A Programming Model for Semi-implicit Parallelization of Static Analyses

Dominik Helm
Florian Kübler
Jan Thomas Kölzer
helm@cs.tu-darmstadt.de
kuebler@cs.tu-darmstadt.de
jan.koelzer@stud.tu-darmstadt.de
Technische Universität Darmstadt
Department of Computer Science
Germany

Philipp Haller
phaller@kth.se
KTH Royal Institute of Technology
School of Electrical Engineering and Computer Science
Sweden

Michael Eichberg
Guido Salvaneschi
Mira Mezini
mail@michael-eichberg.de
salvaneschi@cs.tu-darmstadt.de
mezini@cs.tu-darmstadt.de
Technische Universität Darmstadt
Department of Computer Science
Germany

ABSTRACT

Parallelization of static analyses is necessary to scale to real-world programs, but it is a complex and difficult task and, therefore, often only done manually for selected high-profile analyses. In this paper, we propose a programming model for semi-implicit parallelization of static analyses which is inspired by reactive programming. Reusing the domain-expert knowledge on how to parallelize analyses encoded in the programming framework, developers do not need to think about parallelization and concurrency issues on their own. The programming model supports stateful computations, only requires monotonic computations over lattices, and is independent of specific analyses. Our evaluation shows the applicability of the programming model to different analyses and the importance of user-selected scheduling strategies. We implemented an IFDS solver that was able to outperform a state-of-the-art, specialized parallel IFDS solver both in absolute performance and scalability.

CCS CONCEPTS

• **Theory of computation → Program analysis**; *Parallel computing models*; • **Software and its engineering → Automated static analysis**.

KEYWORDS

static analysis, concurrency, parallelization

ACM Reference Format:
Dominik Helm, Florian Kübler, Jan Thomas Kölzer, Philipp Haller, Michael Eichberg, Guido Salvaneschi, and Mira Mezini. 2020. A Programming Model for Semi-implicit Parallelization of Static Analyses. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '20), July 18–22, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3395363.3397367

ISSTA '20, July 18–22, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-8008-9/20/07…$15.00
https://doi.org/10.1145/3395363.3397367

1 INTRODUCTION

Parallelization of static analyses is a promising avenue (a) for making analyses scale to ever-growing code bases, and (b) for shortening development cycles by reducing delays caused by build pipelines and continuous integration. By leveraging modern multi-core CPUs and GPUs, parallelization can also increase energy efficiency, e.g., when executing static analyses on virtualized cloud infrastructures. However, parallelizing static analyses is challenging, especially since advanced static analyses, e.g., general points-to [4], advanced call graph [8], or purity [14] analyses, are not data-parallel.

Relying on shared-memory threads and conventional synchronization primitives, such as locks and monitors, makes parallelization of static analyses cumbersome and requires significant expertise to ensure correctness, while, nevertheless, often not fully exploiting all available hardware parallelism [10].

Existing approaches to parallel static analysis have addressed the problem in two limited ways. Some have done so by targeting only specific high-profile analysis techniques [25, 26, 29], respectively specific frameworks [33]. Others require fundamental rethinking of the implementation strategy (e.g., avoiding explicit worklists) in order to adjust to the parallelism model of GPUs [25, 29]. For example, the approach of [25] requires (a) an adaptation of the data structures used by the analysis to the GPU memory model, (b) an explicit distribution of work to threads, and (c) novel concurrent algorithms amenable to the execution on GPUs. Enabling parallelization of static analyses *in a generic way*, i.e., not specialized to a single analysis or framework, and *implicit* [13], i.e., without requiring to restructure analysis algorithms or even devise completely new ones, remains an open challenge.

In this paper, we propose a new approach for deterministic parallel execution of static analyses. The approach offers a programming model not bound to any specific analysis kind (*analysis-independent*) and providing *semi-implicit parallelization* [23], i.e., not requiring invasive changes to common implementation strategies for static analyses in order to parallelize their execution. The exposure of the analysis writer to aspects relevant to parallelism is minimal; specifically, there is no need to (a) modify data structures used by the analysis or (b) to distribute work explicitly to threads.

The analysis-independent, semi-implicit parallelization is enabled by a declarative reactive programming model. Such a programming model provides high-level constructs for declaring dependencies between analysis results, while automating the propagation of updated analysis results across dependency chains.

All propagations are concurrent, except for those marked explicitly as sequential by the user if they share mutable state, thus avoiding non-determinism due to race conditions. In addition, the domain of analysis results is restricted to be a (semi-)lattice to ensure *determinism of concurrent updates*. This is, however, not really a restriction, since even complex, state-of-the-art static analyses compute results representable in bounded (semi-)lattices. Declaration of sequential updates and the requirement to use (semi-)lattices are the only aspects of parallelism exposed to analysis writers.

Declarative dependencies enable defining propagation strategies that are tuned to the domain of static analyses. For example, certain analyses benefit from a prioritization of updates where the more (source) dependencies a computation has, the lower the priority of that computation is; such a prioritization allows "batching" updates such that a larger number of more-up-to-date values is queried before a target computation is performed. Our approach also enables analysis writers to plug in analysis-specific strategies.

The contribution of this paper is to provide language abstractions that enable developers to formulate static analyses as a network of cells with dependencies among them. This approach allows the framework to handle concurrency semi-automatically while at the same time being applicable to a wide range of static analyses. Building on our previous work [10], we introduce four important concepts: (a) sequential cells that enable analyses to pass mutable state between callback invocations; (b) custom cell updaters that enable omitting costly join operations; (c) pluggable scheduling strategies that can be adapted to the needs of each analysis; and (d) aggregation of updates in order to reduce computation overhead.

The approach is implemented as a concurrency library in Scala, called Reactive Async 2 (RA2). We open-sourced the implementation on GitHub.[1]

We evaluate RA2 along two dimensions. First, we give evidence on the approach being analysis-independent and enabling semi-implicit parallelism by applying it to two different kinds of analyses (see Section 3.1), a purity analysis and the IFDS framework [32].

Second, we empirically evaluate the performance and scalability of RA2 (Sections 3.2 to 3.4). Our evaluation shows that the performance scales well w.r.t. the number of processor cores. With 10 threads on a 10-core CPU, we achieve speed-ups greater than 4.8x and, hence, successfully parallelize over 88% of the computation according to Amdahl's law. A comparison of different scheduling strategies shows performance differences up to 1100%, suggesting that the choice of scheduling strategy is of great importance. On a single core, our parallel IFDS is only 28% slower than an optimized sequential implementation using the fixed-point computation framework provided by the static analysis framework OPAL [6, 28]. Already with 2 threads the parallel implementation runs 1.3x faster than OPAL. Compared to the state-of-the-art IFDS solver Heros [3, 15], RA2 has comparable single-thread performance (RA2 is 15% faster) and consistently achieves significantly higher speed-ups. While Heros achieves a maximum speed-up of 2.36 using 8 threads, the speed-ups of RA2 are 3.53 using 8 threads and continues to improve for higher thread counts. A thorough comparison to an actor-based parallel IFDS solver [33] was unfortunately not possible as the implementation of that system is not available, but the speed-ups are comparable.

The rest of the paper is organized as follows: Section 2 presents the programming model including an in-depth example of a simple analysis. The approach is evaluated in Section 3. Section 4 presents potential threats to validity to our evaluation. We conclude the paper discussing related work in Section 5 and providing a summary and outlook in Section 6.

2 APPROACH

In this section, we present our approach to analysis-independent and semi-implicit parallelization of static analyses. We start with a high-level overview of the programming model that the approach imposes (2.1) followed by advanced concepts to ensure correctness in the presence of concurrent updates and shared mutable state (2.2). Next, we introduce RA2's solver, which performs the parallelization and resolves (cyclic) dependencies (2.3). We present pluggable scheduling strategies, which enable analysis-specific tuning and aggregation of updates with the goal to improve on performance (2.4) and finally give an in-depth example of a simple purity analysis implemented in RA2 (2.5).

2.1 Programming Model Basics

To illustrate how analyses are implemented in RA2, assume we want to develop a method purity analysis determining whether a method is deterministic and free of side-effects (see, e.g., [14]).

The goal of any static analysis is to compute a specific property for a given entity. In the example of the purity analysis, methods are the entities and the property is the purity, which can have one of two values: pure or impure. In RA2, each property must form a lattice (or at least a partial order with a bottom element), common data structures for properties calculated by static analyses. The property of an entity may depend on some other properties of related entities. In a purity analysis, if `foo` calls `bar`, the computation of `foo`'s purity requires the purity of `bar`. We say `foo` *depends on*/*is a depender of* `bar`, or `bar` *is a dependee of* `foo`.

In RA2, analyses are implemented as two functions—an *initial analysis function* and a *continuation function*. Both are invoked by the reactive framework underlying RA2.

Given an entity, the initial analysis function computes the initial property of that entity based on the information already available, i.e., local information, such as the source code, and the current property values of dependees. The initial analysis function also queries and collects the dependencies of an entity. The dependencies may have a non-final property value (or none at all) at the time the initial function is executed. The list of dependencies is used to get notified of potential future changes of their property values. These changes may in turn lead to updates of the entity's property value. The initial analysis function returns the initial property and registers the dependencies along with a continuation function.

A continuation function of an entity e is invoked by RA2 whenever dependencies of e (e.g., the purity information for a called

[1] https://github.com/phaller/reactive-async

method) change. Its result defines how the property value of e is to be updated, whereby the updated value must be a monotonic refinement according to the property's lattice. Unless the continuation function declares its result as final, it will be invoked again for further updates of dependees. Both the initial analysis and invocations of the continuation functions are *tasks* that RA2 executes concurrently.

Analysis results and dependencies are maintained in *cells*. When implementing a static analysis, we create one cell per pair of analyzed entity and property. Cells are shared by the different concurrent tasks. Every cell is explicitly associated with the *lattice* of the property values that it manages (e.g., the cells in Listing 1 are associated with the `Purity` lattice). A cell that was created to store purity information therefore cannot be used—at some later point in time—to store data-flow information.

Dependencies are created by connecting two cells using the continuation function—to respond to updated cells, continuation functions are used as callbacks. To ensure that a cell is notified about the update of its dependees, its dependencies are explicitly declared and registered using the `when` method. For instance, `cell2.when(cell1)(continuation)` registers `cell1` as a dependee of `cell2`, the function `continuation` is called *when* the value of `cell1` changes.

The continuation function processes the changed dependee's (`cell1` in our case) new value and returns an `Outcome` object, which decides whether and how the dependent cell (`cell2` in our case) should be updated. RA2 provides three types of `Outcome` objects: A `NextOutcome(v)` result means that the dependent cell should be updated with value `v` according to the specified updater (cf. Section 2.2). `FinalOutcome(v)` states that additionally this update is final. If `NoOutcome` is returned, the value of `cell2` is not changed at all.

To illustrate the use of cells and continuations, consider Figure 1 which graphically depicts the cells and dependency from Listing 1 and the propagation of an update for this dependency. Assume that the two cells, `cell1` and `cell2`, have been initialized with `Pure` (A) through the use of `NextOutcome`, i.e., their values can still change. In Lines 4 to 7, a dependency between `cell1` and `cell2` is introduced (B). Recall that for a purity analysis, this is necessary if the method represented by `cell2` invokes the one represented by `cell1`. Whenever `cell1` is updated (using `FinalOutcome` or `NextOutcome`), the continuation is executed and the returned `Outcome` is used to update `cell2`. Consider the case when a final update for `cell1`—with the value `Impure`—occurs (C). This will cause the continuation to be invoked with `Impure` as an argument (the new value of `cell1`) (D). Since a method that calls an impure method is also impure, the continuation returns a `FinalOutcome` with value `Impure`. Thus, `cell2` is completed with value `Impure` (E), and the dependency is removed (F). If the continuation returned a `NextOutcome`, the update of `cell2` would be an intermediate one.

2.2 Advanced Constructs for Correctness

To ensure correctness and termination, cell updates must have a well-defined semantics. RA2 executes updates concurrently, and thus without a guaranteed order. Yet, monotonic updates ensure determinism regardless of the order. Mutable stated shared between the continuations of a cell may also lead to non-determinism when the updates are executed concurrently.

Listing 1: Example of dependencies and continuations

```
1 val cell1: Cell[Purity] = ...
2 val cell2: Cell[Purity] = ...
3
4 cell2.when(cell1) { update =>
5   if (update.head.get.value == Impure) FinalOutcome(Impure)
6   else NoOutcome
7 }
```

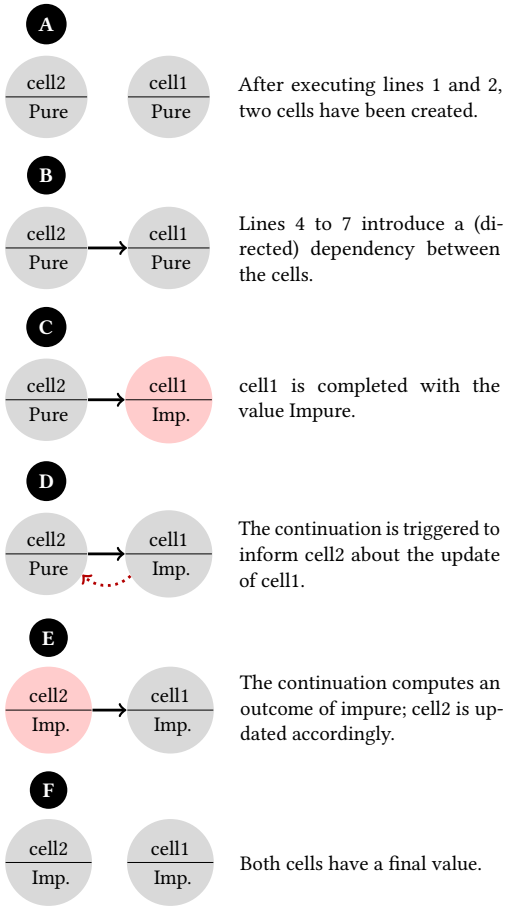

Figure 1: Execution of Listing 1

Monotonic Updates. To guarantee determinism of cell updates, they must be monotonically increasing according to the underlying lattice. RA2 therefore encapsulates cell updates in so-called *updater* objects that determine how updates for the cell are processed.

Monotonicity can be automatically guaranteed by using the *join* operator of a cell's lattice to aggregate new values with the previous value of the cell. The outcome returned by each continuation is joined with the cell's previous value to compute the least upper bound, which then becomes the cell's new value. RA2 provides the updater type `AggregationUpdater` that offers this semantics. However, there are several good reasons for supporting other updater semantics. In general, performing joins can be expensive when dealing with lattices that are not based on singleton values, e.g.,

sets. Complex analyses may already have to perform the *join* explicitly as part of the continuation function, thereby guaranteeing the monotonicity by design and making another implicit join during the cell update obsolete. To cover such needs, RA2 provides the `MonotonicUpdater` as a drop-in replacement for the `AggregrationUpdater`.

The `MonotonicUpdater` does not perform a *join* operation, but only checks whether the given update fulfills monotonicity. This check is defined by the lattice. For expensive checks, it can be used only during development and simplified or disabled in production when the analysis is known to guarantee monotonicity. The `MonotonicUpdater` also allows for using partial orders instead of lattices. The partial orders, however, still require a *bottom element* and must fulfill the ascending chain condition, i.e., monotonically increasing operations must converge eventually – we use this kind of updater, e.g., for our IFDS solver (see Section 3). As the IFDS solver's computations are required to be performed sequentially and only introduce additional flow edges, updates are guaranteed to be monotonic and no other (potentially expensive) join operation is required.

Which updater should be used for a specific cell can be defined when creating the cell by the `HandlerPool` – the interface of the analysis implementations to the underlying reactive system of RA2, which is presented in Section 2.3.

Sequential Updates. Shared mutable state affected by updates needs to be thread-safe. Advanced analyses require maintaining mutable state between cell updates. For example, an IFDS analysis keeps track of already computed path edges in order to extend them once updates on the analyzed method's callees become available. Mutable state can also be used to explicitly keep track of the set of dependencies. Updates that affect such mutable state need to be sequential. Otherwise, continuations for several incoming updates could be executed concurrently, leading to non-determinism in the presence of mutable state due to race conditions.

To enable the thread-safe use of mutable state, RA2 provides cells with *sequential updates*, whose continuations are ensured to run sequentially. As such, cells with sequential updates free developers from the need to use locks or other concurrency mechanisms to protect shared mutable state. The example in Listing 2 illustrates cells with sequential updates. Those cells are created using the `mkSequentialCell` method of the `HandlerPool` (cf. Section 2.3). While `dependee1` and `dependee2` may be updated concurrently, RA2 ensures that all callbacks targeting `seqCell1`—`continuation1` and `continuation2`—are invoked sequentially (though with no guarantee on the relative order). Hence, both callbacks may safely access shared state – as long as that state is only shared among callbacks targeting `seqCell1`. To still exploit the benefits of parallel computations, `continuation3` is run concurrently, even with `continuation1` being triggered by the same dependee, `dependee1`. Hence, `continuation1` and `continuation2` must not share state with `continuation3`, as the latter targets a different cell.

2.3 Handler Pool

A central unit of RA2 is the `HandlerPool`. It creates cells and manages the execution of initial functions and the propagation of updates along cell dependency chains by executing respective continuations. It implements the parallelization which analyses can use out-of-the-box and do not need to implement on their own. For each initial

Listing 2: Example of sequential updates

```
1 val seqCell1 = pool.mkSequentialCell(...)
2 val seqCell2 = pool.mkSequentialCell(...)
3
4 seqCell1.when(dependee1)(continuation1)
5 seqCell1.when(dependee2)(continuation2)
6 seqCell2.when(dependee1)(continuation3)
```

analysis function and each triggered continuation, the runtime system creates a task that is eventually executed by an idle thread. The `HandlerPool` keeps track of all active threads and all tasks being registered for execution and schedules their execution.

The `HandlerPool` is RA2's fixed-point solver, as such it also resolves situations where the execution gets stuck. As it keeps track of running and pending tasks, the pool is able to detect *quiescence*. The system is quiescent when there are no unfinished submitted tasks currently queued or running. However, even when quiescence is reached, not all cells are guaranteed to contain final results. This is true in the following cases:

(1) A chain of dependencies such that each cell's result depends on another cell's result where these dependencies form a cycle is called *cyclic dependency*. A simple example are two cells A and B, where A depends on B and B depends on A. Such a cyclic dependency where each cell in the cycle solely depends on cells of that cycle is called a closed strongly connected component (cSCC).
(2) Cells that do not depend on any other cells are called *independent*. Independent cells that have not been completed are referred to as *independent unresolvable cells* (IUC). An IUC arises when all dependees of a cell have been completed (and the corresponding dependencies have therefore been dropped), but the cell itself was not completed yet. This happens if on the last invocation of the continuation, that continuation—without tracking dependencies explicitly—cannot recognize that there will not be any future invocations. In the example of Figure 1, `cell2` becomes independent after the final update of `cell1` as the dependency between `cell2` and `cell1` is removed by the system (F). This shows the case, when `cell2` is completed. But, if `cell1` had instead been completed with a result property value *pure*, the continuation would have returned `NoOutcome` and, therefore, `cell2` would not be completed. As the dependency would still be removed, `cell2` would become an IUC.

Once cycles and independent cells are detected, two methods, that are implemented by the analysis designer when specifying the lattice, are used to resolve them: `resolve` and `fallback` The `resolve` method takes a list of all cells in one cSCC and returns for each cell the final value it should be resolved to. Resolving one cSCC does not trigger callbacks of cells in that cSCC, but does so for their dependers outside of it. The `fallback` method works similarly, but is given a list of IUCs. Different to the cells of a cSCC, the IUCs must be resolved independently of each other. Each cell is then completed with the returned associated value, which is typically the cell's current value.

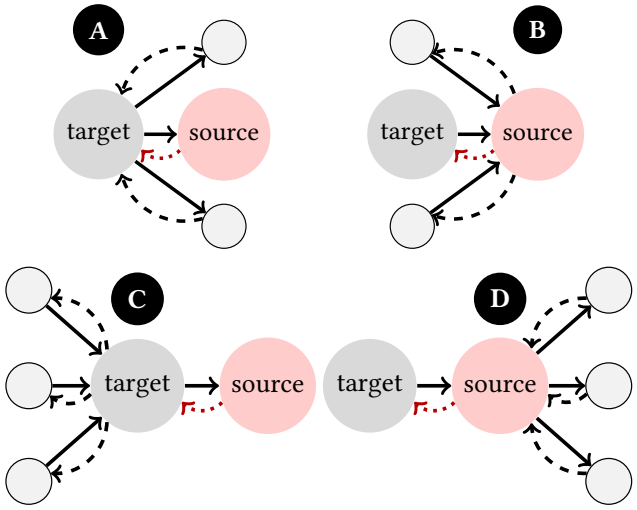

Figure 2: Scheduling Strategies

2.4 Scheduling

The order in which individual tasks are scheduled can be of utmost importance in order to provide effective parallelization for a specific (analysis) domain [33]. In the case of IFDS-A, scheduling strategies that prioritize values that might have a bigger impact were shown to be beneficial. Thus, different scheduling strategies are needed.

The `HandlerPool` is parametric in the scheduling strategy to enable an analysis designer to plug in a scheduling strategy to influence the order in which tasks ready for execution are picked up. Whenever a task is submitted to the pool for execution, the scheduling strategy is invoked to calculate a priority for the dependency in question. This priority is used in a priority queue that tracks all tasks that may be executed concurrently.

Dependency continuations that target sequential cells must not run concurrently. To ensure this, each sequential cell keeps track of all tasks updating it. Again, this is done via a priority queue that uses the developer-supplied scheduling strategy. Tasks can be dequeued from this queue in the order of respective priority, hence respecting the scheduling strategy. ues,

Besides a default last-in-first-out scheduling strategy with first-in-first-out work stealing, RA2 provides several other general-purpose scheduling strategies out-of-the-box. These are applicable to any kind of analysis as they only take into account how many dependees a cell has and how many cells (directly) depend on the value that is being updated via the continuation. These strategies are illustrated in Figure 2 (A–D). As in Figure 1, straight arrows are dependencies, while discontinuous arrows represent potential update messages. The strategies determine the priority of the message from `source` to `target` (dotted red message in Figure 2).

(A) **`TargetsWithManySourcesFirst/Last`.** The higher the number of cells (in addition to `source`) that `target` depends on, the higher/lower the priority.

(B) **`SourcesWithManyTargetsFirst/Last`.** The higher the number of cells depending on `source` (in addition to `target`), the higher/lower the priority.

Listing 3: A scheduling strategy for lattice values

```
case class LatticeValueStrategy[V](prioritizedValue: V)
  extends SchedulingStrategy[V] {
  override def calcPriority(
    tgt: Cell[V], src: Cell[V], v: Outcome[V]
  ): Int = calcPriority(tgt, v)

  override def calcPriority(
    tgt: Cell[V], v: Outcome[V]
  ): Int = v match {
    case FinalOutcome(prioritizedValue) => -1
    case _ => 1
  }
}
```

(C) **`TargetsWithManyTargetsFirst/Last`.** The higher the number of cells depending on `target`, the higher/lower the priority.

(D) **`SourcesWithManySourcesFirst/Last`.** The higher the number of cells that `source` depends on, the higher/lower the priority.

The generic-purpose strategies are simple and only take the (local) shape of the dependency graph into account. Nonetheless, they can influence the analysis' behavior significantly. Some strategies (the `xFirst` ones) try to prioritize updates that may have a greater impact on the result by influencing many cells. This may lead to faster stabilization of the result. Other strategies (the `xLast` ones) delay such influential updates, providing more opportunities to aggregate them (cf. below), potentially reducing the overall number of updates required. Our evaluation (cf. Section 3.2) shows that this has a significant impact on scalability and performance.

Scheduling strategies also allow to encode and take into consideration domain-specific knowledge about the relevance of different property values. For our purity analysis, propagating `Impure` may be more relevant to target cells than propagating `Pure`. This is because a call to an impure method is impure, thus a single impure dependee will result in the cell being completed. In contrast, a call to a pure method does not change the outcome as long as there are other dependees that might be impure. To accelerate the propagation of specific lattice values, we can use a strategy that returns a higher priority for the respective continuations.

The implementation of such a strategy prioritizing a given lattice element is shown in Listing 3. RA2 provides two functions to calculate the priority for scheduling: the first one is used in scheduling continuations while the second one is used for resolving cycles and IUCs where no source cell exists. Note that, based on Java's priority queues, a low value will prioritize the task to be scheduled early.

As the invocation of continuations may be delayed according to prioritization, the cell that triggered a continuation may be updated again before the continuation is actually invoked. This enables an important optimization: instead of invoking the continuation with the first updated value and later invoking the continuation again, the continuation will be invoked only once using the most recent value. RA2 can also aggregate updates from multiple sources that are passed to the continuation as a list of updates. Both kinds of aggregations can reduce the overhead of continuation invocations. This also has a transitive effect: less invoked continuations produce less intermediate results that in turn trigger less continuations.

2.5 Reactive Async 2 at Work

In the following, we demonstrate how to apply the proposed programming model to implement a very simple purity analysis. We show the complete analysis, including the lattice definition, the initial analysis function, its continuation, and how to bootstrap the analysis, leaving out just minor details, e.g., error handling.

Listing 4: A simple lattice for purity information

```
1  sealed trait Purity
2  case object Pure extends Purity
3  case object Impure extends Purity
4
5  object Purity {
6    implicit object PurityLattice extends Lattice[Purity] {
7      override def join(v1: Purity, v2: Purity): Purity = {
8        if (v1 == Impure) Impure else v2
9      }
10
11     override val bottom: Purity = Pure
12   }
13 }
```

Listing 4 shows the specification of the lattice, including the bottom value and the *join* function. The lattice has two elements, namely `Pure` (its bottom element) and `Impure`.

The `analysis` function shown in Listing 5 computes the purity of a given method by checking its instructions. If there is an instruction affecting the purity, e.g., a static field write, the method is immediately considered impure. Additionally, we consider native methods, methods with reference type parameters, or non-monomorphic (i.e., virtual and interface) method calls as impure. For method calls that are not self-recursive, the callee's cell is added as a dependency. If such callee would be impure, the analyzed method itself would be impure. With a call graph available, polymorphic calls could simply be handled by adding a dependency for each possible callee. After checking all instructions, the method is found to be pure, and can only be refined to impure by an update of a dependency. To cover the latter case, we use `when` to register the continuation function to react on updates for these dependencies.

The continuation function handling updates of dependencies is shown in Listing 6. As stated above, impure callees lead to an impure method. We take advantage of RA2's aggregation of updates: The continuation function may receive updates for multiple dependees at once and if a single one is impure, the cell is completed.

Listing 7 shows the fallback and cycle-resolution strategies as explained in subsection 2.4. For the purity analysis, all impure values will be marked as final immediately. Therefore, unresolved cells must be `Pure` for both, cycles and IUCs.

Listing 8 shows how to (1) initialize and (2) start the analysis, (3) await its completion, and (4) retrieve the results. Parallelization is done by the `HandlerPool` – the number of threads is set explicitly in this example. We also specify the scheduling strategy here, using the lattice-based strategy from Listing 3 to prioritize updates of of impure methods. `Cell`s are created through the `HandlerPool` and their initial analysis is then triggered. In particular, note how we specify the `AggregationUpdated` for each cell (line 11). This explicit specification is for demonstration only, as aggregating updates is the default in RA2. Triggering the cells starts the concurrent computation of the initial analysis functions. After quiescence has been reached, the cells contain the final results.

Listing 5: Determine the purity of a method

```
1  def analyze(method: Method): Outcome[Purity] = {
2    val cell = methodToCell(method)
3
4    if (method.isNative || method.hasReferenceTypeParameter())
5      return FinalOutcome(Impure)
6
7    val dependencies = mutable.Set.empty[Cell[Purity]]
8    for (instruction <- method.instructions) {
9      instruction match {
10       case gs: GETSTATIC =>
11         resolveFieldReference(gs) match {
12           case Some(field) if field.isFinal =>
13             /* Constants do not impede purity */
14           case _ =>
15             return FinalOutcome(Impure)
16         }
17
18       /* For simplicity: only handle monomorphic calls */
19       case INVOKESPECIAL | INVOKESTATIC =>
20         resolveNonVirtualCall(instruction) match {
21           case Success(callee) =>
22             /* Self-recursive calls do not impede purity */
23             if (callee != method)
24               dependencies.add(methodToCell(callee))
25
26           case _ /* Unknown callee */ =>
27             return FinalOutcome(Impure)
28         }
29
30       case NEW | PUTSTATIC | ... =>
31         return FinalOutcome(Impure)
32
33       case _ =>
34         /* All other instructions are pure. */
35     }
36   }
37
38   if (dependencies.isEmpty) {
39     FinalOutcome(Pure)
40   } else {
41     cell.when(dependencies)(continuation)
42     NextOutcome(Pure)
43   }
44 }
```

Listing 6: Continue with updates for callees

```
1  def continuation(
2    v: Iterable[(Cell[Purity], Outcome[Purity])]
3  ): Outcome[Purity] = {
4    if (v.exists(_._2 == FinalOutcome(Impure)))
5      FinalOutcome(Impure)
6    else
7      NoOutcome
8  }
```

3 EVALUATION

Our evaluation aims to answer the following research questions:

- RQ1: Is RA2 applicable across different kinds of static analyses to enable semi-implicit parallelization?

Listing 7: Resolving cycles and IUCs

```
object PurityKey extends Key[Purity] {
  def resolve(
    cells: Iterable[Cell[Purity]]
  ): Iterable[(Cell[Purity], Purity)] = {
    cells.map(cell => (cell, Pure))
  }

  def fallback(
    cells: Iterable[Cell[Purity]]
  ): Iterable[(Cell[Purity], Purity)] = {
    cells.map(cell => (cell, Pure))
  }
}
```

Listing 8: Setting up and starting the analysis.

```
def main(project: Project): Unit = {
  // 1. Initialize HandlerPool and Cells
  val pool: HandlerPool[Purity] = new HandlerPool(
    key = PurityKey,
    parallelism = 10,
    schedulingStrategy = LatticeValueStrategy(Impure)
  )
  var methodToCell = Map.empty[Method, Cell[Purity]]
  for (method <- project.allMethods) {
    val cellCreator = pool.mkCell(_ => analyze(method))
    val cell = cellCreator(AggregationUpdater)
    methodToCell += method -> cell
  }

  // 2. Start analyses
  for (method <- project.allMethods) {
    methodToCell(method).trigger()
  }

  // 3. Wait for completion
  val fut = pool.quiescentResolveCell()
  Await.ready(fut, 30.minutes)
  pool.shutdown()

  // 4. Retrieve results
  val pureMethods =
    methodToCell.filter(_._2.getResult() == Pure).keys
}
```

- RQ2: How do scheduling strategies affect performance?
- RQ3: What is the overhead compared to sequential analyses?
- RQ4: How does the RA2-based IFDS implementation compare to other state-of-the-art IFDS analyses?

For the evaluation, we used a machine with an Intel(R) Core(TM) i9-7900X CPU @ 3.30GHz (10 cores / 20 threads) and 128 GB RAM. The OS running is Ubuntu 18.04.3. The analyses were run using OpenJDK 1.8_212. The JVM was started with 16 GB of heap memory (`-Xmx16G`). We analyzed the JRE 1.7.0 update 95 from the publicly available Doop benchmarks project [5] to ensure repeatability. The purity analysis was executed on the complete JRE, while the IFDS analysis was executed on the runtime jar only. For each experiment we report the median runtime of seven executions.

While RA2 itself is framework independent, we used the OPAL framework [6] to provide bytecode parsing and an intermediate representation for our analyses to work upon.

3.1 RQ1: Applicability

To answer RQ1, we take another look at our purity case study from Section 2.5. As it shows, the implementation of a simple purity analysis, including definition of the lattice and an execution harness takes just about 80 lines of code (excluding whitespace and comments). We observe that RA2's programming model allows implicit parallelization in this case: No explicit handling of parallelization, e.g., creation of threads, locks, or tasks, is required by the actual analysis[2]. In particular, no rethinking of the algorithm is required, as it is, e.g., for GPU-based solutions [25, 29] to map the execution to the parallel hardware. Extending the analysis to a more powerful purity analysis would require changes only to the analysis and continuation functions (and potentially an extension of the lattice for a finer granularity) and would not introduce any additional complexity related to the parallelization.

For a more complex analysis, we implemented a solver[3] for IFDS. This implementation was adapted from the IFDS solver in the OPAL framework with only minor changes to support the different dependency handling. IFDS (cf. Appendix) is a general framework for dataflow analyses and has been implemented numerous times (e.g., in WALA [39], Heros [3], Flix [21] or IFDS-A [33]). It has been parallelized in the past and we evaluate our performance against the state-of-the-art IFDS solver Heros. IFDS-A's implementation was never publicly available, however. WALA and Flix do not even have a parallel implementation. Flix's implementation in particular is only a proof-of-concept that ceased to work with current versions of Flix.

Our implementation makes use of `MonotonicUpdater`s, which reduces the number of large set operations. It requires sequential updates as it maintains mutable state between the continuation invocations to keep track of dataflow edges already known. Thus, the parallelization is semi-implicit in the case of the IFDS solver.

Using our IFDS solver, we implemented a taint analysis as a client analysis used in the rest of the evaluation. This analysis is inspired by FlowTwist [20] and identifies public or protected methods in the Java Runtime 7 (`rt.jar`) with return type `java.lang.Object` or `java.lang.Class` that have a string parameter that is later used in an invocation of `java.lang.Class.forName`. If such flows are found, attackers may get the ability to load a class of their choice. The analysis tracks local variables and is field-sensitive.

RA2 is able to provide (semi-)implicit parallelization to the simple purity analysis as well as the significantly more complex IFDS solver. Both analyses are very different in their kind and there is no specialized support for any of them implemented in RA2, indicating that RA2 is applicable to different kinds of static analyses.

3.2 RQ2: Scheduling Strategies

To answer RQ2, we evaluated different scheduling strategies for both the IFDS and purity analysis introduced above. The execution times reported are for ten threads each, corresponding to the number of physical cores of our system.

IFDS Analysis. We evaluated the performance for the analysis-independent strategies. Table 1 shows the median execution times

[2] Only within the `main` function one `Future` needs to be awaited.
[3] https://github.com/phaller/reactive-async/tree/v2.0.0/core/src/test/scala/com/phaller/rasync/test/opal/ifds

Table 1: Performance of scheduling strategies for IFDS
Speed-up shown compared to (a) default and (b) slowest strategy.

Strategy	Run time [s]	Speed-up (a)	Speed-up (b)
DefaultScheduling	27.29	0.00%	13.3%
SourcesWithManyTargetsLast	21.00	23.1%	33.3%
TargetsWithManySourcesLast	19.84	27.3%	37.0%
TargetsWithManyTargetsLast	29.31	-7.40%	6.86%
SourcesWithManySourcesLast	21.40	21.6%	32.0%

for each strategy when using ten threads. The percentages show the speed-up of each strategy compared to (a) the default strategy and (b) the slowest strategy.

We did not measure the `xFirst` strategies, as they were determined to perform poorly for IFDS. For example, `TargetsWithManySourcesFirst` took more than 900 seconds using a single thread, thereby performing more than 1100% worse than its counterpart, the best-performing `TargetsWithManySourcesLast` strategy.

The data shows that using a suitable scheduling strategy can have a significant impact on execution time. Considering the evaluated strategies, `TargetsWithManySourcesLast` is the best strategy for our IFDS analysis. It is 37.0% faster than the worst strategy presented here (excluding the above-mentioned poorly performing strategies) and 27.3% faster than the default. Relative standard deviations (RSD) for the reported measurements are between 4.0% and 10.4%. Figure 3 gives a graphical representation of the performance of the different scheduling strategies for different thread counts.

The effect of each strategy is application-dependent and may differ with the number of cells, the number of dependencies and cycles, and the costs of the used continuation functions. The advantage of `TargetsWithManySourcesLast` for the IFDS analysis can be explained by considering the aggregation of results; i.e., avoiding notifications of dependers. In the case of cells with many sources it pays off to hold back the update as long as possible, because this potentially allows aggregation with updates from other sources. Recall that before a continuation is actually invoked, the most current value(s) is(are) queried again and passed to the continuation in an aggregated form.

That way, the target cell can compute its result on a larger batch of information in one step as opposed to multiple small steps, which would be needed, if the cell was informed prematurely. This strategy works well for IFDS, because all propagations need to be handled in the same way and there are no special values, which would lead to early finalization of cells and could therefore be advantageous, as in the case of the purity analysis.

Purity Analysis. In addition to the analysis-independent strategies, we used the previously introduced `LatticeValueStrategy` adapted to the purity analysis. It makes use of the specific effect an update of a source cell may have on a target cell. If a dependee is *impure*, we can immediately decide on the purity of the depender and complete it with the value *impure*. The strategy gives such propagations a high priority as they lead to final results quicker. In contrast, if a cell is completed with *pure*, a target cell can *just* kill the dependency, because that update will not affect the current cell's value.

Table 2: Runtimes and speed-ups for the Purity Analysis
Speed-up shown compared to (a) default and (b) slowest strategy.

Strategy	Run time [s]	Speed-up (a)	Speed-up (b)
DefaultScheduling	0.42	0.00%	40.8%
SourcesWithManyTargetsFirst	0.70	-66.7%	1.41%
SourcesWithManyTargetsLast	0.70	-66.7%	1.41%
TargetsWithManySourcesFirst	0.71	-69.0%	0.00%
TargetsWithManySourcesLast	0.71	-69.0%	0.00%
TargetsWithManyTargetsFirst	0.69	-64.3%	2.82%
TargetsWithManyTargetsLast	0.68	-61.9%	4.23%
SourcesWithManySourcesFirst	0.68	-61.9%	4.23%
SourcesWithManySourcesLast	0.69	-64.3%	2.82%
LatticeValueStrategy	0.71	-69.0%	0.00%

Table 2 shows the results for all strategies using ten threads along with their relative speed-ups. The `DefaultStrategy` is significantly faster than the other strategies for this experiment. The other strategies, including the `LatticeValueStrategy` that prioritizes impure updates, show no significant differences in runtime. Relative standard deviations are between 1.0% and 6.1%. The `HandlerPool` uses a `java.util.concurrent.ThreadPoolExecutor` for pluggable scheduling strategies as this allows the necessary priority queue to be supplied. For `DefaultStrategy`, however, a `java.util.concurrent.ForkJoinPool` is used that already implements a work stealing LIFO queuing scheme. We believe that the `DefaultStrategy` is faster, because the continuation tasks created by the simple purity analysis are extremely small. This gives the default `ForkJoinPool` an advantage over the more complex `ThreadPoolExecutor` used for the other strategies.

The purity and IFDS analyses benefits from different strategies, emphasizing the need for pluggable, user-supplied strategies. While the analysis specific strategy did not benefit the simple purity analysis, it has been shown in the past [33] that such strategies can further improve performance.

3.3 RQ3: Scalability with Thread Count

To answer RQ3, we measured the speed-ups that RA2 achieves for different thread counts compared to the nearly identical solver that uses OPAL's single-threaded, but highly optimized fixed-point computations framework[4] [28]. The IFDS client analysis is identical. The goal is to evaluate the overhead and benefits of parallelization.

Figure 3 shows how the performance changes with the number of threads used for the IFDS analysis. Depending on the scheduling strategy, the speed-up with two threads compared to one thread is between 1.6x and 2.0x. Note that better strategies in general show lower speed-ups. When increasing the number of threads further, the speed-up increases up to 6.2x for 20 threads. With speed-ups of more than 4.8x for 10 threads, according to Amdahl's law, more than 88% of the execution is parallelized. Relative standard deviations for the reported measurements are between 1.8% and 13.7%.

The RA2-based implementation of the IFDS analysis is 28% slower than the sequential implementation in OPAL when a single

[4] Version 3.0.1-SNAPSHOT

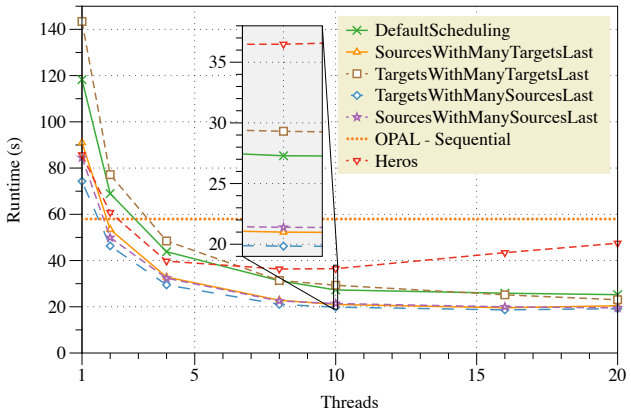

Figure 3: Performance with different numbers of threads

thread is used. The latter yielded runtimes of 58.0 seconds (RSD 8.5%), compared to 74.3 seconds for our best strategy. As expected, RA2 is slowed down by overhead related to enabling concurrency, which in this case is not needed. However RA2 clearly outperforms OPAL as soon as multiple threads are used. For the best strategy the speed-up over OPAL ranges between 1.3x with two threads, 2.9x with 10 threads, and 3.0x with 20 threads.

3.4 RQ4: Comparison to Other IFDS Solvers

To answer RQ4, we compare our IFDS analysis to several other state-of-the-art systems.

First, we compare the performance and speed-ups achieved to Heros[5] [15]. We used Heros because it is parallelized and also independent of the static analysis framework. It is very mature, widely used and freely available. As we also compared to OPAL's solver and as our analysis is based on OPAL's three address code representation, we again used this representation as the basis for our analysis in Heros. By using the exact same base technology stack for both analyses, we ensure that we compare the raw performance of the solvers and that the results are not skewed by other technical differences. The IFDS client analysis was adapted to Heros' interfaces, but performs the same analysis. Heros took 85.7 seconds on a single thread, 28% less than our `DefaultScheduling`, but already 15% more than our best strategy. For all thread counts measured, Heros had lower speed-ups than our best strategy (which is also the one with the lowest speed-ups), with a maximum of 2.36x at 8 threads. In comparison, our best strategy had a speed-up of 3.53 at 8 threads. Using more than 8 threads, Heros' performance decreased significantly, while RA2's performance increased until 16 threads (with a speed-up of 3.98x for the best strategy) and did not decrease significantly for 20 threads. Relative standard deviations for Heros were between 2.1% and 5.7%.

A direct empirical comparison with a parallelized implementation of IFDS using Actors [33] is unfortunately not possible. The solution was never publicly available and—according to the authors whom we contacted—is now practically impossible to get working again due to dependencies on unavailable and outdated beta versions of libraries. However, they have also benchmarked their solution against a sequential implementation and we compare their speed-ups with our speed-ups against the sequential implementation using OPAL. The authors of IFDS-A reported a speed-up of 3.35x on 16 threads on eight cores compared to their own sequential implementation. Our implementation, on the other hand, using again 16 threads (on 10 cores), achieves a speed-up of 3.11x over OPAL's highly optimized sequential implementation. Therefore, the performance of our implementation seems to be comparable to theirs, while our programming model is analysis-independent and, therefore, not specialized to IFDS. Among other things, the fact that two different baselines are used makes obvious that this comparison must be considered with caution and only as a work around the fact that a real comparison is not possible.

Finally, we evaluate against WALA 1.5.2 [39] as it provides another mature single-threaded IFDS implementation. As with Heros, we adapted the IFDS client analysis, this time with more effort because WALA's IFDS solver is not framework independent. The analysis was, however, thoroughly checked to be equivalent to the one used with RA2 and OPAL. To overcome framework differences related to the underlying call graphs used by the different frameworks [31], we generated and serialized WALA's RTA call graph using Judge [30] and deserialized it with OPAL. This ensures that the analyzed state space is equal. As WALA timed out after 10 hours when, analyzing the JDK, we performed a comparison with WALA on JavaCC 5.0. For this setup, WALA took 12.7 seconds (RSD 3.1%), while RA2, using the `DefaultScheduling` strategy, took 4.2 seconds on one thread (RSD 15.1%) and gave a speed-up of 4.5x (0.9 seconds, RSD 7.1%) using 16 threads. `TargetsWithManySourcesLast`, RA2's best strategy, took 0.70 seconds on 16 threads (RSD 4.9%).

Concluding remarks. Overall, the experiments reported in this subsection clearly indicate that RA2 outperforms state-of-the-art parallel IFDS solvers. Compared with the state-of-the-art IFDS solver Heros, our implementation is faster on one thread, achieves higher speed-ups and scales to more threads. It also outperforms WALA's IFDS solver significantly and seems to provide at least similar speed-ups as a specialized actor-based IFDS solver, despite being semi-implicit and analysis-independent.

4 THREATS TO VALIDITY

A threat to the validity of our claim about RA2 being able to provide semi-implicit parallelization of static analyses independent of the analysis, could be the evaluation of only two specific analyses, namely a purity analysis and an IFDS solver. We have tried to mitigate this threat by choosing two analyses as case studies that are fundamentally different in several ways. First, in their complexity, with the purity analysis being very simple and the IFDS solver being significantly more complex, but also in their use of the features provided by RA2's programming model: while the purity analysis uses a singleton value lattice and the `AggregationUpdater` to perform joins automatically, the IFDS solver uses set-based properties and the `MonotonicUpdater` as joins are performed implicitly by the analysis. Additionally, the IFDS solver makes use of mutable state shared between the continuation invocations while the purity analysis does not. The evaluation also showed that these analyses benefit from pluggable scheduling strategies in different ways. Based on

[5] Commit id: 46dda652

this, we claim that our two studies span a wide range of analysis kinds.

A threat to the validity of our evaluation results is that a direct comparison was only possible to the parallel IFDS solver Heros, but not to the related IFDS-A solver, as its implementation is not available. We, however, compared our speed-ups against an equivalent sequential analysis in OPAL to those reported by the authors of IFDS-A. We also evaluated our analysis against another sequential IFDS solver from WALA, while another related IFDS solver from Flix is not working anymore and thus could not be compared.

5 RELATED WORK

Parallel Static Analyses. There exist several previous efforts to parallelize the solution of static analysis problems.

Heros [3, 15] is a parallel, state-of-the-art IFDS solver [32]; it is one of the benchmark implementations in our experimental evaluation (see Section 3). Later approaches, e.g., Boomerang [38] that built upon Heros, were not parallelized at all, however.

Méndez-Lojo et al. [26] parallelized a points-to analysis algorithm using the Galois system [17, 18] - a programming system for thread-safe parallel iteration over unordered sets. Like RA2, their approach relies on an underlying programming framework to provide thread-safety out-of-the-box to the analyses. Unlike the programming model underlying RA2, Galois is, however, a generic framework for concurrent programming over unordered sets and it is not specifically tailored to static analysis. For example, it does not provide support to automatically find fixed-points. As a result, the approach by Méndez-Lojo et al. is not directly applicable for the parallel execution of static analyses like RA2.

Rodriguez and Lhotak present IFDS-A [33], an algorithm for solving IFDS dataflow analysis problems using the actor model [1, 16] of concurrency. In order to apply IFDS-specific scheduling strategies, the authors were required to completely exchange the Scala Actors scheduler [12] with their own implementation. Combined with their custom strategy, this was necessary for significant performance improvements. The authors reported that IFDS-A outperforms their own equivalent sequential IFDS reference implementation with 4 or more cores with a speed-up of 3.35x using 16 threads. In contrast to IFDS-A, our approach is not limited to parallelizing IFDS, e.g., Section 3 evaluates a parallel purity analysis *not* based on IFDS. While being more general and using a scheduling strategy not specific to IFDS, our approach achieves a speed-up of up to 3.11x over an equivalent, optimized sequential implementation using 16 threads. At the same time, our approach outperforms sequential IFDS using 2 cores only (by 1.3x) instead of 4 cores as in the case of IFDS-A. Finally, our pluggable scheduling strategies enable significant performance improvements (see Section 3).

In the area of points-to analyses, Datalog has been used extensively as the underlying programming framework [4, 9, 40, 41]. Analyses are specified in terms of Datalog rules and executed using Datalog solvers. The use of parallel solvers enables automatic parallelization. Soufflé [37] is a parallel Datalog solver specifically developed for static analyses. It takes the analysis specification as an input and compiles it to a C++ program. Soufflé makes use of OpenMP, e.g., to parallelize nested join loops. Using 4 cores, a speed up of 2.1x compared to the sequential version was achieved. Similar speed-ups of over 2x have been reported for the Doop static analysis framework when using Soufflé as its underlying solver, but using 4 to 8 threads on 24 cores [2]. In contrast to RA2, these approaches only work on set-based lattices.

Flix [21] overcomes this issue–it is a declarative, rule-based language inspired by Datalog, capable of solving arbitrary fixed-point computations on lattices. Even though Flix is amenable to automatic parallelization due to its declarative nature, no parallel solver has been proposed for it yet. At some point in the past, Flix also provided a proof-of-concept implementation of IFDS, however, this is not working with the current version of Flix anymore.

Reactive Frameworks for Static Analyses. Reactive programming provides abstractions for event streams and time changing values (signals) [7, 22, 24, 27, 35, 36], which are well-suited for smart dependency management for static analyses. However, general-purpose reactive programming approaches cited above organize computations in an acyclic graph – by being general-purpose, they have no means to resolve cycles out-of-the box and hence the requirement that the graph is acyclic. However, cyclic data dependencies are essential for the target domain of static analysis. To address this need, the reactive programming framework underlying RA2 is more specialized. It is a reactive framework for time efficient, concurrent, fixed-point computation on lattices, which enables it to resolve cycles in a general way. We build on our previous work, Reactive Async [10], a programming system for deterministic concurrency in Scala that extends lattice-based shared variables, LVars [19], with cyclic data dependencies, which are resolved after the computation has reached a quiescent state and shared variables are no longer updated. RA2 significantly extends upon Reactive Async. First, RA2 allows shared variables (cells) to use a sequential update strategy, enabling continuations to safely access shared mutable state by executing them sequentially. Supporting shared mutable state is essential for implementing a state-of-the-art IFDS solver which is the basis for our experimental evaluation (see Section 3). Second, RA2 allows custom cell updaters such that monotonicity violations are detected dynamically while expensive additional joins can be omitted. Third, RA2 supports pluggable scheduling strategies which, as shown in Section 3, have a significant performance impact and allow analysis-specific tuning of the parallelization. Finally, aggregation of updates, both for a single dependee and across multiple dependees, reduces the number of continuation invocations for further performance improvements.

6 SUMMARY AND FUTURE WORK

In this paper, we proposed a new programming model for parallelizing static analyses based on the ideas and concepts of reactive programming, that (I) is semi-implicit, requiring only minor adaptions to analyses to benefit from parallelization, (II) is analysis-independent, lending itself very well towards the implementation of a wide range of static analyses, including purity and data-flow analyses. On top of our framework, which implements the proposed model, we implemented an IFDS solver and used the latter to implement a classical taint-flow analysis. The evaluation shows that our approach, while analysis-independent, is able to significantly outperform the highly-specialized, explicitly parallelized IFDS solver Heros, achieving both better performance and scalability.

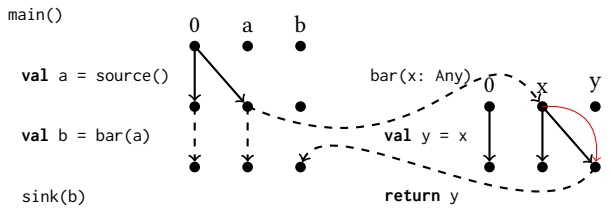

Figure 4: Simple IFDS taint analysis

In future work, we plan to investigate whether on-the-fly profiling could be used by Reactive Async to automatically find the best scheduling strategy. Furthermore, we will investigate how to integrate speculative parallelization techniques to further increase the overall degree of parallelization. The presented programming model requires explicit continuations for processing updates of analysis results. By building on previous work on direct-style concurrency in Scala [11, 34], we plan to remove these explicit continuations, thereby making the programming model easier to use. Moreover, we would like to conduct user studies in order to evaluate the effect of our semi-implicit parallelization approach on program comprehension. Finally, our experimental evaluation suggests that the development of a standard benchmark suite for parallel program analyses will be needed in the future to ensure faithful comparisons between a growing number of approaches and frameworks.

ACKNOWLEDGMENTS

This work was supported by the DFG as part of CRC 1119 CROSSING, by the German Federal Ministry of Education and Research (BMBF) as well as by the Hessen State Ministry for Higher Education, Research and the Arts (HMWK) within their joint support of the National Research Center for Applied Cybersecurity ATHENE.

APPENDIX

This appendix provides a short introduction to the IFDS analysis framework [32] for interprocedural, finite, distributive subset problems. IFDS is a dataflow analysis framework based on graph reachability that provides context- and flow-sensitivity. IFDS analyses are given by a finite domain of boolean facts that may or may not hold at a specific statement of the program. Four so-called flow-functions then need to be defined by users to define the effects of statements on the dataflow facts.

As shown in Figure 4, which depicts a simple taint analysis, the dataflow facts (here, whether a local variable is tainted) at each statement are represented by nodes. The edges are given by the flow-functions which describe the effects of non-call statements (normal flow, regular edges) and of call statements (call flow mapping facts from the caller's scope to the callee's scope, dashed edge from `main` to the beginning of `bar`; return flow mapping facts back to the caller's scope, dashed edge from the end of `bar` back to `main`; call-to-return flow for facts unaffected by the call, dashed edges inside `main`). The resulting graph is known as the exploded supergraph of the program as it is an expansion of the program's interprocedural control-flow graph. The special fact 0 represents the tautological fact that holds everywhere. A fact is considered to hold at a program statement if it is reachable from the 0 fact. IFDS is efficient by reusing parts of the graph computed for one method by computing path edges (e.g., thin red edge in Figure 4) which summarize for a function whether a node is reachable given the reachability of a node at the method's start.

REFERENCES

[1] Gul A. Agha. 1990. *ACTORS - A Model of Concurrent Computation in Distributed Systems.* MIT Press.
[2] Tony Antoniadis, Konstantinos Triantafyllou, and Yannis Smaragdakis. 2017. Porting Doop to Soufflé: a Tale of Inter-Engine Portability for Datalog-Based Analyses. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis.* ACM, 25–30.
[3] Eric Bodden. 2012. Inter-procedural Data-flow Analysis with IFDS/IDE and Soot. In *SOAP.* 3–8.
[4] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'09).* 243–262.
[5] DoopBenchmarks [n.d.]. Doop Benchmarks. https://bitbucket.org/yanniss/doop-benchmarks.
[6] M. Eichberg, F. Kübler, D. Helm, M. Reif, G. Salvaneschi, and M. Mezini. 2018. Lattice Based Modularization of Static Analyses. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops (ISSTA'18).* ACM, 113–118.
[7] Conal Elliott and Paul Hudak. 1997. Functional Reactive Animation. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming (ICFP '97).* ACM, New York, NY, USA, 263–273.
[8] David Grove and Craig Chambers. 2001. A Framework for Call Graph Construction Algorithms. *TOPLAS* 23, 6 (Nov. 2001), 685–746.
[9] Elnar Hajiyev, Mathieu Verbaere, and Oege De Moor. 2006. Codequest: Scalable Source Code Queries with Datalog. In *European Conference on Object-Oriented Programming (ECOOP'06).* Springer, 2–27.
[10] Philipp Haller, Simon Geries, Michael Eichberg, and Guido Salvaneschi. 2016. Reactive Async: Expressive Deterministic Concurrency. In *SCALA@SPLASH.* ACM, 11–20.
[11] Philipp Haller and Heather Miller. 2019. A Reduction Semantics for Direct-Style Asynchronous Observables. *J. Log. Algebr. Meth. Program.* 105 (2019), 75–111.
[12] Philipp Haller and Martin Odersky. 2009. Scala Actors: Unifying Thread-Based and Event-Based Programming. *Theoretical Computer Science* 410, 2-3 (2009), 202–220.
[13] Tim Harris and Satnam Singh. 2007. Feedback Directed Implicit Parallelism. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP'07).* Association for Computing Machinery, New York, NY, USA, 251–264.
[14] Dominik Helm, Florian Kübler, Michael Eichberg, Michael Reif, and Mira Mezini. 2018. A Unified Lattice Model and Framework for Purity Analyses. In *ASE.* ACM, 340–350.
[15] Heros [n.d.]. Heros IFDS/IDE Solver. https://github.com/Sable/heros.
[16] Carl Hewitt, Peter Boehler Bishop, and Richard Steiger. 1973. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence (IJCAI). Standford, CA, USA, August 20-23, 1973.* William Kaufmann, 235–245.
[17] Milind Kulkarni, Martin Burtscher, Rajasekhar Inkulu, Keshav Pingali, and Calin Cascaval. 2009. How much Parallelism is there in Irregular Applications?. In *PPoPP.* ACM, 3–14.
[18] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. 2007. Optimistic Parallelism Requires Abstractions. In *PLDI.* ACM, 211–222.
[19] Lindsey Kuper, Aaron Turon, Neelakantan R. Krishnaswami, and Ryan R. Newton. 2014. Freeze After Writing: Quasi-Deterministic Parallel Programming with LVars. In *POPL.* ACM, 257–270.
[20] Johannes Lerch, Ben Hermann, Eric Bodden, and Mira Mezini. 2014. FlowTwist: Efficient Context-Sensitive Inside-Out Taint Analysis for Large Codebases. In *SIGSOFT FSE.* ACM, 98–108.
[21] Magnus Madsen, Ming-Ho Yee, and Ondrej Lhoták. 2016. From Datalog to Flix: A Declarative Language for Fixed Points on Lattices. In *PLDI.* ACM, 194–208.
[22] A. Margara and G. Salvaneschi. 2018. On the Semantics of Distributed Reactive Programming: The Cost of Consistency. *IEEE Transactions on Software Engineering* 44, 7 (2018), 689–711.
[23] Simon Marlow, Simon Peyton Jones, and Satnam Singh. 2009. Runtime Support for Multicore Haskell. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP'09).* Association for Computing Machinery, New York, NY, USA, 65–78.

[24] Erik Meijer. 2010. Reactive Extensions (Rx): Curing Your Asynchronous Programming Blues. In *ACM SIGPLAN Commercial Users of Functional Programming (CUFP '10)*. ACM, New York, NY, USA, Article 11, 1 pages.
[25] Mario Méndez-Lojo, Martin Burtscher, and Keshav Pingali. 2012. A GPU Implementation of Inclusion-Based Points-to Analysis. In *PPoPP*. ACM, 107–116.
[26] Mario Méndez-Lojo, Augustine Mathew, and Keshav Pingali. 2010. Parallel Inclusion-Based Points-to Analysis. In *OOPSLA*. ACM, 428–443.
[27] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. 2009. Flapjax: A Programming Language for Ajax Applications. In *Proceeding of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA '09)*. ACM, New York, NY, USA, 1–20.
[28] Opal [n.d.]. OPAL. https://github.com/stg-tud/opal.
[29] Tarun Prabhu, Shreyas Ramalingam, Matthew Might, and Mary W. Hall. 2011. EigenCFA: Accelerating Flow Analysis with GPUs. In *POPL*. ACM, 511–522.
[30] Michael Reif, Florian Kübler, Michael Eichberg, Dominik Helm, and Mira Mezini. 2019. Judge: Identifying, Understanding, and Evaluating Sources of Unsoundness in Call Graphs. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 251–261.
[31] Michael Reif, Florian Kübler, Michael Eichberg, and Mira Mezini. 2018. Systematic Evaluation of the Unsoundness of Call Graph Construction Algorithms for Java. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*. ACM, 107–112.
[32] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. 1995. Precise Interprocedural Dataflow Analysis via Graph Reachability. In *POPL*. ACM, 49–61.
[33] Jonathan Rodriguez and Ondrej Lhoták. 2011. Actor-Based Parallel Dataflow Analysis. In *CC*. Springer, 179–197.
[34] Tiark Rompf, Ingo Maier, and Martin Odersky. 2009. Implementing First-Class Polymorphic Delimited Continuations by a Type-Directed Selective CPS-Transform. In *Proceeding of the 14th ACM SIGPLAN International Conference on Functional programming (ICFP'09)*. ACM, 317–328.
[35] Guido Salvaneschi, Gerold Hintz, and Mira Mezini. 2014. REScala: Bridging Between Object-oriented and Functional Style in Reactive Applications. In *Proceedings of the 13th International Conference on Modularity (MODULARITY'14)*. ACM, New York, NY, USA, 25–36.
[36] Guido Salvaneschi and Mira Mezini. 2016. Debugging for Reactive Programming. In *Proceedings of the 38th International Conference on Software Engineering (ICSE'16)*. Association for Computing Machinery, New York, NY, USA, 796–807.
[37] Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. 2016. On Fast Large-Scale Program Analysis in Datalog. In *Proceedings of the 25th International Conference on Compiler Construction*. ACM, 196–206.
[38] Johannes Späth, Lisa Nguyen Quang Do, Karim Ali, and Eric Bodden. 2016. Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java. In *ECOOP (LIPIcs)*, Vol. 56. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 22:1–22:26.
[39] Wala [n.d.]. Wala. http://wala.sourceforge.net.
[40] John Whaley, Dzintars Avots, Michael Carbin, and Monica S Lam. 2005. Using Datalog with Binary Decision Diagrams for Program Analysis. In *Asian Symposium on Programming Languages and Systems*. Springer, 97–118.
[41] John Whaley and Monica S. Lam. 2004. Cloning-Based Context-Sensitive Pointer Alias Analysis Using Binary Decision Diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI'04)*. Association for Computing Machinery, New York, NY, USA, 131–144.