BlaSt: A Generic Framework for Collaborative Static Analyses

Anonymous Author(s)

ABSTRACT

Current approaches combining multiple static analyses deriving different, independent properties focus either on modularity or performance. Whereas declarative approaches facilitate modularity and automated, analysis-independent optimizations, imperative approaches foster manual, analysis-specific optimizations.

In this paper, we present BlaSt, a static analysis framework that leverages the modularity of blackboard systems and combines declarative and imperative techniques. Our framework allows exchangeability, and pluggable extension of analyses in order to improve sound(i)ness, precision, and scalability and explicitly enables the combination of otherwise incompatible analyses. Using BlaSt, we were able to implement various dissimilar analyses, including a points-to analysis that outperforms an equivalent analysis from Doop, the state-of-the-art points-to analysis framework.

ACM Reference Format:
Anonymous Author(s). 2020. BlaSt: A Generic Framework for Collaborative Static Analyses. In *Proceedings of The 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE 2020).* ACM, New York, NY, USA, 12 pages.

1 INTRODUCTION

Complex static analyses often involve solving several related but distinct sub-problems to achieve an overall goal. As an example consider a method's purity analysis that involves calculating the mutability of classes and fields [39, 60] and may also use escape information [15, 47]. Encoding solutions for such a complex static analysis' sub-problems in separate modules fosters their reusability, maintainability, and extensibility. Yet, the execution of modules for inter-dependent sub-problems needs to be interleaved to enable exchanging intermediate results. The latter is often necessary for optimal precision, as it has been proven by the theory of reduced products in abstract interpretation [19] and was more recently demonstrated for other kinds of analyses [11, 28, 38].

Traditional imperative static analyses have been implemented in a monolithic manner, i.e., one super-analysis also computes the results of related sub-analysis problems. Such design, where a super-analysis coordinates the communication of sub-analyses, is, however, very complex for mutually dependent problems [12]. Individual sub-analyses cannot be developed in isolation, exchanged for different implementations, or reused for other monolithic analyses.

More recently, declarative approaches to static analyses [12, 52, 72] are gaining increased popularity—especially in the area of points-to analyses [12, 67, 69, 72], using the Datalog language. With such approaches, analyses are implemented as sets of rules that are evaluated by an underlying constraint solver. Thus, as demanded, complex analyses can be broken down into modular independently-developed simpler analyses. The underlying solver transparently resolves their dependencies, thereby propagating intermediate updates according to the rules specified by the analyses.

ESEC/FSE 2020, 8 - 13 November, 2020, Sacramento, California, United States
2020.

It can apply analysis-independent optimizations, e.g., by rearranging the computation order (although manual optimization is still required [12, 66]), and/or can automatically parallelize the execution [43]. However, using Datalog and giving solvers full control comes with *drawbacks in terms of performance and generality.*

First, it is not possible to take analysis-specific knowledge into account in the way the execution and dependencies of the analyses are managed. Such knowledge can help boosting scalability. For example, an imperative purity analysis that determines whether a method is deterministic by, among others, checking the mutability of fields $f_1, ..., f_n$ could drop further checks as soon as any f_i is found to be mutable. A declarative analysis whose execution is driven by a general-purpose solver cannot take this short-cut. Second, the solver uses analysis-independent data structures and analyses cannot exploit data structures that are optimized for their specific needs. Such optimized data structures, like tries, can be crucial for achieving performance. Last but not least, the one-size-fits-all approach of declarative approaches may also limit the framework's generality. For instance, by relying on relations, Datalog-based approaches support only set-based lattices. Yet, many common analyses require other kinds of lattices, e.g., constant propagation is usually implemented via singleton-value-based lattices, making it infeasible to implement it using Datalog [52, 68].

To address these issues, we propose BlaSt, a novel generic framework together with a proof-of-concept implementation for lattice-based fixed-point computations with support for lattices of any kind including singleton-value-based, interval, and set lattices. Like fully declarative approaches, BlaSt features modular analyses that are encoded as independently compilable, exchangeable, and extensible units. However, BlaSt does not rely on a general-purpose declarative framework and its underlying constraint solver. It offers a specialized approach mixing imperative and declarative styles.

The developer of a BlaSt analysis implements its core functionality in an imperative style, but declaratively specifies its dependencies, e.g., the lattice that the analysis computes and lattices it depends on to do so, as well as several constraints regarding its execution. Dependencies and constraints are automatically handled by BlaSt's solver during the execution of analyses. BlaSt's architecture is reminiscent of blackboard systems [17]: Dependent analyses that are implemented in decoupled modules coordinate their executions implicitly by writing into and reading from a central data structure (the "blackboard"), whereby these operations are automatically (and concurrently) scheduled by the solver to satisfy the declaratively specified dependencies and constraints. Whenever an analysis has (intermediate) results to share, it writes those onto the blackboard, which notifies all dependent analyses.

Like declarative approaches, BlaSt decouples mutually dependent analyses, enabling their isolated development and their interleaved parallel execution out-of-the-box. By doing so, it supports adding, removing, and exchanging analyses to trade-off between precision, sound(i)ness, and performance in a fine-tuned way. BlaSt improves over declarative approaches in two regards.

First, beyond automatic and transparent optimizations and parallelization, BLAST's imperative programming style enables analysis-specific optimizations and use of optimized data structures. The possibility to specify fine-grained (analysis-specific) constraints enables further optimizations, e.g., suppressing interleaved execution of some analyses to avoid unnecessary intermediate computations.

Second, BLAST is generic and supports arbitrary kinds of analyses. Existing Datalog-based approaches are limited to set lattices; in BLAST, one can naturally express dataflow and constraint-based static analyses based on arbitrary lattices. A key enabler for generality is the underlying custom solver that is agnostic of the lattices. The solver checks monotonicity of updates as specified by the respective lattice. To ensure termination, we require lattices to satisfy the so-called ascending/descending chain condition. Moreover, declarative dependency declarations enable BLAST to take analysis-specific constraints into consideration when managing dependencies. This enables it to correctly compose incompatible optimistic and pessimistic ones (as defined in [34, 50])—BLAST is the first to explicitly enable lazy computation of results in this case.

To recap, this paper contributes:

- A list requirements on frameworks for collaborative static analysis is distilled from three case studies (Section 3).
- A novel approach, BLAST, satisfying all these requirements to improve on the state-of-the-art for implementing modular dependent analyses to support pluggable soundness/precision/performance while enabling optimizations (Section 4).
- A thorough evaluation of BLAST that supports our claims on generality, showcases BLAST's modularity features, points out performance improvements over Doop, and provides promising results for parallelization (Section 5).

2 BACKGROUND AND TERMINOLOGY

In this section, we give a short introduction into blackboard systems and present the terminology that we use throughout the paper.

2.1 Blackboard Systems

A blackboard system [17] uses a central data structure, called the blackboard, to enable decoupled knowledge sources to jointly contribute to an overall goal. Knowledge sources contribute partial information to the blackboard that can then be queried by other knowledge sources to produce further information. The blackboard is responsible for notifying knowledge sources about new information they might require. This is done through a control mechanism that decides which knowledge sources should be activated in what order. Each execution of a knowledge source is called an *activation*.

2.2 Terminology

Entity. An entity is anything one can compute some information for. Entities can be concrete code elements such as classes, methods, fields, or allocation sites but also abstract concepts such as a project. The set of entities can be created on-the-fly while analyses execute.

Property Kind. A property kind is a specific kind of information that can be computed for an entity. Property kinds include, e.g., mutability of classes, purity of methods, or callees of a specific method. Each property kind represents one lattice of possible results.

Property. A property is a specific value, taken from the lattice of some property kind, that is attached to some entity. For example, classes can be mutable or immutable, methods can be pure or impure, and a specific method may invoke a specific set of methods. Per entity at most one property of a specific kind can be computed.

Analysis Given an entity, an analysis computes the entity's property of a property kind. We say that *an analysis computes a property kind* as a shorthand for an analysis that computes properties of that property kind for a given entity kind. Analyses are knowledge sources in the sense of the blackboard architecture; the program properties that they compute constitute the information that they either contribute to or query from the blackboard.

3 CASE STUDIES

We discuss case studies involving several interrelated sub-analyses[1], *emphasize* concepts whenever they occur during the discussion, and distill a list of requirements on static analysis frameworks. The case studies represent very dissimilar kinds of analyses. In particular different kinds of lattices are required, including singleton-value lattices (e.g. in 3.3) and set-based lattices (e.g. in 3.2). This motivates the first requirement: Static analyses frameworks need to support arbitrary kinds of analyses with varied domain lattices (**R1**).

3.1 Three-Address Code

Case study one presents an analysis that produces a three-address code representation (TAC) of bytecode. In its basic version, TAC uses def/use, type, and value information (including constant propagation) provided by abstract interpretation (AI). TAC may be enhanced with analyses that refine the return type and the value information for methods and fields to increase precision. However, such additional analyses may negatively affect the runtime. Hence, a systematically investigation of the precision/performance trade-off is needed. Here, a separation into modules helps, as they can be enabled/disabled. In general, we derive the following two requirements related to supporting modular pluggable analyses.

Static analysis frameworks should supply support for easily en/disabling analyses that rely on each other's results for the systematic study of precision/soundness/performance trade-offs (**R2**). To maximize pluggability, analyses should be defined in utterly decoupled modules, while still being able to collaboratively compute properties. This enables, what we call *collaborative analyses*.

Since individual analyses can be disabled it should be possible to provide soundly over-approximated *fallback values*[2] for the properties that they compute, which can be used by dependent analyses to compensate missing results (**R3**).

Moreover, an approach for modular collaborative analyses should support their *interleaved execution*. Two analyses are executed interleaved, if they can interchange *intermediate results* without knowing of each other's existence(**R4**). This is important for optimal precision [19]: knowledge gained during the execution of some analysis a_1 may be used by the execution of some other analysis a_2 on-the-fly to refine its result and, in turn, this may enable further

[1]A graphical depiction of the dependencies between the analyses presented here can be found in the supplementary material.

[2]To minimize the effect of fallback values on precision, it makes sense to compute the fallback by using locally available information, e.g., using declared type information, instead of always returning the same over-approximated value.

refinement for a_1. The field- and return-value refinement analyses are examples of analyses, whose precision would profit from interleaved executions. They depend on each other cyclically. If a method `m` returns the value of a field `f`, then the return value of `m` depends on `f`'s value. Similarly, if the value returned by `m` is written into `f`, then `f`'s value also depends on `m`'s return value.

Interleaved execution is not, however, always appropriate. There are cases when it must be suppressed for correctness and other cases, when it may be suppressed for performance.

An example of the first case concerns the composition of *pessimistic* and *optimistic* analyses. Pessimistic analyses start with a sound but potentially imprecise assumption and eventually refine it. Optimistic analyses start with an unsound but (over)precise assumption and progress by reducing (over-)precision towards a sound result. Field and return value refinement analyses are pessimistic—the declared return type of method *m*, say `List`, is a sound but eventually imprecise initial value for the return-value analysis; during the execution, the analysis may find out that *m* actually returns the more precise result, say `ArrayList`. AI is an optimistic analysis—it starts with the unsound assumption that all code is dead and refines it by adding statements found to be alive towards a sound, but potentially less precise result.

Optimistic and pessimistic analyses are *incompatible* for interleaved execution. They cannot exchange intermediate results since these can cause inconsistencies, as they refine their lattices in opposite directions, thereby, violating monotonicity. Thus,the analysis framework must enforce that only *final results* of pessimistic analyses are passed to a dependent optimistic analysis (and vice-versa), avoiding interleaving and,therefore, *suppressing* non-final updates.

To illustrate suppression for performance, consider the relation between TAC and AI. Both are optimistic and TAC could in principle use intermediate AI results. But, these results are typically not useful, hence, it can be beneficial to suppress them and instead perform the TAC transformation only once on the final AI result.

To recap, the decision about execution interleaving is highly analyses specific. Hence, we pose the requirement that a modular analysis framework should be parameterized by fine-grained knowledge about inter-analysis relationships related to interleaving to be specified by analysis developers (**R5**).

3.2 Modular Call Graph Construction

Inter-procedural analyses presume a call graph (CG): Given a method `m`, CG provides information about (a) methods that may be invoked at a call site in `m` (callees) and (b) call sites from which `m` may be invoked (callers). We use the CG case to motivate the need for supporting different kinds of interleaving executions (beyond the one covered by requirement R3) as well as some further requirements.

The previous case study (composition of analyses for refining field and return values with TAC) illustrated the need for interleaved execution of analyses with circular dependencies calculating different properties and operating on different entities. The CG use case illustrates two further kinds of interleaved execution.

First, the approach needs to support interleaved execution of multiple instances of the same analysis operating on different code entities to collaboratively compute a single property, whereby each instance contributes partial results (**R6**). For example, different executions of a CG analysis for different callers of a method *m* need to contribute their *partial results* to collaboratively derive all of *m*'s callers (computing callers of a method is inherently non-local).

Second, a modular analyses framework also needs to support interleaving of independent analyses to collaboratively compute a single property (**R7**). For illustration, consider the computation of the callees of a method `m`. A CG analysis can in principle consider *m* in isolation. A monolithic analysis for callees is nonetheless not suitable. It makes sense to distinguish between a sub-analysis (like CHA [23], RTA [4], points-to-based [12] analysis) that handles standard invocation instructions and analyses dedicated to non-standard invocation instructions encoding specific language features, e.g., reflection, native methods, or functionality related to threads, serialization, etc. Non-standard invocation instructions require specific handling (e.g., one may deliberately not want to perform reflection resolution, or may want to perform it based on dynamic execution traces). By offering such specialized analyses as decoupled modules, they become highly reusable be combined with different call-graph analysis for standard invocation instructions. This makes the call graph construction highly configurable, which allows for fine-tuning its performance and sound(i)ness.

Hence, not only a method's callers but also its callees need to be computed collaboratively. This time different analyses targeting different language features, rather than different executions of the same CG analysis contribute to the same property. Handling such special features may even integrate results of external tools or precomputed values, e.g., for native methods (**R8**). For instance, one may choose to integrate the results of TamiFlex [10] for reflective calls, or external tools for analyzing native methods.

The CG case study also motivates support for specifying precise *default values* (**R9**) (besides sound fallback values). For illustration consider the case of an unreachable method *m*. The CG analysis, will never compute callees or caller information for *m*. However, this lack of results is an inherent property of the entity, as opposed e.g., being the result of disabling an analysis. An over-approximating fallback value to compensate the deactivation of the CG module for *m* may have to include all methods, which would be too imprecise. Instead, analyses depending on the CG should get the information that *m* is unreachable—the precise default value, which the developer of the analysis knows about and can tell the framework.

Another requirement is motivated by the CG case. The CG construction unfolds along the transitive closure of methods reachable from some entry points. Hence, it does not make sense to execute the decoupled modules collaboratively constructing the CG—each handling a particular language feature—globally on all methods of a program. Instead, they should be *triggered* only when the overall analysis progress discovers a newly reachable method. Hence, the framework must support triggering analyses once the first (intermediate) result for a property is recorded (**R10**).

3.3 Mutability, Escape, and Purity Analysis

The analyses in this case study interact tightly and compute properties that may be relevant for both end users and further analyses.

We present analyses for method purity, class and field mutability [39, 60] and escape information [15, 47]. The latter includes aggregated information on field locality and return-value freshness

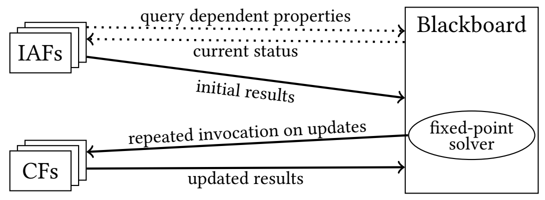

Figure 1: Overview

(cf. [38]). Complex dependencies exists between all these analyses. To fine-tune the precision/performance trade-off, several analyses for these property kinds with different precision can be exchanged as needed; all are *optimistic* and use TAC and/or the CG information.

The analyses in this subsection illustrate the need to support further kinds of activation modes in addition to triggered analyses, illustrated in the previous subsection. This includes (a) *eager analyses* (**R11**), which refers to computing an analysis for all entities in the analyzed program, and (b) *lazy analyses* (**R12**), i.e., executing an analysis a_1 only for the entities, for which the property that a_1 computes is queried by some (potentially the same) analysis a_2.

Since the results of analyses in this case study may be of interest to the end user, it is useful to compute them for all possible entities eagerly. e.g., computing the mutability of all fields in the analyzed program. However, when the field mutability is only used to support, e.g., the purity analysis, it may be beneficial for the overall execution time to compute it lazily, i.e., only for the fields for which the mutability is queried by the purity analysis. The case study illustrates that a framework for modular analyses should support both eager and lazy execution modes. Eager and lazy versions of an analysis should typically share the analysis code and only be registered with the framework in different ways.

The class mutability analysis motivates another requirement: Instead of relying on the solver to select an order in which different classes are analyzed, it can be beneficial for performance to have the analysis enforce a specific order, in this case, top-down; namely, that parent classes are analyzed before the analysis of their child classes is started. Thus, the framework should allow analyses to specify which computations to perform next (**R13**).

3.4 Summary of Requirements

Table 1 gives a summary of all requirements yielded by the case studies along with the case studies motivating them.

Existing frameworks do not fulfill all of the above requirements. Imperative frameworks lack support for modularity, especially **R5**, **R6**, and **R7**. Declarative approaches such as Doop [12] have other limitations. First, by being bound to relations for modeling properties, they fail to address **R1**, which restricts their ability to express the range of different analyses represented by our case studies. Second, declarative approaches fail to support sound interactions between incompatible analyses (**R5**). Finally, by giving the solver full control of the execution, they also do not support different kinds of analysis-specific activation modes of analyses (**R11**-**R13**).

Table 1: Summary of Requirements

Lattices and values	
R3	Fallbacks of properties when no analysis is scheduled (3.1, 3.3)
R9	Default values for entities not reached by an analysis (3.2)
R1	Support for different kinds of lattices (3.1, 3.2, 3.3)
Composability	
R2	Support for enabling/disabling individual analyses (3.1, 3.2, 3.3)
R4	Interleaved execution with circular dependencies (3.1, 3.2, 3.3)
R5	Combination of optimistic and pessimistic analyses (3.1)
R6	Different activations contributing to a single property (3.2)
R7	Independent analyses contributing to a single property (3.2)
Initiation of property computations	
R8	Precomputed property values (3.2, 3.3)
R11	Start computation eagerly for a predefined set of entities (3.3)
R12	Start computation lazily for entities requested (3.1, 3.3)
R10	Start computation once an analysis reaches an entity (3.2)
R13	Start computation as guided by an analysis (3.3)

4 APPROACH

BlaSt is the first static analysis approach to build upon the concept of blackboard systems. In this section, we will illustrate how our approach achieves modular decomposition of analyses, broad applicability, and automated parallelization.

Its static analyses are the knowledge sources and the blackboard is a store of properties computed by them. In doing so, we combine benefits from declarative frameworks with the freedom and optimizations from imperative analyses.

As already mentioned in the introduction, BlaSt provides the best of both the imperative and declarative analysis styles. This has also influenced the design of our approach (Figure 1).

In order to use BlaSt, an analysis developer needs to implement the lattice representation (4.1) and the analysis itself (4.2). The latter is composed of two *imperative* functions, the *initial analysis function* (IAF) and the *continuation function* (CF). The initial analysis function is executed for each entity; it analyzes the program and queries the blackboard for any additionally required properties. Once it completes, it returns an (intermediate) result (4.3) to the blackboard along with any, yet unfulfilled dependencies.

This passes control to the blackboard, that, similar to the solver of *declarative* frameworks, will resolve the dependencies and invoke the continuation functions to process any relevant update. In Figure 1, the declarative control passes are represented by straight lines. Guided by a declarative specification of dependencies (4.4), the blackboard is responsible for ensuring execution constraints (4.5) and that a fixed-point is eventually reached (4.6). The blackboard can automatically provide scheduling and parallelization to all analyses (4.7).

4.1 Representing Properties

Different analyses must be able to both find and interpret the data that is stored on the blackboard. In BlaSt, developers explicitly implement *property kinds* in code[3], thereby reifying them. Reification gives property kinds both an identifier which can be used to query

[3] A concrete example can be found in the supplementary material.

the blackboard and an interface which can be used to work with the data. While a property kind's implementation fixes the structure of each property kind and the way its values must be interpreted. Analyses computing them can be implemented in any suitable way as long as they provide their results in the specified structure.

A property kind comprises values from some lattice, including singleton value-based lattices, interval lattices, or set-based lattices (**R1**). When defining a property kind, developers can choose the most suitable data structures to allow for efficient computation. We use the lattice's bottom value to model the best possible value (e.g., pure for method purity) while the lattice's top value is used to represent the sound over-approximation (e.g., impure). The lattices must fulfill the ascending (descending) chain condition to ensure termination of optimistic (pessimistic) analyses.

When implementing property kinds, developers can specify *fallback* and *default* values. The blackboard will provide fallback values if no analysis computing the requested property kind is available (**R3**). The lattice's top value as a sound over-approximation is typically a good choice. However, the fallback value can also be provided by a limited analysis function that does not query the blackboard and, thus, avoids cyclic dependencies.

A *default value* on the other hand is used if there is an analysis for a property kind, but it did not produce any result for some entity (**R9**). This occurs, e.g., in call graph analyses that only examine methods reachable from some entry points and consider the other methods dead. A default value can be used to state that a method is dead and therefore has no (relevant) callees, while a sound fallback value would have to include all possible methods as callees of a method that was not analyzed. Thus, the default value gives more information in this case than a fallback value would. If no default value is given, the fallback value is used as it is always sound.

4.2 Analysis Structure

Developers of analyses in BlaSt structure them in two parts: An *initial analysis function* and one or more *continuation functions*.

The initial analysis function collects information directly from the program's code in order to compute the analysis result. When additional information is required for other entities or from other analyses, the initial analysis function queries the blackboard for these dependencies, using the specific entity and property kind to find the relevant information. The blackboard will return the currently available information. If that is not available or not final, because the respective analysis was not yet executed or its dependencies were not satisfied, the analysis has to remember these open dependencies. Once the initial analysis of the code is completed, the initial analysis function returns a result computed from the currently available information to the blackboard.

If there are open dependencies, these are reported to the blackboard. In order to process further updates, the blackboard invokes continuation functions, which are registered at the end of the initial analysis functions as well. Executions of the initial analysis function and the continuation functions are called *analysis activations.*

This analysis structure ensures that analyses do not wait when information is not yet available. Therefore, BlaSt can handle cyclic dependencies (**R4**). Additionally, this enables efficient lazy computation of dependencies: computation can be started once they are queried while the analysis continues without waiting (**R12**).

Apart from obeying this basic structure, developers may use any suitable strategy to implement an analysis. They may, e.g., not traverse all code of a method, but instead focus on specific statements (there are simple pre-analyses available that can provide, e.g., all statements that access a specific field). Analyses can also use any suitable data structures internally to achieve good performance.

There are two important semantic constraints that the implementations of the analyses have to fulfill. First, updates of results must be monotonic according to the lattice used. Analyses that optimistically start at a lattice's bottom value may only refine their approximations upwards, pessimistic analyses may only refine downwards. As the direction of possible refinement is known, monotonicity allows analyses to interpret intermediate results. Monotonicity of updates can be checked automatically by the framework.

Second, analyses must be *scheduling independent*: Whenever they report an intermediate result, this result must have been computed based on all information that the analysis has available. This is necessary to guarantee that all information has been processed once there are no further updates. For example, once the purity analysis of a method `m` knows that `m`'s callee `nd` is non-deterministic, it may no longer report that `m` could be pure. Analyses can perform new queries to the blackboard and establish more dependencies in their continuation, as long as scheduling independence is preserved.

4.3 Reporting Results

Analyses write both their intermediate and final results to the blackboard. They can report one or multiple results at the same time. These results may report a single lattice value to the blackboard. Alternatively, they may specify a function that updates the current blackboard value to incorporate the analysis' results.

The latter is used to deal with properties that are not localized to a specific part of the analyzed program.For properties like these, constraint-based analyses [1, 55] have been used in the past. Declarative analyses provide such incremental updates using deltas, too.

As the provided function merges one activation's results to the current state (e.g., adding one caller to an existing set of known callers), activations of one or several analysis can contribute to a property collaboratively (**R6**, **R7**). To ensure determinism, BlaSt executes the update functions for a single property sequentially, while the analysis activations themselves can run concurrently.

Some analyses benefit from a specific order in which properties for different entities are computed, e.g., traversing the class hierarchy. Therefore, in BlaSt, upon returning a result to the blackboard, the developer may additionally declaratively specify a number of computations to be scheduled next (**R13**).

4.4 Specifying Dependencies and Entities

Besides defining an analysis, the developer also specifies the property kinds computed by the analysis, the analysis' dependencies, and for which entities the analysis will be executed. These specifications are evaluated when the analysis is registered to the blackboard before it takes over to control the analysis' activation. Listing 1 gives an example for the lazy class mutability analysis.

```
1  override def derivesLazily = Optimistic(ClassMutability)
2  override def uses = Set(Optimistic(FieldMutability))
3  override def register() = {
4    val analysis = new ClassMutabilityAnalysis
5    Blackboard.registerLazyAnalysis(this, analysis.analyze)
6  }
```

Listing 1: Registration of Class Mutability Analysis

In Line 1 it is specified that the analysis optimistically derives class mutability. Dependency specifications (Line 3) declare (a) queried property kinds and (b) whether the analysis can process intermediate values form optimistic/pessimistic analyses or whether it can handle only final values. Specifications regarding entity selection consist of (a) information on which entities to apply the analysis to and (b) when the blackboard starts the analysis' execution for each entity. For the different cases shown next, respective registration methods need to be invoked (Line 5) on the blackboard.

Analyses can eagerly select a set of entities (e.g., all methods of the analyzed program) if all or most of that information is likely required (**R11**) during the execution. This is especially useful for analyses that are of interest to the end user, e.g., if the user is interested in the purity of all methods of the analyzed program.

Alternatively, analyses can be registered to be invoked lazily [9, 40]. A lazy analysis only computes an entity's property if it is queried (**R12**), either by another analysis or by the end user. In the example, the mutability of a certain class will only be computed if, e.g., the purity analysis queries the blackboard for this class.

Finally, an analysis can be started for some entity once another property for that entity is computed (**R10**). Here, the analysis specifies which the property kind as well as the to be triggered analysis.

When registering an analysis, the developer may also report precomputed values to the blackboard (**R8**). This may include, e.g., native methods or base cases (e.g., specifying `Object` as immutable).

4.5 Execution Constraints

Once the end user chooses a set of analyses to be executed (**R2**), BlaSt first checks and automatically enforces restrictions on analyses that can be executed together based on the declarative specifications from Section 4.4. First, BlaSt ensures that any property kind is computed by at most one analysis, or in a collaborative way; this is to avoid that conflicting results are reported to the blackboard. Second, if several analyses derive a property kind collaboratively, BlaSt ensures that they are all either optimistic or pessimistic. Finally, BlaSt ensures that all property kinds that are queried by some analyses are derived by another analysis or a fallback is provided; this is to ensure that dependencies can be satisfied.

BlaSt's blackboard may run optimistic and pessimistic analyses simultaneously. But, when doing so, it ensures that no intermediate results are propagated among them (**R5**). Given a property kind p that is computed optimistically and a pessimistic analysis a depending on p, BlaSt does not forward any intermediate values of p to a's CF. The latter is triggered only when a value of p is submitted marked as final. We say that the dependency of a on p is *suppressed*. There are subtle interactions between dependency suppression and cyclic and collaborative computations, which we explain next.

First, there can be no cyclic dependencies between pessimistic and optimistic analyses. The correctness of cyclic dependency resolution relies on the assumption that all intermediate approximations have been processed and no further updates to any property involved in the cycle may happen (cf. Section 4.6). This obviously is not the case when updated are suppressed.

The interaction between dependency suppression and collaboratively computed properties is more involved. Assume a collaboratively computed property p_1 that (transitively) depends on another collaboratively computed property p_2 and consider the case when one or more of the transitive dependencies between them may be suppressed[4]. In this case, BlaSt must ensure that p_2 values are committed as final before p_1's values can be committed as final, too. This ensures that final values are propagated along the suppressed dependencies. To this end, BlaSt derives a *commit order* when checking the execution constraints before executing analyses. The commit order is a partial order between collaboratively computed property kinds: p_1 must be finalized later than any other collaboratively computed property kind p_2 on which p_1 depends, when there is suppression between them.

4.6 Fixed-Point Computation

Computation is started for the entities selected by eager analyses (**R11**) (cf. Section 4.4). The blackboard schedules activations whenever intermediate values for properties are submitted, distributing updated results to analyses that depend on them. It starts new computations for properties that are requested lazily (**R12**), are triggered by some analyses reaching a certain entity (**R10**), or whenever it is guided to do so by running analyses (**R13**). The above activations are done until no further updates are generated – the blackboard has reached a *quiescent* state. At this point, however, state of the properties may not necessarily final, as there still may be unresolved dependencies. There are three cases to be considered.

First, an analysis was scheduled for some property kind p, but it did not analyze some entity e, for which p was requested, e.g., because e was not reachable in the call graph. In this case, the *default value* (**R9**) is inserted, which may trigger further computations, until quiescence is reached again.

Second, properties that cyclically depend on each other are not finalized yet. If such properties form a *closed strongly connected component*, i.e., they do not have any dependees outside of the cycle (but other properties may still depend on them), they are now finalized to their current value. By requiring analyses to report their results in a monotonous and scheduling independent way (cf. Section 4.2), BlaSt guarantees that the cycle resolution is deterministic and sound. Again, further computations may arise from resolving cyclic dependencies (including supplying more default values and resolving further cycles), until quiescence is reached again.

Finally, the blackboard commits final values for collaboratively computed properties. It respects the *commit order* computed previously (cf. Section 4.5): After committing a set of collaboratively computed properties, computation is resumed again. Only once quiescence is reached again, the next property kinds, as given by

[4]On a chain of dependencies, more than one may be suppressed. Also, if p_1 depends on p_3 and p_4 and each of those depends on p_2, there is more than one path between p_1 and p_2, on which dependencies may get supressed.

the commit order, are committed. This is repeated until all collaboratively computed properties have been committed.

4.7 Scheduling and Parallelization

Blackboard systems require a control component that upon updates of the blackboard decides which knowledge sources to activate next. In our case, this control component determines the order in which activations of dependent analyses are executed and is called *scheduler*. The order in which dependent analyses are activated can have significant effects on performance [64].

BlaSt allows for the scheduler to be easily exchanged in order to select the best performing one for any chosen set of analyses. Apart from general strategies such as first-in-first-out, more specific algorithms may use the dependency structure or the values of intermediate approximations to decide the scheduling order. This is similar to the control component of blackboard systems asking knowledge sources for an estimated information gain.

Blackboard systems lend themselves well to parallelization. The individual knowledge source, i.e., analyses in our case, are decoupled and their activations both for the initial analysis as well as for updates of dependee information, can be executed fully in parallel on multiple threads. Updates to the blackboard, on the other hand, can be synchronized on a special thread or, if that becomes a bottleneck, distributed to several threads based on the property kind and/or entity. A simple implementation may just have any thread able to access all data and synchronize each individual access.

4.8 Summary

BlaSt's approach fosters strong decoupling of reified lattices (choice of data structures), analyses (choice of algorithm), and the blackboard-based solver infrastructure (the concrete fixed-point solving implementation). This enables exchanging and optimizing these parts in independently. As reified lattices are the basis for all communication between analyses, different versions of analyses can be implemented at different trade-offs. The solver manages execution of analyses, tracks dependencies and propagates updates, performs monotonicity checks, and computes the fixed-point solution.

5 EVALUATION

We evaluate our approach by answering the following questions:

RQ1 Does BlaSt support modularization of a broad range of static analysis kinds with varying requirements?
RQ2 Does exchangeability of analysis modules benefit the end user and the developer?
RQ3 Can the framework be parallelized?
RQ4 What is the benefit of analysis-specific data structures?
RQ5 How does the performance of BlaSt's analyses compare to state-of-the-art declarative approaches?

We implemented BlaSt as a Scala library on top of the OPAL framework for JVM bytecode analysis [26]. However, the approach is framework and language independent. We answer the above research questions using the case studies of Chapter 3 to analyze the DaCapo 2006 benchmark [7]. We choose DaCapo because Doop, which we compare to in Section 5.5, has special support for it.

All measurements were performed in a Docker container[5] on a server with two AMD(R) EPYC(R) 7542 @ 2.90 GHz (32 cores / 64 threads each) CPU and 512 GB RAM. The analyses were run using an OpenJDK 11.0.5+10 (64-bit) with 32 GB of heap memory. Experiments were run seven times and we report the median runtime of their executions. We report only excerpts of the results in the paper; the entire results can be found in the supplementary material.

5.1 Modularization of Various Analyses

To answer **RQ1**, we implemented the case studies from Chapter 3 using BlaSt and argue that these are representatives of different analysis kinds. The first case study represents pessimistic analyses in the context of improving the precision of a three-address code representation (TAC)—it shows how basic analyses can be extended by analyses that are specialized to increase sub-problems' precision. The modular call graph of the second case study involves two tightly interacting yet decoupled analyses (points-to and call graph) and demonstrates how one can plug in further modular analyses that handle special cases of Java in order to increase the call graph's soundness. The third case study introduced several exchangeable analyses for different high-level properties (immutability, escape information, purity). The individual analyses are relatively simple and can focus on their respective property, but by using the results of other analyses as their execution unfolds, they can be more precise than a monolithic analysis of medium complexity.

To achieve this modularity, several requirements need to be satisfied (cf. Table 1). Section 4 explains how BlaSt supports all of them. On the contrary, as we argue in Section 3.4, no current imperative or declarative framework supports all these requirements.

We additionally implemented a solver for *inter-procedural, finite, distributive subset problems* (IFDS) [63], a well-known general framework for dataflow problems based on graph reachability. Similar to other IFDS solvers, e.g., Heros [8], users provide a domain for their dataflow facts and four flow-functions that together specify the IFDS problem. The solver starts one computation per pair of method and entry dataflow fact and these tasks need to communicate their results. We chose IFDS as it is a general framework that allows implementing many dataflow analyses and it is dissimilar from three case studies' analyses. In particular, it shows BlaSt's support for general solvers as individual analyses.

■ *BlaSt's programming model enables the implementation of dissimilar analyses fostering their modularization into a set of comprehensible, maintainable, and pluggable units. BlaSt is the only static analysis framework satisfying all requirements from Section 3.4.*

5.2 Exchangeability of Analyses

BlaSt is the first framework that strictly decouples reified property kinds from the analyses that compute them. Thus, the framework can provide different analyses computing the same property kind to cover a wide range of precision, sound(i)ness, and performance trade-offs. We examine in two experiments how this exchangeability fosters rapid probing and, ergo, benefits the analysis' developer and end user alike (**RQ2**): We explore the impact on precision in experiment one and impact on soundness in experiment two.

[5]Link omitted for double-blind review

Table 2: Purity results for different configurations (hsqldb)

Configuration	#Pure	#SEF	#Other	#Impure	⏱ Analysis
$PA_2/FMA_1/E_1$	417	482	245	2 635	2.42 s
PA_2/E_1	363	536	245	2 635	2.40 s
$PA_2/FMA_1/E_0$	417	481	241	2 640	1.93 s
PA_2	362	504	225	2 688	0.98 s
PA_1/FMA_1	415	431	0	2 933	0.93 s
PA_0/FMA_1	104	0	0	3 675	0.70 s
PA_0	100	0	0	3 679	0.13 s

Table 3: Results for different call graph modules for Xalan

Configuration	#Reachable Methods	#Edges	⏱ Analysis
RTA	6 141	46 946	8.58 s
RTA_C	6 162	47 154	8.76 s
RTA_R	8 404	63 821	10.07 s
RTA_X	12 937	106 516	12.99 s
RTA_C_X	12 958	106 743	12.86 s
RTA_S_T_F_C_X	12 970	106 778	13.35 s

C=Configured native methods; R=Reflection; X=Tamiflex; S=Serialization; T=Threads; F=Finalizer;

In our first experiment, we run various configurations of our purity analysis (cf. Section 3.3), with different supporting analyses for field mutability or escape information with different precision-scalability trade-offs. Table 2 shows the results for *hsqldb*. High indices indicate more precise analyses. Comparing the least precise analysis PA_0 with the most precise $PA_2/FMA_1/E_1$, we observe a reduction in the number of reported impure methods by ~28%, but a runtime slowdown by 18.6x. Some configurations even have a large impact on runtime for almost no gain in precision, e.g., comparing the most precise one with that using simpler escape analysis E_0.

In the second experiment, we evaluate the RTA call graph with different supporting modules for different JVM features. Results for *Xalan* are shown in Table 3, displaying the active modules, the number of reachable methods, call edges, and respective construction time. Compared to the baseline, RTA with support for preconfigured native methods (RTA_C), reaches 21 more methods and produces ~200 more call edges. Reflection support (RTA_R) brings over 2 000 additional reachable methods and 16 000 call edges; at the same time, the construction time increases by roughly 15%. Using the Tamiflex (RTA_X) module instead increases the call graph's size (and soundness) more but introduces further slowdown. With all modules enabled, we reach 111% more methods and 127% more call edges, at the cost of a 55% increased runtime. Moreover, the data suggests that different supporting modules benefit different projects. Tamiflex impacted *Xalan* and *jython*, the reflection module *fop*, and the serialization module had significant influence on *hsqldb*. This last observation highlights that it may be worth investigating tradeoffs even at the level of individual projects.

Overall, both experiments confirm that BLAST maintains exchangeability benefits from Datalog-based analyses, while generalizing these results to a broader range of lattices.

■ *BLAST facilitates systematic investigation of different configurations, supporting users and developers in finding the best trade-off between precision, sound(i)ness, and scalability.*

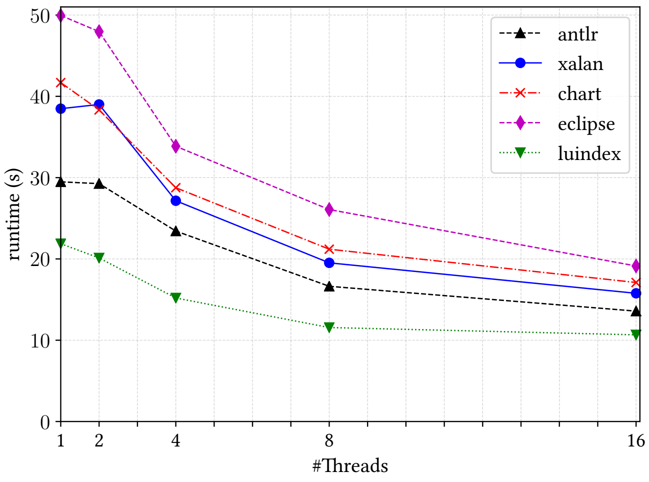

Figure 2: Parallel architecture performance

5.3 Parallelization

We implemented a proof-of-concept parallel version of our blackboard and control (**RQ3**). Using this, we measured the execution time for the points-to-based call graph with different numbers of threads. The results for five DaCapo projects are shown in Figure 2. The projects were selected to have similar runtime to facilitating graph readability. Benefits of parallelization over one thread appear at about two to four threads and we achieve speedups of up to two times for 16 threads. Beyond this, further improvement is negligible; instead, it slightly decreases due to growing communication overhead. These results are encouraging, given that the parallel version is not at all optimized. An optimized version of it is expected to scale better. Designing such an optimized version requires further research to identify the optimal way to parallelize the computation.

■ *BLAST's computation can be parallelized and that parallelization holds potential for increased performance.*

5.4 Benefits of Specialized Data Structures

To answer **RQ4**, we compare two versions of the same points-to based call-graph algorithm. Both encode points-to, caller, and callee information as integer values. The standard version uses specialized, trie-based data structures, the second one uses standard Scala sets.

Results are given in the last two columns of Table 4. Due to high memory consumption of, we had to run the version using Scala's data structures with 128 GB of heap space; *jython*'s analysis even required 256 GB. Using tailored data structures, BLAST's runtime decreased by 65% to 98% compared to naively using Scala's sets.

■ *Selecting suitable data structures adapted to the specific analysis needs is an important factor for analysis performance. While the analysis developer can freely select optimized data structures in BLAST, strictly declarative approaches do not support such choices.*

Table 4: Runtime of points-to based call graph algorithms

Project	⏱ DOOP			⏱ BLAST	⏱ BLAST
	Compile	Facts	Analysis		(Scala)
antlr	107 s	35 s	41 s	28.36 s	305.90 s
bloat	109 s	21 s	33 s	34.43 s	266.08 s
chart	109 s	38 s	45 s	40.13 s	516.37 s
eclipse	109 s	19 s	17 s	44.89 s	343.69 s
fop	110 s	41 s	35 s	18.87 s	56.64 s
hsqldb	109 s	38 s	32 s	19.65 s	55.69 s
jython	108 s	24 s	90 s	77.65 s	3 341.62 s
luindex	108 s	21 s	19 s	19.34 s	62.57 s
lusearch	108 s	21 s	20 s	21.03 s	70.55 s
pmd	109 s	39 s	36 s	21.47 s	75.47 s
xalan	108 s	37 s	30 s	35.59 s	246.97 s
geo. mean	108.54 s	29.09 s	32.51 s	29.68 s	191.26 s

5.5 Comparison with Declarative Approaches

After evaluating individual unique features of BLAST in isolation, in this section we present the results of an experiment that directly compares the performance of BLAST with that of Doop (**RQ5**), which on top of the Soufflé [43] Datalog engine, is a highly optimized state-of-the-art tool for declarative Java points-to and call-graph analyses. Specifically, we compare the runtime of our points-to based call-graph algorithm from Section 3.2 to that of Doop [12]. Doop represents the state-of-the-art in the restricted domain of points-to analysis. Furthermore, its declarative approach assembles a fair comparison as it supports most features we aim for, while offering good trade-offs between pluggable precision/recall and performance. This makes Doop an appropriate subject for comparison.

For better comparability, we disabled the reflection support in both tools, because the respective approaches are different. The applications were analyzed together with OpenJDK 1.7.0_75 (used for the TamiFlex data in Doop's benchmarks). Minor differences remain, but these are in Doop's favor, since they result in more work to be done by BLAST[6]. Still, the fifth column of Table 4 shows that our complete analysis, including all preprocessing, is often faster than Doop's analysis (9% in the geometric mean). Further, Doop additionally requires time for rule compilation and fact generation.

Note that we used BLAST's single-threaded implementation here since it seems that Doop is hardly parallelized (fact generation was done with 128 threads, but did not significantly vary with other values for the `fact-gen-cores` parameter and the `souffle-jobs` parameter did not show any effects). Using the unoptimized parallel version, e.g., with eight threads, BLAST should be able to outperform Doop even more significantly as shown in Section 5.3.

■ *Despite being more general, i.e., not tuned for points-to analyses but supporting many different kinds of analyses, BLAST clearly outperforms Doop.*

6 RELATED WORK

In this section, we discuss several related approaches in various areas of static analysis as well as in blackboard systems.

[6]For instance, BLAST does handle some cases of reflection more soundly even with reflection handling disabled in order to process the DaCapo benchmark correctly.

6.1 Blackboard Systems

The blackboard metaphor was introduced by Newell [54] and implemented within the HEARSAY-II speech-recognition system [31]. Blackboard systems were used for image recognition [51], vessel identification [57], or industrial process control [25]. The structure of blackboard systems is described, e.g., by Nii [56], Craig [21], and Corkill [17]. Corkill also discusses concurrent execution of knowledge sources and the control component [16], similar to BLAST.

For these domains, no efficient, deterministic way of solving these problems is known. This leads to several problems as mentioned by Buschmann et al. [14]: Nondeterminism makes testing difficult, good solutions are not guaranteed, performance suffers from wrong hypotheses, and development effort is high due to ill-defined problem domains. As static analysis have a well-defined domain and deterministic algorithms, these do not apply to BLAST.

BLAST resembles a more modern interpretation of blackboard systems [22]: its blackboard is not organized hierarchically and analyses may keep state between activations. Information is, however, never erased and all communication is done via the blackboard.

Brogi and Ciancarini used the blackboard approach to provide concurrency for their Shared Prolog language [13]. Like static analyses, this domain is well-defined. Their knowledge sources are restricted to be Prolog logical programs, while BLAST's analyses can be implemented in a way best suited to the analysis needs.

Decker et al. [24] discuss the importance of heuristics for scheduling concurrent knowledge source activations. Focusing on static analyses and well-defined dependency relations, BLAST provides good general heuristics which are agnostic to individual analyses.

6.2 Abstract Interpretation

Cousot et al. [19] have proven that multiple (possibly cheap) abstract domains (i.e., analyses) can be combined using the reduced product to increase overall precision. In abstract interpreters, such as Astrée [20] or Clousot [32], dependencies between domains are restricted by the execution order. Thus, the same program statement must be analyzed multiple times which is superfluous with BLAST's explicit dependency management. Also, abstract interpretation typically aims to compute abstract *approximations* [18] of concrete values, such as an integer variable's value. BLAST further allows natural expression of analyses on all granularity levels. Keidel and Erdweg [45] allow modular and reusable abstract semantics for different language features allowing soundness proofs from composition of already sound components. The analyses again approximate single concrete program values. BLAST supports analyses to be based on abstract interpretation and includes such analyses, but generalizes to a much broader range of static analyses.

6.3 Declarative Analyses Using Datalog

Datalog is often used to implement static analyses in a strictly declarative fashion [27, 35, 48, 62, 71, 72]. Properties are represented as relations and rules specify how to compute them. This enables modularization, as rules can be easily exchanged and/or added (e.g. for new language features). The Doop [12] framework, building on top of the highly optimized Datalog solver Soufflé [43], has shown that the rule-based approach enables precise and scalable points-to analyses. For this reason, Doop became the state-of-the-art for such analyses [44, 65, 67, 69, 70]. Datalog-based frameworks, however, are limited in their expressiveness by using relations, i.e., set-based abstractions, to represent all analysis results. BlaSt's approach combining imperative and declarative features provides similar benefits as Datalog-based approaches, while allowing for more expressive ways to represent data and to implement analyses.

Datalog's limitation to relations has also been pointed out by Madsen et al. [52]. They propose Flix to overcome this restriction using a language inspired by both Datalog and Scala to specify declarative pluggable analyses using arbitrary lattices as in BlaSt. However, Flix focuses on verifying soundness and safety properties of static analyses and not on performance. Szabó et al. [68] also extends Datalog to allow arbitrary lattices for static analysis. Their solver IncA focuses on incrementalization. BlaSt allows optimizations, e.g., of used data structures or scheduling strategies. Furthermore, the analyses' coarser granularity compared to individual rules reduces overhead in parallelization.

6.4 Attribute Grammars

Attribute grammars [46] used in compilers such as JastAdd [30] enable modular inference of program properties by adding computation rules to the nodes of a program's abstract syntax tree (AST). Circular reference attribute grammars [33, 37, 42, 53] enable attributes to depend on arbitrary AST nodes and allow circular dependencies. In traditional attribute grammars, attributes may only depend on parent, sibling, and child nodes. Still, analyses are tightly bound to the AST, impeding natural expression of analyses based on different structures, such as a control-flow graph. Similar to BlaSt, JastAdd enables pluggability for new language features. However, JastAdd requires at least one attribute in a cyclic dependency to be marked explicitly, while BlaSt handles this transparently.

Öqvist and Hedin [58] proposed concurrent evaluation of low complexity attributes in circular reference attribute grammars. BlaSt on the other hand supports arbitrary granularity of concurrent computation. BlaSt's explicit dependency management enable analyses to drop dependencies and commit final results early for improved performance. Finally, as memorization of properties is done in BlaSt's blackboard, local temporary values are garbage collected automatically, whereas JastAdd requires explicit removal.

6.5 Imperative Approaches and Parallelization

Lerner et al. [49] proposed a framework to modularly compose dataflow analyses. Analyses communicate implicitly through optimizations of the analyzed code or explicitly through *snooping*. A standard fixed-point algorithm repeatedly reanalyzes the code, while BlaSt's explicit dependency management avoids reanalysis. Their approach is restricted to standard dataflow analyses, while BlaSt enables a wide range of analyses including dataflow analyses.

CPAchecker [5] is a tool for configurable software verification and analysis. For any combination of analyses, CPAchecker requires defining a compound analysis to integrate the results of the individual analyses and to manage their interaction. In the case of CPA+ [6], combined analyses must also work with the same domain and must provide an explicit measure of result precision. In contrast, BlaSt enables tight interaction and interleaved execution of independently-developed analyses without requiring an additional compound analysis layer or explicit measure of precision.

Johnson et al. [41] present a framework for collaborative alias analysis. Clients ask queries which are processed by a sequence of analyses. Each analysis can answer the query or forward it to the next one. Analyses can generate additional (premise) queries to be answered by the framework. To ensure termination, a complexity metric must be defined and premises must be simpler than the queries they originate from. Therefore, cyclic dependencies, required for optimal precision, are not supported. Partial results combined from different analyses are also not supported.

Parallel execution of static analyses is performed by Magellan [29]. In this framework, dependencies are given by the data processed instead of explicitly by the analyses.

Haller et al. [36] concurrently execute tasks based on lattices and apply this to static analysis. Their framework requires dependencies to be managed fully by the client while BlaSt requires only the specification of dependencies and manages them automatically.

7 THREATS TO VALIDITY

One threat to the validity of our evaluation is the use of the relatively old and small DaCapo benchmark. It is, however, widely used to evaluate Doop [12] and to compare other approaches with Doop [2, 3, 59, 69]. Doop's special support for the benchmark makes it a particularly fair evaluation set. Furthermore, our experiment design, based on relative comparisons, should yield the same results with any well-assembled benchmark.

Also, our results are threatened if our points-to analysis is not sufficiently similar to Doop. To achieve comparability, we tailored our points-to analysis to be as similar as possible, i.e., the call graph derived from the points-to results should be almost identical. In order to ensure this, we systematically studied Doop's Datalog rules and validated the resulting call graphs using both Judge [61] and manually inspected points-to sets from deviating call graphs.

8 CONCLUSION

We presented BlaSt, a framework for modular static analyses that can be developed in isolation and exchanged for fine-tuning precision, sound(i)ness, and performance. BlaSt is general enough to support a broad range of analyses and is the first framework to explicitly support lazy collaboration of optimistic and pessimistic analyses. Based on a blackboard architecture, BlaSt combines imperative and declarative features to enable analysis-specific optimizations, outperforming state-of-the-art declarative analysis.

Our evaluation suggests to research more efficient ways to parallelize BlaSt. Results on scheduling are promising and more scheduling strategies should be examined, both general and analysis specific. This includes scheduling strategies that abort computations whose results are no longer of interest. It may be possible to have scheduling strategies as well as analyses change their behavior dynamically during the execution in order to increase performance for no or minor impact on precision and/or soundness. More different kinds of analyses should be implemented in the framework to ensure that it is usable for as many static analyses as possible.

BlaSt is open source and available on GitHub[7].

[7]Link omitted for double-blind review

REFERENCES

[1] Alexander Aiken. 1999. Introduction to set constraint-based program analysis. *Science of Computer Programming* 35, 2-3 (1999), 79–111.

[2] Karim Ali and Ondřej Lhoták. 2012. Application-only call graph construction. In *European Conference on Object-Oriented Programming (ECOOP)*. Springer, 688–712.

[3] Karim Ali and Ondřej Lhoták. 2013. Averroes: Whole-program analysis without the whole program. In *European Conference on Object-Oriented Programming (ECOOP)*. Springer, 378–400.

[4] David F Bacon and Peter F Sweeney. 1996. Fast static analysis of C++ virtual function calls. *ACM Sigplan Notices* 31, 10 (1996), 324–341.

[5] Dirk Beyer, Thomas A Henzinger, and Grégory Théoduloz. 2007. Configurable software verification: Concretizing the convergence of model checking and program analysis. In *International Conference on Computer Aided Verification*. Springer, 504–518.

[6] Dirk Beyer, Thomas A Henzinger, and Grégory Théoduloz. 2008. Program analysis with dynamic precision adjustment. In *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 29–38.

[7] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 169–190.

[8] Eric Bodden. 2012. Inter-procedural Data-flow Analysis with IFDS/IDE and Soot. In *SOAP*. 3–8. https://doi.org/10.1145/2259051.2259052

[9] Eric Bodden. 2018. The Secret Sauce in Efficient and Precise Static Analysis. In *International Workshop on State Of the Art in Java Program analysis,(SOAP)*. 84–92.

[10] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. 2011. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*. ACM, 241–250.

[11] Martin Bravenboer and Yannis Smaragdakis. 2009. Exception analysis and points-to analysis: better together. In *Proceedings of the eighteenth international symposium on Software testing and analysis (ISSTA)*. ACM, 1–12.

[12] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses *(OOPSLA)*.

[13] Antonio Brogi and Paolo Ciancarini. 1991. The concurrent language, shared prolog. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 13, 1 (1991), 99–123.

[14] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. 1996. *A System of Patterns*. Pattern-Oriented Software Architecture, Vol. 1. Wiley.

[15] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C. Sreedhar, and Sam Midkiff. 1999. Escape Analysis for Java. In *OOPSLA*.

[16] Daniel D Corkill. 1989. Design alternatives for parallel and distributed blackboard systems. In *Blackboard Architectures and Applications*.

[17] Daniel D Corkill. 1991. Blackboard systems. *AI expert* 6, 9 (1991), 40–47.

[18] Patrick Cousot and Radhia Cousot. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL)*. ACM, 238–252.

[19] Patrick Cousot and Radhia Cousot. 1979. Systematic design of program analysis frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages (POPL)*. ACM, 269–282.

[20] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2006. Combination of abstractions in the ASTRÉE static analyzer. In *Annual Asian Computing Science Conference (ASIAN)*. Springer, 272–300.

[21] Iain D Craig. 1988. Blackboard systems. *Artificial Intelligence Review* 2, 2 (1988), 103–118.

[22] Iain D Craig. 1993. *A New Interpretation of The Blackboard Metaphor*. Technical Report. Technical report, Department of Computer Science University of Warwick.

[23] Jeffrey Dean, David Grove, and Craig Chambers. 1995. Optimization of object-oriented programs using static class hierarchy analysis. In *European Conference on Object-Oriented Programming (ECOOP)*. Springer, 77–101.

[24] Keith Decker, Alan Garvey, Marty Humphrey, and Victor R Lesser. 1991. Effects of Parallelism on Blackboard System Scheduling. In *International Joint Conferences on Artificial Intelligence (IJCAI)*. 15–21.

[25] Roland Dodd, Andrew Chiou, Xinghuo Yu, and Ross Broadfoot. 2009. Industrial process model integration using a blackboard model within a pan stage decision support system. In *2009 Third International Conference on Network and System Security (NSS) (NSS)*. IEEE, 489–494.

[26] Michael Eichberg and Ben Hermann. 2014. A Software Product Line for Static Analyses: The OPAL Framework. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*. ACM, ACM, 1–6.

[27] Michael Eichberg, Sven Kloppenburg, Karl Klose, and Mira Mezini. 2008. Defining and continuous checking of structural program dependencies. In *Proceedings of the 30th international conference on Software engineering (ICSE)*. ACM, 391–400.

[28] Michael Eichberg, Florian Kübler, Dominik Helm, Michael Reif, Guido Salvaneschi, and Mira Mezini. 2018. Lattice based modularization of static analyses. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*. ACM, 113–118.

[29] Michael Eichberg, Mira Mezini, Sven Kloppenburg, Klaus Ostermann, and Benjamin Rank. 2006. Integrating and Scheduling an Open Set of Static Analyses. *(ASE)*.

[30] Torbjörn Ekman and Görel Hedin. 2007. The JastAdd Extensible Java Compiler. In *Proceedings of the ACM on Programming Languages (OOPSLA)*.

[31] Lee D Erman, Frederick Hayes-Roth, Victor R Lesser, and D Raj Reddy. 1980. The Hearsay-II speech-understanding system: Integrating knowledge to resolve uncertainty. *ACM Computing Surveys (CSUR)* 12, 2 (1980), 213–253.

[32] Manuel Fähndrich and Francesco Logozzo. 2010. Clousot: Static contract checking with abstract interpretation. In *International Conference on Formal Verification of Object-oriented Software (FoVeOOS)*. Springer, 10–30.

[33] Rodney Farrow. 1986. Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. In *ACM SIGPLAN Notices*, Vol. 21. ACM, 85–98.

[34] David Grove and Craig Chambers. 2001. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 23, 6 (2001), 685–746.

[35] Elnar Hajiyev, Mathieu Verbaere, and Oege De Moor. 2006. Codequest: Scalable source code queries with datalog. In *European Conference on Object-Oriented Programming (ECOOP)*. Springer, 2–27.

[36] Philipp Haller, Simon Geries, Michael Eichberg, and Guido Salvaneschi. 2016. Reactive Async: expressive deterministic concurrency *(SCALA)*.

[37] Görel Hedin. 2000. Reference attributed grammars. *Informatica (Slovenia)* 24, 3 (2000), 301–317.

[38] Dominik Helm, Florian Kübler, Michael Eichberg, Michael Reif, and Mira Mezini. 2018. A unified lattice model and framework for purity analyses *(ASE)*.

[39] W. Huang, A. Milanova, W. Dietl, and M. D. Ernst. 2012. ReIm & ReImInfer: Checking and inference of reference immutability and method purity *(OOPSLA)*.

[40] Simon Holm Jensen, Anders Møller, and Peter Thiemann. 2010. Interprocedural analysis with lazy propagation. In *International Static Analysis Symposium*. Springer, 320–339.

[41] Nick P Johnson, Jordan Fix, Stephen R Beard, Taewook Oh, Thomas B Jablin, and David I August. 2017. A collaborative dependence analysis framework. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization (CGO)*. IEEE Press, 148–159.

[42] Larry G Jones. 1990. Efficient evaluation of circular attribute grammars. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12, 3 (1990), 429–462.

[43] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. 2016. Soufflé: On synthesis of program analyzers. In *International Conference on Computer Aided Verification (CAV)*. Springer, 422–430.

[44] George Kastrinis and Yannis Smaragdakis. 2013. Hybrid context-sensitivity for points-to analysis. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 423–434.

[45] Sven Keidel and Sebastian Erdweg. 2019. Sound and Reusable Components for Abstract Interpretation. In *Proceedings of the ACM on Programming Languages (OOPSLA)*, Vol. 3. ACM, 176.

[46] Donald E Knuth. 1968. Semantics of context-free languages. *Mathematical systems theory* 2, 2 (1968), 127–145.

[47] Thomas Kotzmann and Hanspeter Mössenböck. 2005. Escape Analysis in the Context of Dynamic Compilation and Deoptimization. In *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments (VEE)*.

[48] Monica S Lam, John Whaley, V Benjamin Livshits, Michael C Martin, Dzintars Avots, Michael Carbin, and Christopher Unkel. 2005. Context-sensitive program analysis as database queries. In *Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, 1–12.

[49] Sorin Lerner, David Grove, and Craig Chambers. 2002. Composing dataflow analyses and transformations *(POPL)*.

[50] Ondřej Lhoták and Laurie Hendren. 2003. Scaling Java points-to analysis using Spark. In *International Conference on Compiler Construction*. Springer, 153–169.

[51] Hongyi Li, Rudi Deklerck, Bernard De Cuyper, A Hermanus, Edgard Nyssen, and Jan Cornelis. 1995. Object recognition in brain CT-scans: knowledge-based fusion of data from multiple feature extractors. *IEEE Transactions on medical imaging (TMI)* 14, 2 (1995), 212–229.

[52] Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. 2016. From Datalog to FLIX: A declarative language for fixed points on lattices. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 194–208.

[53] Eva Magnusson and Görel Hedin. 2007. Circular reference attributed grammars—their evaluation and applications. *Science of Computer Programming* 68, 1

(2007), 21–37.
[54] Allen Newell. 1962. *Some problems of basic organization in problem-solving programs*. Technical Report. Rand Corp Santa Monica CA.
[55] F. Nielson, H. Nielson, and C. Hankin. 2005. *Principles of Program Analysis*.
[56] H Penny Nii. 1986. The blackboard model of problem solving and the evolution of blackboard architectures. *AI magazine* 7, 2 (1986), 38–38.
[57] H Penny Nii, Edward A Feigenbaum, and John J Anton. 1982. Signal-to-symbol transformation: HASP/SIAP case study. *AI magazine* 3, 2 (1982), 23–23.
[58] Jesper Öqvist and Görel Hedin. 2017. Concurrent circular reference attribute grammars. In *10th ACM SIGPLAN International Conference on Software Language Engineering (SLE)*. ACM, 151–162.
[59] Edgar Pek and P Madhusudan. 2014. Explicit and symbolic techniques for fast and scalable points-to analysis. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis (SOAP)*. ACM, 1–6.
[60] Sara Porat, Marina Biberstein, Larry Koved, and Bilha Mendelson. 2000. Automatic detection of immutable fields in Java. In *Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative research (CASCON)*. IBM Press, 10.
[61] 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 (ISSTA)*. ACM, 251–261.
[62] Thomas W Reps. 1995. Demand interprocedural program analysis using logic databases. In *Applications of Logic Databases*. Springer, 163–196.
[63] Thomas W Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*. ACM, 49–61.
[64] Jonathan Rodriguez and Ondřej Lhoták. 2011. Actor-based parallel dataflow analysis *(CC)*.
[65] Yannis Smaragdakis, George Balatsouras, George Kastrinis, and Martin Bravenboer. 2015. More sound static handling of Java reflection. In *Asian Symposium on Programming Languages and Systems*. Springer, 485–503.
[66] Yannis Smaragdakis and Martin Bravenboer. 2010. Using Datalog for fast and easy program analysis. In *International Datalog 2.0 Workshop*. Springer, 245–251.
[67] Yannis Smaragdakis, Martin Bravenboer, and Ondřej Lhoták. 2011. Pick your contexts well: understanding object-sensitivity. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 17–30.
[68] Tamás Szabó, Gábor Bergmann, Sebastian Erdweg, and Markus Voelter. 2018. Incrementalizing lattice-based program analyses in Datalog. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (2018), 1–29.
[69] Tian Tan, Yue Li, and Jingling Xue. 2016. Making k-object-sensitive pointer analysis more precise with still k-limiting. In *International Static Analysis Symposium*. Springer, 489–510.
[70] Tian Tan, Yue Li, and Jingling Xue. 2017. Efficient and precise points-to analysis: modeling the heap by merging equivalent automata. *ACM SIGPLAN Notices* 52, 6 (2017), 278–291.
[71] John Whaley. 2007. *Context-sensitive pointer analysis using binary decision diagrams*. Ph.D. Dissertation. Stanford University.
[72] John Whaley and Monica S Lam. 2004. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. *ACM SIGPLAN Notices* 39, 6 (2004), 131–144.