList* and the *RetWorklist*. Since we argue above that both *PathWorkList* and *RetWorklist* in combination express the same worklist items as the original worklist, the correctness argumentation from Arzt still holds.

4.2.2 Generalizing Garbage Collection for IDE

When a procedure p gets evicted by the original GC from Arzt, all jump functions corresponding to that procedure are removed. However, when performing an analysis that uses IDE's edge functions, one needs to ensure that the value computation (cf. Subsection 2.1) can still be performed correctly. To solve the value computation problem for an ESG node $(n, d) \in N \times D$, the edge functions annotated to all jump functions that lead to node (n, d) have to be evaluated and thus need to be present. For example, removing the intermediate jump function $\langle \Lambda, n_3, a, \lambda\ell.3 \rangle$ in Figure 1 would prevent that the analysis computes the result relation $(n_3, a) \mapsto 3$. This makes garbage collection for IDE's jump functions impossible when the values for all ESG nodes must be computed. Fortunately, many analyses can define for which ICFG nodes $n_i \in N$ analysis-result queries may be raised before starting the solving process. For example, in a typestate analysis, only the API call nodes that are relevant for the analyzed usage pattern may be queried. We call those nodes n_i *interesting*. At *interesting* nodes, we erase no jump functions in the GC to ensure that at those nodes the complete analysis results including edge values will be present.

However, we have to retain additional jump functions: The value-propagation phase (cf. Subsection 2.1) first propagates initial edge values from the entry points to the starting nodes of all reachable procedures. This is done by iteratively querying and evaluating the jump functions at all call sites to map the initial values to the start of all reachable procedures. This initial value-propagation is necessary for the other jump functions to be evaluated, as it determines the input values for these jump functions. Therefore, for the value propagation to work properly, one must also retain the jump functions at all call sites, even if they are

not considered *interesting*, such that the value propagation to the starting points of all procedures can succeed. Hence, when using IDE's edge functions, the garbage collection must retain more jump functions than just the ones corresponding to *interesting* nodes, making it potentially less effective.

In the evaluation, we demonstrate that the garbage collection is still effective in a real world setting, even in a single-threaded environment and when using IDE without restrictions.

5 Implementation

We implemented the IDE algorithm including the optimizations proposed in Section 4 on top of the PhASAR framework [22]. PhASAR is able to analyze LLVM IR [13] in a fully automated manner and already provides an implementation of IDE, called `IDESolver` [22,23]. The `IDESolver` is parametrizable with an user-defined description of an IDE analysis problem that shall be solved. After solving the analysis problem, the `IDESolver` can answer queries about which data-flow facts hold at a given ICFG node and which edge value has been computed for a given node–data-flow fact pair $(n, d) \in N \times D$. We chose to provide the same interface in the new solver such that it can be used as a drop-in replacement. Note that the determination of *interesting* nodes for the garbage collector is completely opt-in, so only IDE analyses that use both the garbage collector and edge functions may need to implement it. We call our new solver `IDESolver++`.

The existing solver provides several configuration options that influence how the analysis problem should be solved (e.g., whether the value computation in IDE Phase II should be performed). Our new implementation is configurable as well, but we chose to lift the configuration from runtime to compile-time. This allows to specialize the solver for the selected configuration such that the algorithms and data structures can be selected precisely for the requested needs. For example, if the implementation detects at compile-time that the to-be-solved analysis problem does not need edge functions, the jump functions table will replace its inner map by a set, eliding the storage for associated edge functions that would otherwise all default to the identity function $\lambda x.x$.

In Section 4, we have shown different representations of the table storing the jump functions, and we concluded that this representation is critical for optimal performance of the overall solving process. Therefore, we chose to use open-addressing[6] hash maps to store the concrete mappings of the structures JF_{ND} and JF_{N}, as well as JF_{old}. Open-addressing hash maps are particularly performant because of their cache efficiency and small number of dynamic memory allocations. However, their performance degrades with increasing size of the entries to store. The domains N and D are user defined for both solvers (the current `IDESolver` and our `IDESolver++`) making them generic over the program representation to analyze and the type of data-flow facts. Therefore, we do not use these types directly as keys and values in the hash maps to guarantee predictable performance. Instead, we chose to introduce an intermediate layer that maps each used node and data-flow fact to 32-bit integers in the contiguous ranges $[0, \ldots, |N| - 1]$ and $[0, \ldots, |D| - 1]$. These integers are then used as keys/values in the actual jump-functions table. The sizes of the intermediate maps are negligible compared to the size of the jump-functions table. We reasonably assume that both N and D do not grow larger than $2^{32} - 1$, since these domains are bound by the size of the input program. For the JF_{N} (and JF_{NE}) approach, the intermediate layer enables one more optimization: The outer map can be replaced by a plain array to further reduce the memory footprint and to improve lookup performance.

[6] Open-addressing hash tables store all buckets in a contiguous block of memory, using probing for collison resolution.

Since the inner maps are very small in many cases, we chose to use `llvm::SmallDenseMap<K,V,4>` for the inner maps to optimize for the case in which these maps do not exceed a capacity of 4. This optimization is critical, especially for JF_{ND} and JF_{old}, because they store a large number of small inner maps, where their sizes mostly do not exceed the initial capacity (48 entries) of a regular `llvm::DenseMap`. Independent from the selected jump-functions representation, the corresponding outer hash map is pre-allocated with a reasonable size that scales linearly with the size of the input program. Together with the small-size optimization, this pre-allocation reduces the total number of potentially expensive (re-)allocations.

Our implementation is openly available in the supplementary material of this paper and we are already in contact with the maintainers of PhASAR for rapid integration into the open source framework.

6 Empirical Study

To empirically evaluate the optimizations proposed in Section 4, we use our IDE implementation (see Section 5) to analyze 31 real-world C/C++ programs. We start with defining our research questions.

6.1 Research Questions

Jump-Functions Table Structure

In Subsection 4.1, we have argued that the structure of the jump-functions table directly influences the performance of the analysis, especially regarding memory consumption. Hence, we ask:

RQ_1 | *What is the influence of choosing one of the proposed data structures,* JF_{ND}*,* JF_{N}*, and* JF_{NE}*, in terms of runtime and memory consumption when analyzing real-world C/C++ programs?*

Jump-Functions Garbage Collection

Arzt [1] has shown that a garbage collector for jump functions not only significantly reduces memory usage of the underlying analysis, but reduces runtime as well. As we have applied significant changes (cf. Subsection 4.2) to the garbage collection by extending it to general IDE problems and mitigating its restriction to multi-threaded analyses, we ask:

RQ_2 | *How effective is the jump functions garbage collector in reducing memory usage and running time when analyzing real-world C/C++ applications without the restrictions to a subset of IDE and a multi-threaded implementation?*

6.2 Experiment Setup

To ensure that our experiments are easily reproducible and comprehensible, we detail on our setup in the following. In Subsubsection 6.2.1, we define what kind of analyses we consider during the evaluation, and in Subsubsection 6.2.3 we present how we perform our measurements as well as the required actions to answer the research questions.

6.2.1 Analysis Problems

To test our solver implementation, we choose to evaluate it using three commonly used analysis problems that put a different amount of load to the solver:

- **TSA:** *Typestate analysis*, configured to find invalid usage patterns of `libc`'s file-IO API
- **LCA:** *Linear constant analysis*
- **IIA:** *Instruction-interaction analysis*, to generate git-blame reports [21].

These analysis problems are available within PhASAR, and we use them unchanged. The *typestate analysis* is expected to put low load on the solver as many programs use `libc`'s file-IO only in few small regions of their code. The *linear constant analysis* should put medium load on the solver, as it needs to propagate all potentially constant integer values; however, the implementation in PhASAR currently is not alias aware, so the load on the solver is still less than for the instruction-interaction analysis, which propagates all potential aliases of the generated data-flow facts. Finally, the *instruction-interaction analysis* puts an extreme load on the solver as it needs to exhaustively track *all* of the target program's variables and capture their interactions with the program's instructions [21]. This way, the size of the data-flow domain D approaches $|N|$ allowing us to approximate the worst-case scenario for field-insensitive analyses.

6.2.2 Target Programs

To ensure that our evaluation results reflect real-world analysis usage as closely as possible, we carefully select the set of 31 target programs shown in Table 1. We select the target programs out of 12 different domains to achieve broad coverage. Further, we choose the target programs in various sizes in the range from 1 676 to 849 623 lines of code in LLVM IR to test the IDE solver with different loads. The target programs have varying properties, such as the number of procedures (66 to 35 134), the number of address-taken functions (0 to 2 696), the number of globals (113 to 15 108), the number of call-sites (314 to 176 350), the number of indirect call-sites (0 to 2 155), and the number of basic-blocks (266 to 111 521).

We include the benchmarked programs from the initial PhASAR paper [22] excluding PhASAR itself, because it has grown significantly since 2019, such that expensive analyses, e.g., the IIA, do not work on that large programs anymore. Still, our evaluation results cannot be compared to the results from Schubert et al. [22], since we use different client analysis problems; the taint analysis used by Schubert et al. is of less interest for our work, since it does not require IDE to be solved efficiently. We also include programs from the evaluation of Sattler et al. [21] as they explicitly report performance problems of PhASAR's IDE solver on their benchmark. In contrast to the PhASAR benchmark, the time and memory results for the programs analyzed by Sattler et al. can be compared to our evaluation results, because the implementation and configuration of the IIA has not changed.

6.2.3 Measurement Setup

Each individual experiment is performed separately for each analysis problem. As analysis targets we use 31 real-world C/C++ programs, which we compile to LLVM 14 IR using WLLVM[7], so that PhASAR's analyses can consume them. To reduce measurement bias, we run each experiment (solver configuration × analysis problem × analysis target) three times and report average values. To validate that our experiments indeed show low variance, we compute the standard deviation of the runtime measurements of the three repetitions. We observe an average standard deviation of 2.2s to 8.3s depending on the jump-functions representation. Normalizing that by the total runtime, the average standard deviation lays

[7] WLLVM: `https://github.com/travitch/whole-program-llvm`

between 0.99% and 1.5% of the measured runtime. As we expect running times in the area of hours instead of seconds, the impact of measurement bias, as well as the variance between repetition is expected to be negligible and therefore, we consider the relative small number of repetitions $k = 3$ as sufficient to achieve reliable results.

We use the UNIX `time` utility to measure the total runtime and peak memory usage for all experiments. We compute speedups for runtime and memory consumption (maximum resident set size) by comparing the statistics of the to-be-evaluated configuration of the `IDESolver++` to the statistics of the respective baseline. Given runtime measurement samples $M_N = \{m_{n_1}, \ldots, m_{n_k}\}$ and baseline-measurements $M_B = \{m_{b_1}, \ldots, m_{b_k}\}$ with the number of samples $k = 3$, the speedup is defined as

$$S = \frac{1}{k^2} \sum_{(m_n, m_b) \in (M_N \times M_B)} \frac{m_b}{m_n}$$

For memory measurements, we use the inverse $\frac{1}{S}$ of the above formular to compute the relative memory usage in percent. We compare each combination of m_n and m_b, as these samples are unordered. This prevents potential biases due to sample ordering. Note that in contrast to Arzt [1] we can make use of the external tool `time` for measuring memory consumption, because our experiments do not run in the JVM that makes external memory measurements unreliable.

We conducted our evaluation on a compute cluster in an isolated and controlled environment to ensure that our measurements are not influenced by external factors. Each compute node is equipped with an AMD EPYC 72F3 8-Core processor and 250GiB of RAM, running a minimal Debian 10.

In addition, to increase the reproducibility of our results, we automate the evaluation process with the VaRA Tool-Suite[8].

Baseline. We also evaluate the existing state-of-the-art `IDESolver` that is openly available in PhASAR as shown in Section 3. As a baseline for our further experiments, we use the `IDESolver++` with the deeply nested jump-functions representation JF_{old}, which the `IDESolver` uses as well. In addition, we compare the both solvers in terms of runtime and memory consumption to assess the influence of our implementation in comparison with the current state-of-the art, when *not* applying the optimizations proposed in Section 4. Note that we do not implement the multi-index table for storing jump functions since the `IDESolver++` does not need it, as discussed in Subsubsection 4.1.2. To achieve a fair comparison, we need to configure the `IDESolver`. We set the configuration option `recordEdges` to `false` to avoid storing the ESG edges in a path sensitive way. We record runtime and memory usage, as well as out-of-memory (OOM) and timeout events of both solvers, providing a baseline to compare against in the evaluations of our research questions.

$\mathbf{RQ_1}$. We evaluate four configurations of our `IDESolver++`, one using JF_{ND}, JF_{N}, JF_{NE}, and JF_{old} as jump-functions table respectively. JF_{old} serves as a baseline for the others. To judge which jump-functions table structure performs best on our target programs, we compute the speedups compared to the baseline and consider the configuration with the highest speedup as best. To verify whether the best configuration is *significantly* best, we perform a t-test with significance level $\alpha = 0.05$. The garbage collector is turned off.

[8] VaRA Tool-Suite: `https://vara.readthedocs.io/en/vara-dev/`

RQ$_2$. We configure the `IDESolver++` as follows: turning the GC on or off and using JF$_{ND}$ or JF$_N$. The `IDESolver++` with GC turned off is used as baseline. We exclude JF$_{NE}$ here, because it stores the jump functions in exactly the same way as JF$_N$, just with one additional table that only contains jump functions which cannot be evicted by the GC at all. So, in total, we have four configurations for this experiment. For the typestate analysis all state transition instructions are considered *interesting*, whereas for the linear constant analysis, all branch conditions are considered *interesting*, which is useful when eliminating dead code, for example. All jump functions at those interesting instructions are ignored by the garbage collector. We exclude the instruction-interaction analysis for RQ_2 as its post-processing needs the results at all instructions [21] rendering the garbage collection useless. To examine the influence of the jump functions garbage collector on the analysis, we compute the speedups of the `IDESolver++` compared to its corresponding versions without GC. We consider the configuration with the highest speedup to perform best.

6.3 Results

We have conducted our experiments on the 31 real-world C/C++ programs listed in Table 1. Although we have already argued on the correctness of our optimizations, we ran an additional, non-measured analysis batch to confirm that the new `IDESolver++` indeed computes the same results as the `IDESolver`. In what follows, we detail on the results of our experiments and answer the before defined research questions.

6.3.1 Baseline

Our evaluation of the baseline shows that in almost all measured configurations the `IDESolver++` is faster and consumes less memory than the `IDESolver`. We measured runtime speedups ranging from 1.16× to 7.2× on average and memory savings from 0.96× to 4.8× compared to the `IDESolver` as shown in Table 3. Due to the variance, the benefits of using our `IDESolver++` may be program dependent. Note that sometimes the `IDESolver++` consumes more memory in the typestate analysis than the `IDESolver`. This is because the `IDESolver++` allocates large buffers in advance to lower the number of re-allocations (cf. Section 5); in addition, the typestate analysis is very sparse; it propagates only a very small number of data-flow facts and therefore does not fill out the pre-allocated buffers which we do not consider as a problem since the total memory usage is negligible.

In contrast to the `IDESolver`, the `IDESolver++` ran out-of-memory very rarely, as is apparent in Figure 4. However, the figure also shows that the number of timeouts is higher for the `IDESolver++` than for the `IDESolver`. That is because analyses that ran out-of-memory in the `IDESolver` were able to run long enough to exceed the given time budget in the `IDESolver++`. All of the experiments that completed with the `IDESolver` were also completed with the `IDESolver++`, showing that the performance does not degrade. In fact, out of the 7 experiments that exceeded the time limit of four hours, three were solved in time with the new solver; out of the five experiments that ran out of memory, one can now be completed within the memory limit of 250GiB. Furthermore, all 7 experiments that required up to 143GiB of RAM can now be solved on an consumer hardware with only 32GiB RAM.

There are several aspects that contribute to the improvements in this baseline experiment. The most notable ones are: The elision of the multi-index storage for jump functions (see Section 4.1.2), the batch-processing (see Algorithm 1) of data-flow fact propagations, and the switch from the `std::unordered_map` to `llvm::SmallDenseMap` (see Section 5).

Table 3 The average speedups/memory savings of the `IDESolver++` with JF_{old} compared to PhASAR's `IDESolver` together with their standard deviations

Analysis	Memory	Runtime
IIA	4.811 ±1.192	7.227 ±2.042
LCA	1.729 ±0.365	4.683 ±2.150
Typestate	0.968 ±0.050	1.162 ±0.143

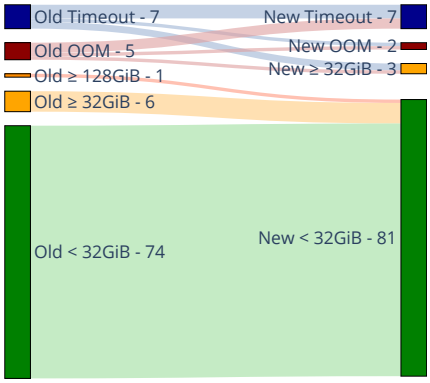

Figure 4 A sankey-plot showing how the number of (target program × analysis type) that finish with out-of-memory (OOM), timeout, or completed changes when switching from PhASAR's `IDESolver` (Old) to our `IDESolver++` (New) with JF_{old} keeping the time-limit of four hours and the memory limit of 250GiB.

Table 4 Results of our per-analysis comparision between the jump-function representations within our `IDESolver++`. We report the mean speedup and its standard deviation for both runtime and memory. Cells highlighted with green background indicate the JF with highest runtime speedup or memory savings for that analysis. In case, the highest speedup is <1 or the difference to the other jump-functions representations is not significant, we omit the highlight.

	JF1		JF2		JF3	
	Memory	Runtime	Memory	Runtime	Memory	Runtime
IIA	1.270 ±0.231	0.927 ±0.059	1.382 ±0.230	0.949 ±0.071	1.371 ±0.221	0.957 ±0.096
LCA	1.126 ±0.097	0.939 ±0.102	1.406 ±0.267	1.064 ±0.061	1.400 ±0.261	1.063 ±0.061
Typestate	1.059 ±0.053	0.996 ±0.023	1.057 ±0.042	1.013 ±0.035	1.057 ±0.042	1.005 ±0.022

Hence, we can already conclude that based on the high speedups for both runtime and memory as well as avoiding out-of-memory events, it is crucial to implement IDE in a performance-oriented way and just changing the implementation of the same underlying IDE algorithm can enable analyses that were not feasible before.

6.3.2 RQ_1: Jump-Functions Table Structure

We evaluated all three data structures JF_{ND}, JF_{N}, and JF_{NE}. We found that they behave differently depending on the target program and analysis. As expected, the instruction-interaction analysis puts a high load onto the solver, whereas the typestate analysis is very sparse and therefore completes within seconds.

Figure 5 shows both the runtime speedups and the memory savings of the different jump-functions representations compared to the deeply nested jump-functions representation JF_{old}. Both the runtime speedups and memory savings differ depending on the client analysis and have high variance over the target programs. In the (left) runtime speedup plot we can

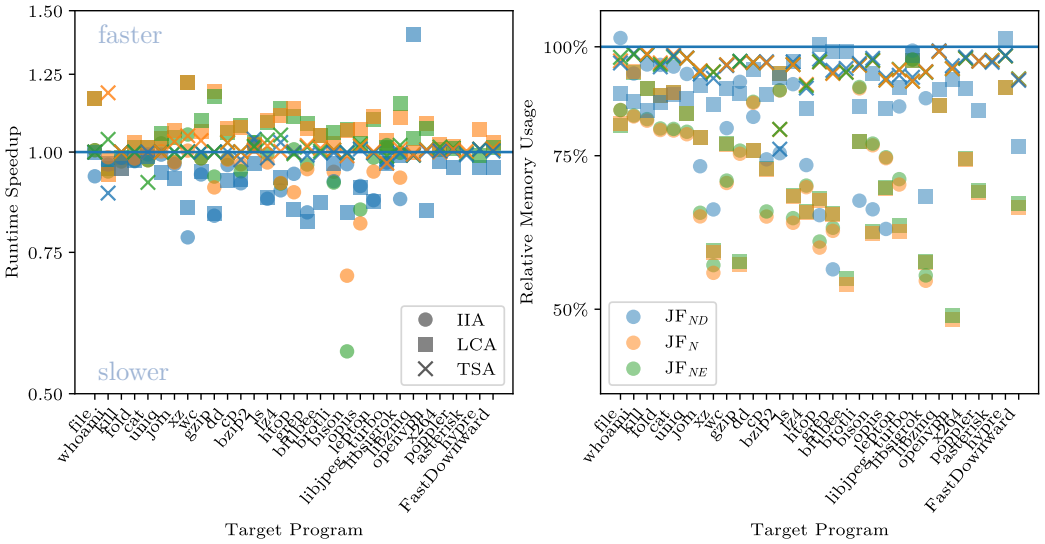

Figure 5 Scatter plots showing the `IDESolver++` with the proposed jump-functions representations compared to the `IDESolver++` using the nested representation inherited from PhASAR's current `IDESolver`. The left plot shows the runtime speedup (higher is better), whereas the right plot shows the relative memory usage (smaller is better). The target programs are sorted in ascending order based on their number of LLVM-IR instructions. The `IDESolver++` was configured to use JF$_{ND}$ (blue), JF$_{N}$ (orange), and JF$_{NE}$ (green). The both horizontal lines are set at 1 meaning no speedup. We use a log-scale to account for the non-linear distribution of speedups.

see that the speedups of the analyses are approximately centered around 1 with a small advantage of JF$_{N}$ and JF$_{NE}$ over JF$_{ND}$ for the LCA. In the (right) relative memory usage plot, it becomes visible that the IIA and LCA consume less memory with any of the proposed jump-functions representations than with JF$_{old}$. However, the variance across the analyzed target programs is high. For the TSA, the relative memory consumption is close to 94% for all jump-functions representations. The target programs in the plots of Figure 5 are sorted in ascending order by their number of LLVM-IR instructions. We provide variants of these plots with different program orderings on our supplementary website (see visualizations). Still, the orderings did not show observable correlations between the speedups and any of the tested program characteristics.

So, there is no clear overall "best" jump-functions table structure, and project- and analysis specific tradeoffs have to be made. However, by taking an analysis-centric view, we can determine the "best" jump-functions representation per analysis as shown in Table 4. For the IIA, JF$_{N}$ has highest average memory improvement with 1.382×(consuming 72% of the memory from JF$_{old}$), but the significance test shows that the difference to JF$_{ND}$ and JF$_{NE}$ is not significant, so in terms of memory, they share the first place. In terms of running time, JF$_{old}$ performed significantly best. For the LCA, JF$_{N}$ is best in terms of both runtime and memory improvements, consuming only 71% of the memory from JF$_{old}$ while being 6.4% faster; the difference to JF$_{NE}$ is not significant, so we consider both JF$_{N}$ and JF$_{NE}$ best for the LCA. While for memory improvement, JF$_{ND}$ is with using 97% of the memory slightly, but significantly better than JF$_{old}$, for runtime speedup, the difference between JF$_{ND}$ and JF$_{old}$ is insignificant. Finally, for the typestate analysis, the jump-functions representations performed similarly; yet the memory improvement of JF$_{ND}$, JF$_{N}$, and JF$_{NE}$ over JF$_{old}$ is significant, consuming around 94% of the memory from JF$_{old}$.

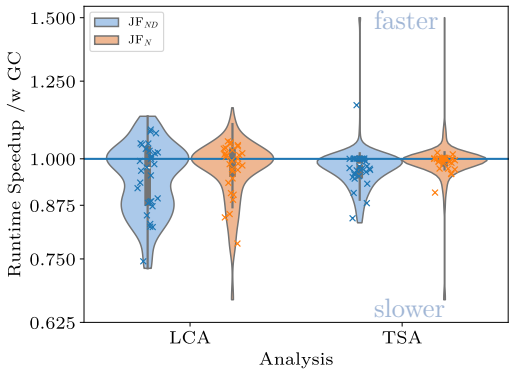

Figure 6a. A violin plot showing the runtime speedups of the `IDESolver++` with garbage collection compared to their versions without GC. The solver was configured to use JF_{ND} (blue) and JF_N (orange). Note, that the y-axis is in log-scale to account for the non-linear distribution of speedups <1 (slowdowns).

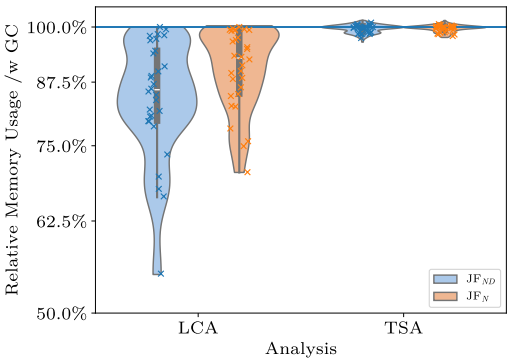

Figure 6b. A violin plot showing the relative memory usage of the `IDESolver++` with GC compared to its versions without GC. The solver was configured to use JF_{ND} (blue) and JF_N (orange). We use a log-scale for the relative memory usages here.

Figure 6 Violin plots showing the impact of enabling garbage collection on runtime and memory usage of the `IDESolver++`.

To answer RQ_1: The performance of the jump-functions representations highly depends on the performed analysis. However, JF_N and JF_{NE} have shown significantly best memory usage for the LCA and perform well for the IIA and TSA; this makes them a generally reasonable default choice. We also conclude that picking the right data structure oftentimes is *no* tradeoff between runtime speedup and memory savings; the same data structure can improve runtime and memory usage at the same time.

6.3.3 RQ_2: Jump-Functions Garbage Collection

The results of evaluating the jump functions garbage collector with JF_N are shown in Figure 6a and Figure 6b. For the LCA we see memory savings, where the analysis consumed, on average, 12% less memory (±10%). Furthermore, Figure 6b shows higher memory savings with JF_{ND} than with JF_N. For the TSA, the analysis with GC saved around 0.4% memory, which is significant, but we consider it negligible in most cases. This is expected because the TSA is very sparse and therefore does not have much to erase during garbage collection. Some analysis runs consumed even more memory than with disabled garbage collection. This is because of the additional book keeping meta-data that the garbage collector requires. In summary, the generalization to IDE indeed makes the GC less effective, but still it can drastically reduce the memory footprint of IDE analyses.

As expected, enabling jump functions garbage collection has non-negligible runtime-performance impact. The reason for this is that – in contrast to the experiments of Arzt [1] – the GC runs in the same thread as the analysis and therefore blocks the analysis process while performing the garbage collection. However, the mean speedup is close to 1 with 96.6% (±8.6%) for LCA and 98.3% (±6.3%) for TSA. Hence, the average runtime cost is still low.

Enabling the GC in single-threaded mode is a tradeoff between runtime and memory, as the GC reduces the memory consumption of IDE at the cost of increased runtime.

To answer RQ_2: Constraining the jump functions garbage collector to work in a single-threaded scenario results in a reduction of the memory consumption of the linear constant analysis of 12%, with only minimal runtime overhead. However, the effectiveness of the GC compared to the original GC from Arzt [1] is reduced, making it impractical for smaller analyses, and for those that do not propagate many data-flow facts.

6.4 Threads to Validity

Internal Validity

Runtime measurement on modern computing systems is a challenging task due to automatic clock boost and throttling as well as context switches enforced by the operating system. This makes reliable runtime measurements hard. We therefore ran our experiments three times and report averages to compensate for this noise. In addition, we ran each experiment in isolation on equivalent machines, ensuring that no other task is running in parallel. Our experiments each utilize only one thread to minimize the influence of the OS scheduler on the measurements.

We evaluated our experiments on a fixed set of target programs, on which we verified that the `IDESolver++` produces the same results as the `IDESolver`. We cannot rule out that there are programs where the solvers produce different results because of bugs in the implementations of either of them. To mitigate this risk, we performed our evaluation on a large set of real-world programs and configured the IDE solvers with three different client analysis problems.

External Validity

The performance of the analysis solvers may be different depending on the target program, that is, there may be programs that we did not benchmark where the analysis solvers behave differently. To mitigate this threat, we selected a diverse set of target programs from various domains and with different sizes and complexities. Furthermore, we configured the analysis solvers with three differently complex analysis problems to have greatest possible variation. This gives us for the first time a comprehensive study on a substantial number of real-world C/C++ programs.

6.5 Discussion

In Section 6, we presented the results of our evaluation, some of them require interpretation.

We have observed that JF_N in many cases has a lower memory consumption than JF_{ND}. This can be explained by the distribution of jump functions: For many analyses an extra experiment run with statistics instrumentation shows that the average size of the inner maps in JF_{ND} is < 4, but still with a high number of total jump functions. Hence, JF_{ND} pays the memory overhead of a hash map for the majority of jump functions, whereas JF_N and JF_{NE} oftentimes store more than 1000 elements in their inner maps which can lead to more efficient use of the provided memory.

On the other hand, depending on the access patterns of the jump-functions table, JF_{ND} can lead to faster jump functions access. For the IIA, we see drastic performance benefits of JF_{ND} and JF_{old} compared to JF_N and JF_{NE} when analyzing BISON. This can be explained by the handling of aliasing in the IIA. All aliases of a data-flow fact are propagated individually

in the IIA. Therefore, for memory-indirection statements, such as `store a to b`, for all aliases of the stored pointer a all aliases of the target pointer b must be generated, which are independent from each other. This leads to the same jump functions to be accessed multiple times, which may be faster if the inner maps do not incur memory indirections because they are small enough for small-size optimization.

Combining the measurements from our baseline (cf. Figure 3) with our specific optimizations from Section 4, we achieve the following overall mean speedups in the `IDESolver++` compared to PhASAR's current `IDESolver`: Memory improvements of 6.9× for IIA, and 2.7× for LCA; runtime speedups of 6.9× for IIA, and 4.9× for LCA. For the typestate analysis, there is no overall mean speedup, but also no mean slowdown.

7 Related Work

Performance problems of IDE implementations are a known issue. He et al. [9] perform sparsification on the ESG by propagating data-flow facts not along ICFG edges, but on their corresponding def-use chains. Arzt and Bodden [3] automatically generate IDE summaries for libraries, which prevents re-analyzing commonly used libraries and lowers the size of the analyzed target programs. Arzt and Bodden [2] improve re-analysis of already analyzed programs by incrementally analyzing only the changes compared to the previously analyzed version. These approaches let any existing implementation of IDE scale better in the circumstances that they optimize. Nonetheless, they can still further profit from an improved solver that scales better in the first place.

Weiss et al. [29] use a database system to store their internal data structures partially on disk effectively increasing the amount of available memory. However, they focus on the specific problem of error-code propagation and do not generalize to arbitrary IDE analyses. Hsu et al. [10] propose a modified IFDS algorithm that no longer needs to store the ESG explicitly and computes the reachability based on Depth-First Tree Intervals instead. While this approach works well for IFDS problems, it cannot be applied to IDE problems directly as composing edge functions requires to store the jump functions in some way.

He et al. [8] improve the garbage collection presented by Arzt [1] by increasing the GC's granularity from method-level to data-flow fact level. However, it suffers from the same restrictions of required multi-threading and also only applies to the same subset of IDE as the original garbage collector [1] that we generalize in this paper.

Apart from IDE, there are other approaches to precise interprocedural static data-flow analysis, such as weighted pushdown systems (WPDS) [12, 18]. As WPDS has the same runtime- and memory complexities as IDE, similar optimizations as the ones presented in this paper may be possible for WPDS as well. Other approaches, such as Boomerang [25] reduce their resource requirements by conducting demand-driven analyses, only computing the data-flow information for specific program locations. While demand-driven analyses work well for pointer analysis where a client analysis requests the demand, exhaustive taint analyses, e.g., a use-after-free analysis would need to issue a demand for each potential sink statement effectively degenerating the demand-driven analysis to a whole-program analysis with similar performance issues.

Yu et al. [31] tackle the performance problem by bringing data-flow analysis to the GPU and optimizing the algorithm, as well as the data-layout for GPU processing. As the CPU and GPU are particularly different hardware components, optimizations for GPU programs usually do not apply to CPU programs, and vice versa.

8 Conclusion

Current state-of-the-art IDE implementations do not scale well to large programs preventing the analysis of many interesting data-flow problems that can be used for bug- and vulnerability detection, as well as other important fields in software engineering. Based on years of experience with implementing and using IDE-based program analyses, we identified two different optimizations of the IDE algorithm. We found that choosing an efficient representation for the jump-functions table structure within the solver implementation has great influence on the performance of the algorithm. Still, it requires further research to select the right data structure for an analysis, or to even automate this process. Yet, we learned that an implementation of IDE has to be designed with performance in mind from the beginning to achieve a scalable implementation. Furthermore, we extended the jump functions garbage collection from Arzt to general IDE problems and removed the restriction to a multi-threaded solver implementation. We evaluated that it still reduces the memory footprint of the IDE analyses, though being less effective than the original.

Our experiments on 31 real-world C/C++ programs show runtime and memory speedups of up to 7× on average compared to the existing IDE implementation in PhASAR and enable the analysis of more target programs than before. We found that especially extremely heavy analyses such as the instruction interaction analysis presented by Sattler et al. [21] can now be run on medium-to large programs that was not possible previously, even with larger server hardware. Still, some analyses require too much memory for being executed on an ordinary developer machine.

References

1 Steven Arzt. Sustainable Solving: Reducing The Memory Footprint of IFDS-Based Data Flow Analyses Using Intelligent Garbage Collection. In *Proc. Int. Conf. Software Engineering (ICSE)*, pages 1098–1110. IEEE, 2021.

2 Steven Arzt and Eric Bodden. Reviser: Efficiently Updating IDE-/IFDS-Based Data-Flow Analyses in Response to Incremental Program Changes. In *Proc. Int. Conf. Software Engineering (ICSE)*, pages 288–298. ACM, 2014.

3 Steven Arzt and Eric Bodden. StubDroid: Automatic Inference of Precise Data-Flow Summaries for the Android Framework. In *Proc. Int. Conf. Software Engineering (ICSE)*, pages 725–735. ACM, 2016.

4 Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick D. McDaniel. FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Proc. Conf. Programming Language Design and Implementation (PLDI)*, pages 259–269. ACM, 2014.

5 Eric Bodden. Inter-Procedural Data-Flow Analysis with IFDS/IDE and Soot. In *Proc. Int. Workshop on State Of the Art in Java Program Analysis (SOAP)*, pages 3–8. ACM, 2012.

6 Eric Bodden. The secret sauce in efficient and precise static analysis: the beauty of distributive, summary-based static analyses (and how to master them). In *Comp. Proc. ISSTA/ECOOP Workshops*, pages 85–93. ACM, 2018.

7 Sigmund Cherem, Lonnie Princehouse, and Radu Rugina. Practical memory leak detection using guarded value-flow analysis. In *Proc. Conf. Programming Language Design and Implementation (PLDI)*, pages 480–491. ACM, 2007.

8 Dongjie He, Yujiang Gui, Yaoqing Gao, and Jingling Xue. Reducing the Memory Footprint of IFDS-Based Data-Flow Analyses using Fine-Grained Garbage Collection. In *Proc. Int. Symp. Software Testing and Analysis (ISSTA)*, pages 101–113. ACM, 2023.

9 Dongjie He, Haofeng Li, Lei Wang, Haining Meng, Hengjie Zheng, Jie Liu, Shuangwei Hu, Lian Li, and Jingling Xue. Performance-Boosting Sparsification of the IFDS Algorithm with Applications to Taint Analysis. In *Proc. Int. Conf. Automated Software Engineering (ASE)*, pages 267–279. IEEE, 2020.

10 Min-Yih Hsu, Felicitas Hetzelt, and Michael Franz. DFI: An Interprocedural Value-Flow Analysis Framework that Scales to Large Codebases. *Comput. Research Repository*, abs/2209.02638, 2022.

11 Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. Model checking for weakly consistent libraries. In *Proc. Conf. Programming Language Design and Implementation (PLDI)*, pages 96–110. ACM, 2019.

12 Akash Lal, Thomas Reps, and Gogul Balakrishnan. Extended Weighted Pushdown Systems. In *Proc. Int. Conf. Computer Aided Verification (CAV)*, pages 434–448. Springer-Verlag, 2005.

13 Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proc. Int. Symp. Code Generation and Optimization (CGO)*, pages 75–88. IEEE, 2004.

14 Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondrej Lhoták, José Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. In Defense of Soundiness: A Manifesto. *Commun. ACM*, 58(2):44–46, 2015.

15 Nomair A Naeem, Ondřej Lhoták, and Jonathan Rodriguez. Practical Extensions to the IFDS Algorithm. In *Proc. Int. Conf. on Compiler Construction (CC)*, pages 124–144. Springer-Verlag, 2010.

16 Oswaldo Olivo, Isil Dillig, and Calvin Lin. Static detection of asymptotic performance bugs in collection traversals. In *Proc. Conf. Programming Language Design and Implementation (PLDI)*, pages 369–378. ACM, 2015.

17 Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *Proc. Symp. Principles of Programming Languages (POPL)*, pages 49–61. ACM, 1995.

18 Thomas Reps, Stefan Schwoon, and Somesh Jha. Weighted Pushdown Systems and Their Application to Interprocedural Dataflow Analysis. In *Proc. Int. Symp. Static Analysis (SAS)*, pages 189–213. Springer-Verlag, 2003.

19 Atanas Rountev, Mariana Sharp, and Guoqing Xu. IDE Dataflow Analysis in the Presence of Large Object-Oriented Libraries. In *Proc. Int. Conf. on Compiler Construction (CC)*, pages 53–68. Springer-Verlag, 2008.

20 Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation. *Theor. Comput. Sci.*, 167(1-2):131–170, 1996.

21 Florian Sattler, Sebastian Böhm, Philipp Dominik Schubert, Norbert Siegmund, and Sven Apel. SEAL: Integrating Program Analysis and Repository Mining. *ACM Trans. Softw. Eng. Methodol.*, 32(5):121:1–121:34, 2023.

22 Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. PhASAR: An Inter-procedural Static Analysis Framework for C/C++. In *Proc. Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 393–410. Springer-Verlag, 2019.

23 Philipp Dominik Schubert, Richard Leer, Ben Hermann, and Eric Bodden. Know your analysis: How instrumentation aids understanding static analysis. In *Proc. Int. Workshop on State Of the Art in Program Analysis (SOAP)*, pages 8–13. ACM, 2019.

24 M Sharir and A Pnueli. *Two approaches to interprocedural data flow analysis*. New York Univ. Comput. Sci. Dept., 1978.

25 Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java. In *Proc. Europ. Conf. Object-Oriented Programming (ECOOP)*, pages 22:1–22:26. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016.

26 Yulei Sui, Ding Ye, and Jingling Xue. Static memory leak detection using full-sparse value-flow analysis. In *Proc. Int. Symp. Software Testing and Analysis (ISSTA)*, pages 254–264. ACM, 2012.

27 Yulei Sui, Ding Ye, and Jingling Xue. Detecting Memory Leaks Statically with Full-Sparse Value-Flow Analysis. *IEEE Trans. Software Eng.*, 40(2):107–122, 2014.

28 Erik van der Kouwe, Vinod Nigade, and Cristiano Giuffrida. DangSan: Scalable Use-after-free Detection. In *Proc. Europ. Conf. Computer Systems (EuroSys)*, pages 405–419. ACM, 2017.

29 Cathrin Weiss, Cindy Rubio-González, and Ben Liblit. Database-backed program analysis for scalable error propagation. In *Proc. Int. Conf. Software Engineering (ICSE)*, pages 586–597. IEEE, 2015.

30 Hua Yan, Yulei Sui, Shiping Chen, and Jingling Xue. Spatio-temporal context reduction: a pointer-analysis-based static approach for detecting use-after-free vulnerabilities. In *Proc. Int. Conf. Software Engineering (ICSE)*, pages 327–337. ACM, 2018.

31 Xiaodong Yu, Fengguo Wei, Xinming Ou, Michela Becchi, Tekin Bicer, and Danfeng Daphne Yao. GPU-Based Static Data-Flow Analysis for Fast and Scalable Android App Vetting. In *Int. Symp. Parallel and Distributed Processing (IPDPS)*, pages 274–284. IEEE, 2020.

32 Zhiqiang Zuo, Yiyu Zhang, Qiuhong Pan, Shenming Lu, Yue Li, Linzhang Wang, Xuandong Li, and Guoqing Harry Xu. Chianina: an evolving graph system for flow- and context-sensitive analyses of million lines of C code. In *Proc. Conf. Programming Language Design and Implementation (PLDI)*, pages 914–929. ACM, 2021.