A Parallel Worklist Algorithm for Modular Analyses

Van Es, Noah; Stiévenart, Quentin; Van der Plas, Jens; De Roover, Coen

Published in:
Proceedings of the 20th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2020)

DOI:
10.1109/SCAM51674.2020.00006

Publication date:
2020

Document Version:
Accepted author manuscript

A Parallel Worklist Algorithm for Modular Analyses

Noah Van Es, Quentin Stiévenart, Jens Van der Plas, Coen De Roover
Software Languages Lab, Vrije Universiteit Brussel, Belgium
{noah.van.es,quentin.stievenart,jens.van.der.plas,coen.de.roover}@vub.be

***Abstract*—One way to speed up static program analysis is to make use of today's multi-core CPUs by parallelising the analysis. Existing work on parallel analysis usually targets traditional data-flow analyses for static, first-order languages such as C. Less attention has been given so far to the parallelisation of more general analyses that can also target dynamic, higher-order languages such as JavaScript. These are significantly more challenging to parallelise, as dependencies between analysis results are only discovered during the analysis itself. State-of-the-art parallel analyses for such languages are therefore usually limited, both in their applicability and performance gains.**

In this work, we propose the parallelisation of *modular* analyses. Modular analyses compute different parts of the analysis in isolation of one another, and therefore offer inherent opportunities for parallelisation that have not been explored so far. In addition, they can be used to develop a general class of analysers for dynamic, higher-order languages. We present a parallel variant of the worklist algorithm that is used to drive such modular analyses. To further speed up its convergence, we show how this algorithm can exploit the monotonicity of the analysis. Existing modular analyses can be parallelised without additional effort by instead employing this parallel worklist algorithm. We demonstrate this for MODF, an inter-procedural modular analysis, and for MODCONC, an inter-process modular analysis. For MODCONC, we reveal an additional opportunity to exploit even more parallelism in the analysis.

Our parallel worklist algorithm is implemented and integrated into MAF, a framework for modular program analysis. Using a set of Scheme benchmarks for MODF, we usually observe speedups between $3\times$ and $8\times$ when using 4 workers, and speedups between $8\times$ and $32\times$ when using 16 workers. For MODCONC, we achieve a maximum speedup of $15\times$.

***Index Terms*—Static Program Analysis, Modular Analysis, Parallelism, Concurrency, Dynamic Languages**

I. INTRODUCTION

In order to be useful, static analyses require both good precision and performance. High precision can be achieved through various techniques, such as increasing context-sensitivity [12], [28], [29] or using a more precise abstract domain [5], [30]. Unfortunately, such precision-increasing techniques often come at the cost of increasing the complexity of the analysis, therefore also impacting its performance. Consequently, a combination of high precision and high performance is harder to achieve, and the lack of performance is often mentioned as one of the prime reasons why developers eschew the usage of static analysers altogether [4], [14], [27].

An obvious approach to speed up the analysis is to exploit today's prevalence of multi-core CPU architectures and *parallelise* the analysis. There is ample existing work on developing such parallel analyses. However, most of the existing parallel analysers target rather static languages such as C [2], [16], [19], [22], [36]. An advantage when parallelising the analysis for such languages is that the control-flow dependencies of the analysed program are almost entirely known beforehand (i.e., the *inter-procedural call graph* is available a priori). For instance, the SATURN analyser [36] exploits these call dependencies by parallelising a *bottom-up analysis*, in which a function is only analysed after all its callees have been analysed. Such a *bottom-up analysis* is almost trivially parallelisable: the analysis can start by analysing all functions at the bottom of the call dependency graph (those without any callees) in parallel, then analyse all subsequent functions whose callees have already been analysed in parallel, and so on[1]. Using this approach, the authors report speedups up to almost $30\times$ on an 80-core machine.

In contrast, less attention has been given so far to the parallel analysis of highly dynamic, higher-order languages such as JavaScript or Scheme. Parallelising an analysis for these languages is more challenging, as the control-flow behaviour of the program and dependencies between analysis results are not known beforehand, and only discovered during the analysis itself. Dewey et al. [9] parallelise JSAI, an abstract interpreter for JavaScript, by exploring multiple independent program states in the analysis in parallel. Compared to parallel analyses for static languages, the speedups are more modest here: on 12 cores, most benchmarks achieve a $2-4\times$ speedup. Furthermore, as the exploration of program states is parallelised by context, their approach is only able to parallelise context-sensitive analyses, and its efficiency is very dependent upon the impact of context-sensitivity for the analysed program.

In this work, we present a novel approach to automatically parallelise a general class of analysers for highly dynamic, higher-order languages. Specifically, we propose the parallelisation of *modular analyses*. In a modular analysis [6], a program is split up into different components (such as the functions in the program), and those components are repeatedly analysed in isolation. The key insight is that the modularity of the analysis can be exploited to parallelise the analysis: as the analysis of a single component is done in isolation, multiple components can safely be analysed in parallel. We consider a general and modern formulation of modular analyses for highly dynamic and higher-order languages, which has recently been used for both traditional inter-procedural analysis [23] (referred to as MODF) and for the inter-process analysis of concurrent programs [31] (referred to

[1]Note that in such a bottom-up analysis, functions that are (mutually) recursive form a strongly connected component (SCC) in the call graph, and need to be analysed together in a single fixed-point computation.

as MODCONC). Although these analyses have been touted for their applicability and scalability, so far no attention has been given to their inherent opportunities for parallelisation. Note that, compared to the bottom-up analysis of SATURN [36], these analyses are not bottom-up, but rather *top-down* modular analyses, as the control-flow dependencies of the program are necessarily only discovered during the analysis itself due to the language's dynamic nature. Parallelisation for such a top-down modular analysis proves to be less trivial. One reason is that due to the lack of a priori information on dependencies, it becomes harder to efficiently determine which components need to be analysed in parallel. Another reason is that such modular analyses are more general than the bottom-up analyses of SATURN: components that are analysed in parallel could lead to new dependencies or changes to the global analysis state, requiring other components that may be impacted by these changes to be re-analysed. We formulate a new parallel worklist algorithm that can analyse multiple components in parallel, while ensuring a correct coordination of the analysis to obtain the exact same result as the traditional sequential worklist algorithm. Since this worklist algorithm is completely agnostic of the particular instantiation of the modular analysis (e.g., MODF or MODCONC), one can apply it to any existing modular analysis *for free*.

The contributions of this paper are the following:

- We propose the parallelisation of modular analyses to easily and efficiently analyse dynamic, higher-order languages in parallel. A parallel worklist algorithm is given to automatically render a modular analysis parallel.
- We apply our novel parallelisation strategy to two existing modular analyses, MODF and MODCONC. In particular, we demonstrate how the MODCONC analysis can be made "doubly parallel" by analysing the MODCONC components using a parallel MODF analysis.
- We implement our parallel worklist algorithm in MAF, a framework to develop modular analyses. In our evaluation on a set of Scheme benchmarks, most speedups range between $8\times$ and $32\times$ when using 16 workers for MODF. For MODCONC, we achieve speedups up to $15\times$.

II. BACKGROUND: MODULAR ANALYSIS

A modular static program analysis splits up a program into several *components*. Ideally, this enables a "divide-and-conquer" approach to program analysis: all components of a given program are analysed in isolation of one another (using an *intra-component analysis*), and the analysis results of the different components are combined to obtain the analysis result for the entire program. The exact definition of a component can be chosen depending on the given program and the goal of the analysis, and typically represents an abstraction of some run-time entities in the program (such as function calls or threads). For instance, in MODF, a component represents (an approximation of) a function call, and the intra-component analysis of a single component amounts to an intra-procedural analysis of that function call.

However, the intra-component analyses of different components may depend on one another. For instance, if components are function calls, and component f_1 is the caller of component f_2, then the return value of f_2 needs to be known for the intra-component analysis of f_1. In turn, the argument values that are supplied by f_1 are necessary for the intra-component analysis of f_2. We say that component f_1 has a *dependency* on the return value of f_2, while component f_2 has a dependency on the argument values supplied by f_1. To deal with such mutual dependencies, we can employ a fixed-point computation, as proposed by Cousot & Cousot [6]. This fixed-point computation is referred to as the *inter-component* analysis. It repeatedly analyses components (using the intra-component analysis) with respect to the current analysis state. This is necessary because the intra-component analysis of some component c_1 may update some part of the analysis state that another component c_2 has a dependency on. When that happens, we say that a dependency of c_2 is *triggered*, and subsequently c_2 is re-analysed using the updated analysis state. The intra-component analysis must be monotone to ensure that the analysis state eventually converges. The fixed-point iteration is repeated until all components have been analysed and the analysis state has converged (so that no dependencies are triggered after analysing some component). Note that for dynamic, higher-order languages, such as JavaScript and Scheme, both components and dependencies are not know beforehand and only discovered during the analysis itself.

A sequential algorithm for this inter-component analysis is given in Algorithm 1. In this algorithm, we leave the definition of components and the corresponding intra-component analysis open as configurable parameters. Different choices for these parameters lead to different instantiations of modular analyses, such as MODF (Section II-A) and MODCONC (Section II-B). The inter-component algorithm discussed here, however, is the same for any of these modular analyses.

Algorithm 1: Sequential worklist algorithm computing the fixed-point for the inter-component analysis.

Data: An initial component c_0 and initial store σ_0
Result: A sound, over-approximated analysis result for the behaviour of the corresponding program

```
1  W = {c0}, V = {c0}, σ = σ0, D = λaddr. ∅
2  while W ≠ ∅ do
3      pick any c ∈ W
4      W := W \ {c}
5      (σ', C, R, T) = ANALYSE(c, σ)
6      σ := σ'
7      W := W ∪ (C \ V)
8      V := V ∪ C
9      foreach a ∈ R do D := D[a ↦ D(a) ∪ {c}]
10     foreach a ∈ T do W := W ∪ D(a)
```

We assume that some initial component c_0 represents the entry point of the program. For instance, if components represent function calls, then c_0 would represent the initial call

to the `main` function of the program. The worklist algorithm maintains the following iteration variables (line 1):

- A worklist W of components that still need to be analysed. Initially, this worklist only contains c_0.
- A visited set V, which is used to keep track of all components that have already been discovered during the analysis, and initially also contains only c_0.
- The global analysis state. For simplicity, we model this using a store σ, although in general the global analysis state can also encompass more than just a store. The store σ models an approximation of the run-time heap of the program. It maps abstract addresses (addresses that approximate real heap addresses) to abstract values (values that approximate real heap values). We assume that an initial store σ_0 (with initial bindings) is given.
- A dependency map D. Dependencies encode some part of the global analysis state that an intra-component analysis can depend on. In this case, the global analysis state is just a store, and so dependencies are addresses of that store. The dependency map D tracks for each dependency (i.e., address) a the set of all components that depend on the value in the store at address a. This way, we immediately know which components need to be re-analysed if a dependency is triggered (which happens when the value at that address in the store is changed).

The algorithm uses a sequential worklist iteration: as long as the worklist is not empty (line 2), it arbitrarily picks (line 3) and removes (line 4) a component c from the worklist. This component is then analysed using the function ANALYSE, which performs the intra-component analysis of component c (line 5). As previously mentioned, this function is considered a parameter of the analysis, returning a tuple (σ', C, R, T):

- σ' is the updated store after the intra-component analysis (note that the current store σ is also passed to ANALYSE). After executing the intra-component analysis, we continue the iteration with this updated store (line 6).
- C is the set of components that have been discovered during the intra-component analysis. For instance, if components represent function calls, then this is the set of all components representing the function calls that were made by the analysed function call component. We add all unseen components to the worklist W (line 7), and register these components in the visited set V (line 8).
- R is the set of dependencies that the analysis of this component relied on. This corresponds to the set of addresses in σ that were read by the intra-component analysis. For every address a, we add the analysed component c to the set of components reliant on that dependency[2].
- T is the set of dependencies that the analysis of this component triggered. This corresponds to the set of addresses in σ that were written to by the intra-component analysis. For every address a, we must re-analyse all potentially impacted components, and hence add all components reliant on a (i.e., $D(a)$) to the worklist (line 10).

[2]We write "$f[x \mapsto y]$" as a shorthand for "$\lambda v.$ if $v = x$ then y else $f(v)$".

In order for the analysis to terminate, we must assume that only a finite number of components can be discovered for a given program. Usually, this is ensured by approximating the actual run-time entities (e.g., function calls) by components holding both the corresponding lexical program elements (e.g., function definitions), which are necessarily finite, plus some contextual information (e.g., the call site of the function call) taken from a finite set. This context is used to have multiple components for the same lexical program element, and is often referred to as the *context-sensitivity* of the analysis. Context-sensitivity can be used to tune the precision of the analysis: using more contextual information allows for more components in the analysis so that each component may be analysed with higher precision, whereas less contextual information means that more run-time entities need to be approximated by the same component.

Similarly, the store σ can only use a finite number of addresses. Abstract values should be taken from a partially ordered set $(L, \sqsubseteq)$ with a commutative and associative *join operator* $\sqcup$, so that $\forall a, b \in L : a \sqsubseteq a \sqcup b$. The choice of L depends on the target analysis: an abstract value in L can approximate a set of real values, or encode other program properties (e.g., for a reaching definition analysis) using any abstract lattice domain. The set L should satisfy the *ascending chain condition* (ACC): every sequence of abstract values $l_0 \sqsubseteq l_1 \sqsubseteq l_2 \sqsubseteq ...$ should eventually converge: there should exist some n so that $l_n = l_i$ for any $i \geq n$. Values at a certain address in the store are never overwritten, but only joined with newer values at that address using the join operator, so that the ACC guarantees that eventually all values in the store will converge[3], and no more dependencies will be triggered. Furthermore, the intra-component analysis should be monotone: the result of ANALYSE(c, σ_2) should subsume the result of ANALYSE(c, σ_1) if $\sigma_1 \sqsubseteq \sigma_2$[4]. It can be shown that these properties, combined with a finite number of components, ensure that the analysis will always terminate with the same result, regardless of the order in which components are picked from the worklist. Nevertheless, the order in which components are analysed can influence in what sequence the abstract values in σ will converge and how many iterations are required in the algorithm. In practice, it can therefore have a significant impact on analysis performance.

A. MODF: *Inter-Procedural Modular Analysis*

One instantiation of this modular analysis framework is MODF [23]. In MODF, a component represents a function call (technically, an *approximation* of several function calls). The intra-component analysis for a component in MODF then boils down to an intra-procedural analysis of the function call(s) represented by that component. The resulting inter-component analysis for MODF, obtained by using this intra-procedural analysis for the ANALYSE function in Algorithm 1, can therefore be seen as an inter-procedural analysis.

[3]Our approach can trivially be extended to also support infinite ascending chains with acceleration techniques such as widening to ensure convergence.

[4]We write "$\sigma_1 \sqsubseteq \sigma_2$" if and only if "$\forall a : \sigma_1(a) \sqsubseteq \sigma_2(a)$".

We do not formally define the ANALYSE function for MODF here, as it is not relevant to the remainder of this paper. Rather, we illustrate at a high-level how MODF works by example.

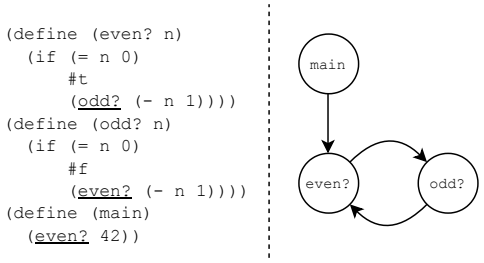

Fig. 1. A MODF analysis example (left: the Scheme program under analysis; right: the call graph computed by a context-insensitive MODF analysis).

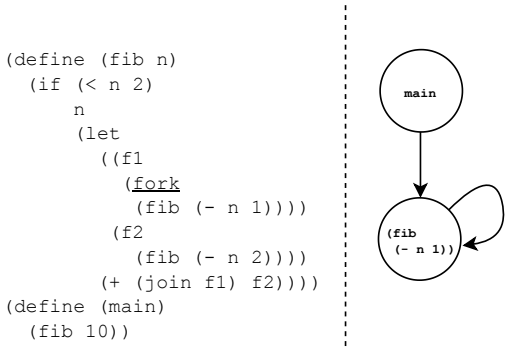

Fig. 2. A MODCONC analysis example (left: the Scheme program under analysis; right: a graph computed by a context-insensitive MODCONC analysis, showing how threads fork other threads in the program).

Consider the snippet of Scheme code shown in Figure 1. For this example, we employ a context-insensitive analysis: this means that all calls to the same function are represented by the same component. Since we only have 3 functions in this program, that means our analysis will only discover 3 components. Assuming that the `main` function is the entry point of our program, the analysis will start with $c_0 = \texttt{main}$. During the intra-procedural analysis of this component (using the function ANALYSE), the call to `even?` does not immediately lead to an analysis of that function. Rather, `even?` is added to the set of discovered components C returned by ANALYSE, which is then added to the worklist W, and later analysed in another iteration of the inter-component analysis. The return value of this call is immediately looked up in the store σ at an address dedicated to the latest return value of the `even?` component. Due to the subsequent dependency on this address, the `main` component will be re-analysed if (after analysing `even?`) the return value of the `even?` component stored at this address is updated. Similarly, the analysis of `even?` will lead to the discovery of the `odd?` component. While the analysis of the `odd?` component discovers a different call to the `even?` function, due to the context-insensitivity that call is also approximated by the already discovered `even?` component. The resulting call graph that can be constructed from this analysis is shown on the right-hand side of Figure 1.

B. MODCONC*: Inter-Process Modular Analysis*

Another instantiation of the modular analysis framework is MODCONC [31], which can be used to analyse concurrent programs that spawn multiple processes. In MODCONC, a component represents a thread (technically, an *approximation* of several threads). The intra-component analysis in MODCONC then boils down to an intra-process analysis of the thread(s) represented by a component. The resulting inter-component analysis for MODCONC, obtained by using this intra-process analysis for the ANALYSE function in Algorithm 1, can therefore be seen as an inter-process analysis.

We again use an example to explain how MODCONC works (Figure 2). Without additional context for components, the analysis uses one component per `fork` expression in the program, plus one component for the (implicit) main thread; the latter is the initial component c_0. Again, when a new thread is forked in the `fib` function, this thread is not immediately analysed; rather, a new component is created that is analysed later using another intra-component analysis for that thread. This thread will spawn new threads that are all approximated by the same component, hence the self-loop in Figure 2.

While the intra-component analysis (i.e., the function ANALYSE) for MODF could easily be carried out using a simple intra-procedural analysis, it is less trivial to design the intra-process analysis required by MODCONC. In general, the behaviour of a single thread is as challenging to analyse as the behaviour of any other single-threaded program, and therefore an intra-process analysis requires a full inter-procedural analysis. We can repurpose MODF to carry out this inter-procedural analysis, meaning that the intra-component analysis in MODCONC can be defined using a MODF analysis.

As a final note, observe that MODF offers a full inter-procedural analysis, given the definition of an intra-procedural analysis. Similarly, MODCONC offers a full inter-process analysis, given the definition of an intra-process analysis. In general, for any definition of "component", Algorithm 1 can be used to design an inter-component analysis, given a definition of an intra-component analysis. This is one of the main strengths of the modular analysis framework, as the former is usually several times harder to design than the latter.

III. PARALLEL MODULAR ANALYSIS

Algorithm 1 shows how the inter-component analysis can be computed using a sequential worklist algorithm. In this section, we propose a novel *parallel* worklist algorithm to compute the inter-component analysis, obtaining the exact same result as the sequential worklist algorithm. The core idea is simple: multiple components in the worklist W can be analysed in parallel. A key benefit of the modular analysis design is that it automatically provides coarse-grained tasks (i.e., entire intra-component analyses) that can be run in parallel because they are executed in isolation of one another.

We first present our core parallel worklist algorithm in Section III-A. Then, we propose some optimisations that can be applied to this algorithm to further increase its parallel efficiency (Section III-B). Finally, we discuss its application to MODF and MODCONC (Section III-C).

A. Parallel Inter-Component Analysis

Our parallel inter-component analysis algorithm uses several worker threads to analyse multiple components in parallel. To avoid synchronisation costs in updating the global analysis state (i.e., σ in Algorithm 1), each component is analysed using its own local copy of that analysis state. A single thread is responsible for processing the incoming results of the intra-component analyses and subsequently updating the global analysis state. Therefore, our parallel worklist algorithm follows the "coordinator-worker" paradigm, which can easily be implemented using several concurrency mechanisms (e.g., using actors). Processing results using a single coordinator thread introduces a sequential bottleneck, but trivially avoids issues with race conditions. We expect throughput to be mostly dominated by the cost of the intra-component analyses, which can be processed in parallel using multiple workers.

Insights. There are two key insights that we leverage for the parallel worklist algorithm.

First, when multiple intra-component analyses are executed concurrently, it may occur that one updates some part of the analysis state that the other depends on. Since state is kept local to each worker during the intra-component analysis, such updates happening in one intra-component analysis are not visible in another. Therefore, after each intra-component analysis, it should be checked that no part of the analysis state that was read during that intra-component analysis has been updated in the meantime; if so, the component should be analysed again. We do not require any additional bookkeeping for this, as we can exploit the dependencies that the intra-component analysis relied on (i.e., the set R returned by the ANALYSE function) to check if another intra-component analysis might have interfered in its computation.

Second, we can exploit the monotonicity of the analysis when updating the analysis state. If an intra-component analysis updates some part of the analysis state, we can always apply these changes to the global analysis state. Even if the local analysis state used during the intra-component analysis is no longer up-to-date (i.e., not using the latest σ), it would be subsumed by the newer state (because σ only "grows"), and the monotonicity of the intra-component analysis guarantees that all observed updates would still be valid. Moreover, because of the associative and commutative properties of the join operator, we can apply those updates on a more up-to-date version of the analysis state. Again, we do not require any additional bookkeeping for this, as we can exploit the dependencies that the intra-component analysis modified (i.e., the set T returned by the ANALYSE function) to track all updates to the analysis state after an intra-component analysis.

Algorithm. Algorithm 2 shows a parallel variant of the inter-component analysis.

Algorithm 2: Parallel worklist algorithm computing the fixed-point for the inter-component analysis.

Data: An initial component c_0 and initial store σ_0
Result: A sound, over-approximated analysis result for the behaviour of the corresponding program

```
1  S = {c0}, V = {c0}, σ = σ0, D = λaddr. ∅
2  function SCHEDULE(c):
3      fork
4          σ_local = σ
5          result = ANALYSE(c, σ_local)
6          send (c, σ_local, result) to coordinator
7  SCHEDULE(c0)
8  while S ≠ ∅ do
9      wait for next result (c, σ_local, (σ', C, R, T))
10     foreach c' ∈ C do
11         if c' ∉ V then
12             V := V ∪ {c'}
13             S := S ∪ {c'}
14             SCHEDULE(c')
15     foreach a ∈ R do D := D[a ↦ D(a) ∪ {c}]
16     foreach a ∈ T do
17         if σ'(a) ⋢ σ(a) then
18             σ := σ[a ↦ σ(a) ⊔ σ'(a)]
19             foreach c' ∈ D(a) do
20                 if c' ∉ S then
21                     S := S ∪ {c'}
22                     SCHEDULE(c')
23     if ∃a ∈ R : σ_local(a) ≠ σ(a) then
24         SCHEDULE(c)
25     else
26         S := S \ {c}
```

Similar to the sequential algorithm, it keeps track of a visited set V, a store σ and a dependency map D (line 1). However, there is no longer a worklist W: components that need to be analysed are directly sent off to a worker. This happens using the function SCHEDULE (lines 2-6): a new task is forked (which we assume is eventually assigned to a worker thread) that will analyse the given component (using the ANALYSE function) and send back the result when finished[5]. A set of components S keeps tracks of which components have been scheduled using this function, ensuring that a single component is never scheduled multiple times at once.

To kickstart the analysis, we schedule the initial component c_0 (line 7). The coordinator thread then enters a loop (lines 8-26) which stops once all the scheduled components have been analysed (i.e., when S is empty). In each iteration, it updates the analysis state after receiving an incoming result of an intra-component analysis of some component c (line 9).

Analogous to the sequential algorithm, it first schedules

[5]In an actual implementation, it is necessary to make σ a `volatile` variable. The coordinator thread modifies σ, while worker threads only read σ. It is not necessary for a worker to have the latest version of σ; the σ_{local} variable is introduced to ensure only a single version of σ is read.

every discovered component that has not yet been visited[6] (lines 10-14), then registers the dependencies of the component that was analysed (line 15),

Updating the analysis state becomes more complicated for the parallel algorithm: we cannot just replace σ with σ' (as in the sequential algorithm), since it is possible that σ has been updated during the intra-component analysis of component c (in which case $\sigma_{local} \neq \sigma$). Therefore, we need to apply each update that happened during the intra-component analysis (as indicated by T) to the current analysis state σ. We first check (line 17) for every updated address a, if the updated value $\sigma'(a)$ is already subsumed by the current value for that address $\sigma(a)$. If so, this means that another intra-component analysis already updated $\sigma(a)$ in a way that the current update is already incorporated in the analysis state. Otherwise, we join the current value $\sigma(a)$ with the updated value $\sigma'(a)$ (line 18). Again, we cannot just replace $\sigma(a)$ with $\sigma'(a)$, since $\sigma(a)$ may have been updated, in which case both $\sigma(a) \not\sqsubseteq \sigma'(a)$ and $\sigma'(a) \not\sqsubseteq \sigma(a)$. The monotonicity of the analysis combined with the commutative and associative properties of the join operator allow us to join both values. By performing updates to the analysis state in this way, the parallel analysis can speed up its convergence. Then, as in the sequential algorithm, we also schedule all components that may have been affected by the triggered dependency a (lines 19-22).

Finally, we must check if component c needs to be re-analysed because some part of the analysis state it depends on (as indicated by R) was updated. Concretely, we check if any dependency a that was used during the intra-component analysis has been updated in the meantime (line 23). If so, we must re-analyse component c to take into account this updated analysis state (line 24). If not, the analysis result of c has been processed, meaning that we can remove it from S (line 26).

Discussion. The parallelism that can be exploited by this algorithm will depend on the number of components that are scheduled at the same time (i.e., based on the size of the set S). The analysis behaviour is non-deterministic in the order in which scheduled components are analysed, the order in which concurrent intra-component analyses finish, and the order in which their results are processed. This significantly influences in what order and in how many iterations the analysis state will converge; however, it has no impact on the final analysis result, which is always the same as that of Algorithm 1.

B. Optimising for Parallel Efficiency

There are several optimisations that can be applied to Algorithm 2 to further enhance its parallel efficiency.

Prioritising components. Although the order in which scheduled components are analysed and subsequently processed does not influence the final analysis result, it can affect performance. If two components c_1 and c_2 are scheduled, and c_1 relies on some analysis state that c_2 updates, then analysing both c_1 and c_2 in parallel would result in having to re-analyse c_1. We can avoid such issues by prioritising which scheduled components need to be analysed first. Xie et al. [36] analyse callees in parallel before their callers are analysed, and other existing work [13] on parallel analysis has also confirmed the importance of prioritised scheduling to improve parallel efficiency. Similarly, we can design a heuristic to prioritise components "at the bottom of the call graph" by assigning a *depth* to each component. The initial component c_0 has depth 0. Whenever a component is discovered (lines 12-14 in Algorithm 2), it is given the depth of the component that discovered it plus one. This does not perfectly calculate the actual depth of each component; however, it makes the heuristic inexpensive to compute, and ensures that in general, a component will have a higher depth than its "ancestors". For instance, in MODF this will generally give a called function a higher depth than its caller (but may not do so for mutually recursive functions, or if the component representing the callee has already been discovered). The worker threads are then configured to analyse components with a higher depth first.

Timestamped dependencies. In practice, checking if no dependencies were changed by comparing abstract values in the store (i.e., using the equality checks on line 23) could be somewhat expensive, at least in this context where we want to avoid a sequential bottleneck by reducing the workload of the coordinator. A simple workaround is to assign a *timestamp* to each dependency (initially zero for every dependency). Whenever the analysis state is updated (line 18), we increase the timestamp of the corresponding dependency. As such, we can eliminate the expensive equality checks on abstract values, instead replacing them with much cheaper equality checks on the timestamps of the dependencies. Using such timestamps for cheaper comparisons goes back to earlier work on optimising the performance of static analysers [15], [28].

Filtering intra-component analysis results. Note that the cost of processing the results is directly linked to the size of the C, R and T sets returned by the ANALYSE function. We can reduce the workload on the single-threaded coordinator further, by first filtering these sets in the worker after computing the intra-component analysis. Specifically, the worker can remove all components in C which are already included in the visited set V, since these are discarded anyway on line 11. Similarly, it can filter the sets R and T to only include dependencies that are not yet registered for the analysed component or where the corresponding updates are not yet subsumed by the current analysis state, respectively. Note that it would not be safe to remove the corresponding if-checks in Algorithm 2: the analysis state may have been updated after the filtering of the results, but before they are processed. On the other hand, the filtering itself is always safe due to the increasing nature of the analysis (e.g., the set V only grows).

C. Application to MODF and MODCONC

Since the parallelisation targets the inter-component analysis, no additional effort is required to parallelise existing modular analyses such as MODF and MODCONC: one only needs to replace the sequential with the parallel worklist algorithm. For MODF, this results in an inter-procedural analysis

[6]Note that $c' \notin V$ implies $c' \notin S$ here.

in which multiple function calls are analysed in parallel. For MODCONC, this results in an inter-process analysis in which multiple processes are analysed in parallel.

However, for MODCONC, the modular analysis design allows us to exploit even more parallelism, since the intra-component analysis of a MODCONC analysis can be implemented using a MODF analysis. By using a parallel MODF analysis, both the inter- and intra-component analysis of MODCONC can trivially be rendered parallel.

IV. EVALUATION

Our parallel worklist algorithm, including the optimisations discussed in Section III-B, has been integrated into the MAF framework [32], where it is made available as a Scala trait that can directly be used with any existing modular analysis. We setup our experiments (Section IV-A) using this implementation, and measure the speedups of the parallel version of MODF (Section IV-B) and MODCONC (Section IV-C) over their respective sequential implementations. In addition, we have also verified the correctness of the parallel implementations by running the built-in soundness tests of MAF and by verifying that analysis results are the same as those of the sequential implementation for all benchmarks.

A. Setup

Execution environment. All experiments are conducted on a server using a 4-socket AMD Opteron 6376 CPU. Each socket contains 2 dies that consists of 8 physical cores (which share cache memory) each, therefore allowing to run 64 threads simultaneously in total. The server uses Java 8 and Scala 2.13.3; we configured the JVM with a fixed heap size of 64GB. Each experiment was preceded by 10 warm-up runs (with a total timeout of 1 minute), and reported results are averaged over 20 measurements.

Benchmarks. We use a total of 22 Scheme benchmarks from the built-in benchmark suite of MAF, 14 for MODF (cf. Table I) and 8 for MODCONC[7] (cf. Table II). In contrast to some of the existing work on parallel analysis [9], our approach is not dependent on the choice of context-sensitivity: for MODF, we include 7 context-insensitive benchmarks, as well as 7 context-sensitive ones (using 2-CFA [28]). Specifically, we use a context-insensitive analysis for benchmarks that would be too slow using a context-sensitive analysis, and vice versa (with the exception of the `sboyer` benchmark, which we analyse both context-senstively and context-insensitively). For MODCONC, we run all analyses with high context-sensitivity (5-CFA), since otherwise they terminated too quickly to produce relevant results. For all benchmarks, the abstract values come from a constant propagation domain [5], allowing the analysis to derive the type and, if possible, the constant value of every expression in the program.

Sequential implementation. As advocated by Dewey et al. [9], we report speedups relative to the sequential implementation (which is not always the case for speedups reported in other related work [18], [21], [22], [34]). For a fair comparison, this sequential implementation uses the same heuristic on the *depth* of components (cf. Section III-B) to pick components from the worklist. The total running time of the sequential MODF and MODCONC implementations are included in Table I and Table II, respectively.

[7]The MODCONC benchmarks are written in an extended version of R5RS Scheme that includes special forms and primitives to support concurrency.

TABLE I
AN OVERVIEW OF THE MODF BENCHMARKS. LOC INDICATES LINES OF CODE (AS CALCULATED BY `CLOC`). A CHECKMARK (✓) IN THE **CS?** COLUMN MEANS THAT THE BENCHMARK WAS ANALYSED CONTEXT-SENSITIVELY (USING 2-CFA). THE LAST COLUMN INDICATES THE RUNNING TIME USING THE SEQUENTIAL MODF IMPLEMENTATION.

Benchmark	LOC	CS?	Sequential
`multiple-dwelling`	402	✗	42s
`decompose`	412	✗	43s
`prime-sum-pair`	394	✗	31s
`eceval`	774	✗	2m9s
`peval`	497	✗	1m58s
`scheme`	767	✗	8m5s
`sboyer-A`	632	✗	55s
`graphs`	384	✓	2m13s
`matrix`	617	✓	1m12s
`nboyer`	625	✓	16s
`sboyer-B`	632	✓	1m36s
`browse`	161	✓	20m46s
`compiler`	466	✓	17s
`leval`	334	✓	16s

TABLE II
AN OVERVIEW OF THE MODCONC BENCHMARKS. LOC INDICATES LINES OF CODE (AS CALCULATED BY `CLOC`). THE LAST COLUMN INDICATES THE RUNNING TIME USING THE SEQUENTIAL MODCONC IMPLEMENTATION.

Benchmark	LOC	Sequential
`actors`	105	1m38s
`crypt`	163	8s
`matmul`	111	2m28s
`minimax`	96	4m44s
`msort`	38	18s
`random`	19	12s
`stm`	130	44s
`sudoku`	84	2s

B. ModF Experiments

The speedups of the parallel MODF implementation for context-insensitive and context-sensitive benchmarks are shown in Figure 3 and Figure 4, respectively. In both cases, we observe rather consistent speedups up to 16 workers: using 4 workers, most benchmarks achieve a speedup between $3\times$ and $8\times$; using 16 workers, most benchmarks achieve a speedup between $8\times$ and $32\times$. When using more than 16 workers, performance gains are mostly dependent on the benchmark under analysis. In general, longer running benchmarks such as `scheme`, `browse` and `graphs` can benefit more from the additional parallelism, achieving high speedups of $217\times$, $101\times$ and $41\times$ for 64 workers, respectively. Other benchmarks improve only slightly (or not at all) when increasing the number of workers further. In particular, the `nboyer` benchmark appears to slow down when too many workers are added. One reason for this slowdown could be that the sequential implementation only takes 16 seconds to analyse

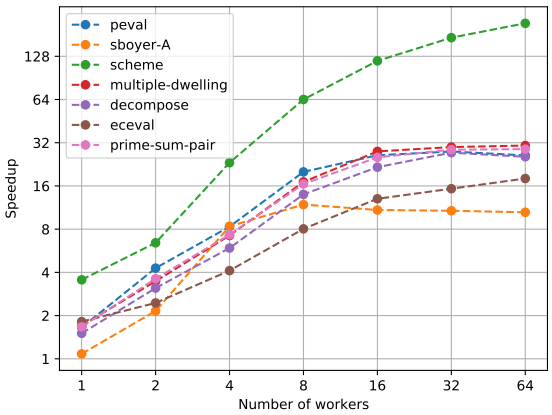

Fig. 3. Speedups for the context-insensitive MODF benchmarks.

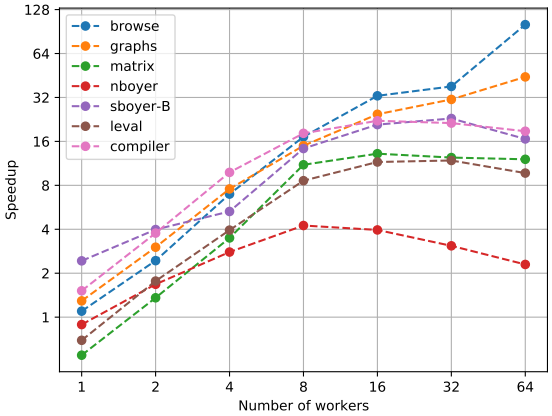

Fig. 4. Speedups for the context-sensitive MODF benchmarks.

this program; using 8 workers reduces this to less than 4 seconds, and it may not be possible to improve performance further through parallelism. Adding more workers at that point could lead to more overhead or a less favourable exploration strategy. In general, we expect speedups to become limited or even to deter for a larger number of workers: there is only a single coordinator to process all incoming results, introducing a sequential bottleneck that puts a limit on the maximum performance we can achieve, regardless of how many workers are used. However, most benchmarks seem to consistently benefit from increasing the number of workers up to 64.

Interestingly, we observe many superlinear speedups. A major factor contributing to this is the order in which components are explored. Both the sequential and parallel implementations use the same heuristic to pick the next scheduled component to analyse. However, as already mentioned in Section III-A, the exploration order is non-deterministic for the parallel implementations: the rate at which workers execute the intra-component analyses determines which components are scheduled, analysed and processed first. While this does not influence the analysis result, it does influence the performance of the analysis (for better or worse), as it determines how the analysis state converges. In addition, our parallel algorithm exploits the monotonicity of the analysis to speed up the convergence of the analysis state: every intra-component analysis can update the analysis state, even if that intra-component analysis was not using the latest analysis state itself. The increased rate at which the analysis state converges could in turn lead to fewer re-analyses of the same component, reducing the total amount of work for the parallel analysis. This indeed turns out to be the case for benchmarks with very superlinear speedups. For instance, the `scheme` benchmark performs 125997 intra-component analyses in total using the sequential implementation, whereas on a sample run of the parallel implementation with 8 workers, only 12576 intra-component analyses are required. This means that for that benchmark, the intra-component analysis workload is reduced by approximately a factor of 10, which, combined with the parallelism of using 8 workers, explains why we are able to achieve a $64\times$ speedup. Our work confirms findings reported in related work [2], [9], [10], [18] that often attributes superlinear or unexpected speedups to the exploration order.

It should also be noted that the parallel implementation with 1 worker is not identical to the sequential implementation. The former uses 2 threads in total for the analysis: 1 worker thread to perform the intra-component analyses, and 1 thread for the coordinator to process the intra-component analysis results. The latter only uses a single thread that does both sequentially. For this reason, we often already notice performance improvements when using just a single worker.

C. ModConc Experiments

As discussed in Section III-C, MODCONC can exploit parallelism at two levels: the inter-component analysis can be made parallel using Algorithm 2, while the intra-component analysis can be made parallel by using a parallel MODF analysis. In Figure 5, for each benchmark we show a matrix reporting the speedups relative to the sequential implementation for a varying number of workers in both the intra- and the inter-component analysis. The labels on top of each column indicate the number of workers used for the MODCONC inter-component analysis (henceforth referred to as parameter n). The labels on the left side of each row indicate the number of workers used *per* intra-component (MODF) analysis (henceforth referred to as parameter m). Since every intra-component analysis can use m workers, the total number of workers that can run concurrently at any given time is $m * n$.

The results highlight the need for this doubly-parallel strategy: when only analysing MODCONC components in parallel (i.e., keeping $m = 1$), we do not consistently achieve the same high speedups as for MODF. Some benchmarks (such as `matmul`, `minimax` and `random`) clearly benefit from analysing multiple processes in parallel; others (such as `actor` and `crypt`) fail to achieve large speedups when using the same configuration, sometimes even slowing down compared to the sequential implementation. One reason for this behaviour could be linked to the heuristic that prioritises components based on their *depth*. For MODF, such a heuristic clearly has merits, since it ensures that callees are usually scheduled with a higher priority than their callers (as advo-

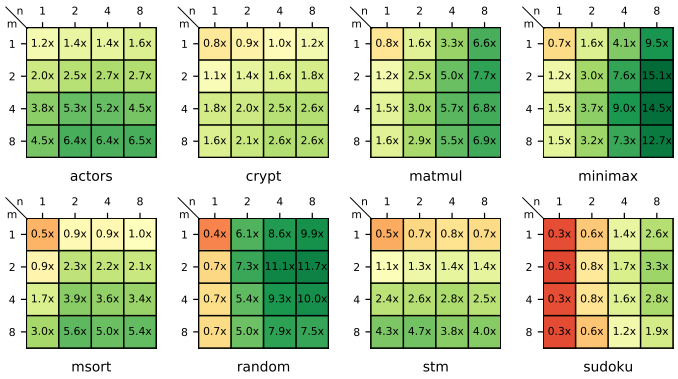

Fig. 5. Speedups for the MODCONC benchmarks, relative to the sequential implementation. Horizontally (left-to-right), we increase the number of workers for MODCONC inter-component analysis. Vertically (top-to-bottom), we increase the number of workers per intra-component (MODF) analysis.

cated in related work [13], [36]). However, for MODCONC, it is not immediately clear if prioritising a process over the process that spawned it has the same benefits. Another reason could be more technical in nature: every single intra-component analysis in MODCONC is a parallel MODF analysis, which requires more overhead to setup, orchestrate and tear down compared to the more lightweight, sequential MODF implementation. Indeed, we notice that for most benchmarks, the parallel MODCONC implementation with $n = 1$ and $m = 1$ is slower than the sequential implementation. However, we believe much of this technical overhead could be avoided in a more optimised implementation.

Increasing the parallelism of the intra-component analysis (i.e., increasing m) does seem to improve performance for most benchmarks. In general, it is recommended to use a combination of inter- and intra-component parallelism (i.e., choosing both $m > 1$ and $n > 1$), as this often appears to deliver significant (and somewhat consistent) speedups over the sequential implementation. It should be noted that the poor results for the `sudoku` benchmark are most likely caused by the fact that the analysis under sequential MODCONC takes less than 2 seconds, therefore leaving little room for improvement due to parallelisation. In contrast, longer-running benchmarks such as `matmul` and `minimax` achieve more significant speedups up to 7.7× and 15.1×, respectively.

D. Threats to Validity

We discuss potential threats to the validity of our empirical findings below. In doing so, we follow the classification recommended by Wohlin et al. [35].

A threat to *external validity* stems from our usage of small- to medium-sized Scheme benchmarks. We argue that Scheme is an excellent target language to analyse, as it is highly dynamic in nature and features higher-order functions, two characteristics that we want to support well in our parallel analysis. We compensated for the smaller scale of the benchmarks by increasing the context-sensitivity of the analysis and selecting benchmarks of interesting complexity (such as programs that run an interpreter). The built-in benchmarks of the MAF framework that we used for our experiments originate from various sources, including the SICP handbook [1] and other well-known benchmark suites[8] for Scheme [11].

A threat to *construct validity* is linked to the non-deterministic behaviour of the parallel analysis. Due to the non-predictable rate at which workers analyse components, different runs of the same parallel analysis can lead to different exploration orders, and therefore to significant differences in performance. We mitigate this threat by repeating every analysis 20 times, and reporting the average of all results.

Finally, a threat to *internal validity* is also linked to this non-deterministic exploration order, which is different to the deterministic exploration order of the sequential implementation. We have tried to keep the exploration order of the sequential implementation as close to that of the parallel analysis by using the same heuristic to pick components from the worklist. However, it is not possible to enforce the exact same exploration order, which can cause some perturbations in the results. Again, repeating our experiments several times and averaging the results also helps to mitigate this problem.

V. FUTURE WORK

There are several potential improvements to our approach that could be explored further. We discuss two of them below.

The first is related to the exploration heuristic. Our current heuristic prioritises components in the worklist based on their *depth*. The results hint that this may work well for function-modular analyses such as MODF, but could potentially be suboptimal for other modular analyses such as MODCONC. The design space for such heuristics appears to be quite large, since they can for instance also take into account the dependencies of components in the modular analysis. Therefore, an extensive study and evaluation might reveal better heuristics.

The second is related to the use of a single coordinator thread that processes incoming analysis results. As discussed earlier, this imposes a sequential bottleneck, potentially limiting speedups when using a very high number of workers. We believe there is opportunity for parallelism in processing the analysis results, although this appears to be much harder to parallelise efficiently. Despite this apparent bottleneck, our approach still appears to scale well up to a high number of workers. It may however be a reason for the suboptimal speedups we can sometimes observe when using 64 workers.

VI. RELATED WORK

Our approach falls within the domain of modular analyses, initially proposed by Cousot and Cousot [6]. In particular, we apply our approach to function-modular analyses [23] and process-modular analyses [31]. We now discuss the extensive existing work on the parallelisation of static program analyses.

A. Parallelisation of Classical Dataflow Problems

Classical dataflow problems have been parallelised by Lee and Ryder [18], achieving a speedup of 7.5× on 8 cores, and by Kramer et al. [17], computing an ideal achievable speedup

[8] http://www.larcenists.org/Twobit/benchmarksAbout.html

of up to $5.4\times$. These are applicable when the control-flow graphs of the program under analysis are known in advance. In this paper, we focus on a more general problem than classical dataflow analysis, since control-flow graphs are not available in advance and only computed during the analysis itself.

B. Parallelisation With a Static Call Graph

There have been many parallelisations of analysers for C and Java programs. For these languages, the call and control-flow graphs of the program under analysis are known statically, which is not a requirement of our analysis.

C analysers. Monniaux [22] described the parallel implementation of the Astrée static analyser [7], achieving a speedup of around $2\times$ on 5 cores. However, Cousot et al. [8] observed that beyond 4 cores, "the cost of synchronisation outweigths the speedup of parallel execution" for Astrée. The SATURN program analysis framework [36] achieved a high parallelisation by performing bottom-up modular analysis, with speedups of up to $29\times$ on 80 cores. McPeak et al. [19] presented a parallel and incremental interprocedural analysis that divides the analysis in parallel work units, integrated within the Coverity checker, achieving a speedup of up to $7\times$ on 8 cores. Recently, Kim et al. [16] have revisited Bourdoncle's algorithm [3] with parallelisation, achieving a speedup of up to $11\times$ on 16 threads. Bourdoncle's algorithm determines an optimal exploration order from a predefined dependency graph; in contrast, our worklist algorithm uses a predefined exploration strategy, since the dependencies are only discovered during the analysis itself. More closely related to our approach, Albarghouthi et al. [2] parallelised a top-down interprocedural modular analysis by relying on MapReduce-style parallelism, achieving a speedup of up to $7.4\times$ on 8 cores. This approach has similarities to ours: the map stage analyses procedures in parallel, similar to our intra-component analyses, and the reduce stage manages inter-procedural dependencies, similar to our inter-component analysis.

Java analysers. Rodriguez and Lhoták [26] presented a parallelisation of IFDS [25] based on actors, where each CFG node is represented by an actor. This is a completely different parallelisation strategy compared to ours, which parallelises the analysis of different components. They achieved a speedup of $6.1\times$ on 8 cores. Edvinsson et al. [10] parallelised *independent nodes* (e.g., independent control-flow branches or targets with different context-sensitivities), achieving a speedup of $4.4\times$ on 8 cores. Again, in our approach we do not require control-flow and call graphs to be known before the analysis.

C. Parallelisation of the Analyses of Dynamic Languages

There has been little existing work on parallelising analyses for dynamic languages, where dependencies may not be known statically. An early effort by Weeks et al. [34] parallelised an abstract interpreter for a parallel higher-order dynamic language, by partitioning the state space of the analysis for different workers. Similar to our approach, each worker uses its own local state, which can be updated without synchronisation costs. They achieve a speedup of $9.4\times$ on 16 threads [9].

More recently, Dewey et al. [9] parallelised an abstract interpreter for JavaScript, by partitioning states per context. The speedups they report range from $2\times$ to $4\times$ on 12 threads, with a few outliers up to $37\times$. In contrast, our approach does not need to partition components per context, and is therefore also applicable to context-insensitive analyses.

D. Other Forms of Parallelisation

Besides the traditional parallelisation approaches that use threads or actors, there have been attempts to parallelise static analyses through other approaches such as GPU computing and distributed systems. In contrast to these, our approach only relies on thread-based parallelism on a single machine.

Mendez-Lojo et al. [21] implemented parallel inclusion-based points-to analyses on GPUs [20], achieving a speedup of up to $3\times$ on 8 cores, and of $7\times$ on a GPU. Similarly, Prabhu et al. [24] implemented an algorithm for higher-order context-insensitive analysis on GPUs, encoding the analysis data as vectors and matrices. They achieved a speedup of up to $72\times$.

Venet and Brat [33] presented the first distributed static analysis implementation, focused on detecting out-of-bounds errors in embedded C programs. It relies on a relational database to store the analysis data, achieving speedups up to $2.7\times$ on 8 distributed CPUs. They identified communication costs as the limiting factor for a distributed analysis, showing that these costs become too high beyond 4 distributed CPUs.

VII. Conclusion

We have presented a novel approach to design parallel analyses for dynamic, higher-order languages. The key insight in our work is that modular analyses offer inherent opportunities for efficient parallelisation, which we exploit with a parallel worklist algorithm that analyses different components in parallel. Despite its non-deterministic behaviour, this algorithm obtains the exact same result as the sequential worklist algorithm, and is able to further exploit the monotonicity of the analysis to speed up its convergence. We applied this parallel worklist algorithm to two existing modular analyses: MODF, a function-modular analysis, and MODCONC, a process-modular analysis. For the latter, we reveal a further opportunity for parallelisation, resulting in a modular analysis with both a parallel inter-component and a parallel intra-component analysis.

Our implementation in the MAF framework reveals significant speedups. For MODF, using 16 workers usually delivers speedups between $8\times$ and $32\times$. For MODCONC, we observe speedups up to $15\times$. As such, in general our parallelisation strategy appears to achieve similar or better speedups compared to existing parallel analyses. In addition, it does not require the control-flow graph of the program under analysis beforehand, and can therefore also be used for analyses that target dynamic, higher-order languages. Finally, it is directly applicable to existing modular analyses, both context-sensitive and context-insensitive ones. To our knowledge, there is no other existing work that features all these qualities.

Acknowledgements

This work was partially supported by the *"Cybersecurity Initiative Flanders"* and by the Research Foundation – Flanders (FWO) (grant numbers 11D5718N and 11F4820N).

REFERENCES

[1] Harold Abelson and Gerald J. Sussman. *Structure and Interpretation of Computer Programs, Second Edition*. MIT Press, 1996.

[2] Aws Albarghouthi, Rahul Kumar, Aditya V. Nori, and Sriram K. Rajamani. Parallelizing top-down interprocedural analyses. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 217–228, 2012. doi:10.1145/2254064.2254091.

[3] François Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Formal Methods in Programming and Their Applications, International Conference, Akademgorodok, Novosibirsk, Russia, June 28 - July 2, 1993, Proceedings*, pages 128–141, 1993. doi:10.1007/BFb0039704.

[4] Maria Christakis and Christian Bird. What developers want and need from program analysis: an empirical study. In David Lo, Sven Apel, and Sarfraz Khurshid, editors, *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*, pages 332–343. ACM, 2016. doi:10.1145/2970276.2970347.

[5] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977. doi:10.1145/512950.512973.

[6] Patrick Cousot and Radhia Cousot. Modular static program analysis. In R. Nigel Horspool, editor, *Compiler Construction, 11th International Conference, CC 2002, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, volume 2304 of *Lecture Notes in Computer Science*, pages 159–178. Springer, 2002. doi:10.1007/3-540-45937-5_13.

[7] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTREÉ analyzer. In *Programming Languages and Systems, 14th European Symposium on Programming,ESOP 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, pages 21–30, 2005. doi:10.1007/978-3-540-31987-0_3.

[8] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. Why does Astrée scale up? *Formal Methods Syst. Des.*, 35(3):229–264, 2009. doi:10.1007/s10703-009-0089-6.

[9] Kyle Dewey, Vineeth Kashyap, and Ben Hardekopf. A parallel abstract interpreter for JavaScript. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2015, San Francisco, CA, USA, February 07 - 11, 2015*, pages 34–45, 2015. doi:10.1109/CGO.2015.7054185.

[10] Marcus Edvinsson, Jonas Lundberg, and Welf Löwe. Parallel points-to analysis for multi-core machines. In *High Performance Embedded Architectures and Compilers, 6th International Conference, HiPEAC 2011, Heraklion, Crete, Greece, January 24-26, 2011. Proceedings*, pages 45–54, 2011. doi:10.1145/1944862.1944872.

[11] Richard P Gabriel et al. *Performance and evaluation of LISP systems*, volume 263. MIT press Cambridge, Mass., 1985.

[12] Thomas Gilray, Michael D. Adams, and Matthew Might. Abstract allocation as a unified approach to polyvariance in control-flow analyses. *J. Funct. Program.*, 28:e18, 2018. doi:10.1017/S0956796818000138.

[13] Dominik Helm, Florian Kübler, Jan Thomas Kölzer, Philipp Haller, Michael Eichberg, Guido Salvaneschi, and Mira Mezini. A programming model for semi-implicit parallelization of static analyses. In Sarfraz Khurshid and Corina S. Pasareanu, editors, *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*, pages 428–439. ACM, 2020. doi:10.1145/3395363.3397367.

[14] Brittany Johnson, Yoonki Song, Emerson R. Murphy-Hill, and Robert W. Bowdidge. Why don't software developers use static analysis tools to find bugs? In David Notkin, Betty H. C. Cheng, and Klaus Pohl, editors, *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 672–681. IEEE Computer Society, 2013. doi:10.1109/ICSE.2013.6606613.

[15] J. Ian Johnson, Nicholas Labich, Matthew Might, and David Van Horn. Optimizing abstract abstract machines. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 443–454. ACM, 2013. doi:10.1145/2500365.2500604.

[16] Sung Kook Kim, Arnaud J. Venet, and Aditya V. Thakur. Deterministic parallel fixpoint computation. *Proc. ACM Program. Lang.*, 4(POPL):14:1–14:33, 2020. doi:10.1145/3371082.

[17] Robert Kramer, Rajiv Gupta, and Mary Lou Soffa. The combining dag: A technique for parallel data flow analysis. *IEEE Transactions on Parallel and Distributed Systems*, 5(8):805–813, 1994.

[18] Yong-Fong Lee and Barbara G. Ryder. A comprehensive approach to parallel data flow analysis. In *Proceedings of the 6th international conference on Supercomputing, ICS 1992, Washington, DC, USA, July 19-24, 1992*, pages 236–247, 1992. doi:10.1145/143369.143415.

[19] Scott McPeak, Charles-Henri Gros, and Murali Krishna Ramanathan. Scalable and incremental software bug detection. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*, pages 554–564, 2013. doi:10.1145/2491411.2501854.

[20] Mario Méndez-Lojo, Martin Burtscher, and Keshav Pingali. A GPU implementation of inclusion-based points-to analysis. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2012, New Orleans, LA, USA, February 25-29, 2012*, pages 107–116, 2012. doi:10.1145/2145816.2145831.

[21] Mario Méndez-Lojo, Augustine Mathew, and Keshav Pingali. Parallel inclusion-based points-to analysis. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*, pages 428–443, 2010. doi:10.1145/1869459.1869495.

[22] David Monniaux. The parallel implementation of the Astrée static analyzer. In *Programming Languages and Systems, Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005, Proceedings*, pages 86–96, 2005. doi:10.1007/11575467_7.

[23] Jens Nicolay, Quentin Stiévenart, Wolfgang De Meuter, and Coen De Roover. Effect-driven flow analysis. In Constantin Enea and Ruzica Piskac, editors, *Verification, Model Checking, and Abstract Interpretation - 20th International Conference, VMCAI 2019, Cascais, Portugal, January 13-15, 2019, Proceedings*, volume 11388 of *Lecture Notes in Computer Science*, pages 247–274. Springer, 2019. doi:10.1007/978-3-030-11245-5_12.

[24] Tarun Prabhu, Shreyas Ramalingam, Matthew Might, and Mary W. Hall. Eigencfa: accelerating flow analysis with GPUs. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 511–522, 2011. doi:10.1145/1926385.1926445.

[25] Thomas W. Reps, Susan Horwitz, and Shmuel Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, California, USA, January 23-25, 1995*, pages 49–61, 1995. doi:10.1145/199448.199462.

[26] Jonathan Rodriguez and Ondrej Lhoták. Actor-based parallel dataflow analysis. In *Compiler Construction - 20th International Conference, CC 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*, pages 179–197, 2011. doi:10.1007/978-3-642-19861-8_11.

[27] Caitlin Sadowski, Edward Aftandilian, Alex Eagle, Liam Miller-Cushon, and Ciera Jaspan. Lessons from building static analysis tools at Google. *Commun. ACM*, 61(4):58–66, 2018. doi:10.1145/3188720.

[28] Olin Shivers. *Control-flow analysis of higher-order languages*. PhD thesis, Carnegie Mellon University, 1991.

[29] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: understanding object-sensitivity. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 17–30. ACM, 2011. doi:10.1145/1926385.1926390.

[30] Quentin Stiévenart, Jens Nicolay, Wolfgang De Meuter, and Coen De Roover. Mailbox abstractions for static analysis of actor programs. In Peter Müller, editor, *31st European Conference on Object-Oriented*

Programming, ECOOP 2017, June 19-23, 2017, Barcelona, Spain, volume 74 of *LIPIcs*, pages 25:1–25:30. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi:10.4230/LIPIcs.ECOOP.2017.25.

[31] Quentin Stiévenart, Jens Nicolay, Wolfgang De Meuter, and Coen De Roover. A general method for rendering static analyses for diverse concurrency models modular. *J. Syst. Softw.*, 147:17–45, 2019. doi:10.1016/j.jss.2018.10.001.

[32] Noah Van Es, Jens Van der Plas, Quentin Stiévenart, and Coen De Roover. MAF: A Framework for Modular Static Analysis of Higher-Order Languages. In *20th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2020, Adelaide, Australia, September 27-28, 2020*. IEEE Computer Society, 2020.

[33] Arnaud Venet and Guillaume P. Brat. Precise and efficient static array bound checking for large embedded C programs. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004, Washington, DC, USA, June 9-11, 2004*, pages 231–242, 2004. doi:10.1145/996841.996869.

[34] Stephen Weeks, Suresh Jagannathan, and James Philbin. A concurrent abstract interpreter. *LISP Symb. Comput.*, 7(2-3):173–193, 1994.

[35] Claes Wohlin, Per Runeson, Martin Höst, Magnus C. Ohlsson, and Björn Regnell. *Experimentation in Software Engineering - An Introduction*, volume 6 of *The Kluwer International Series in Software Engineering*. Kluwer, 2000. doi:10.1007/978-1-4615-4625-2.

[36] Yichen Xie and Alex Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM Trans. Program. Lang. Syst.*, 29(3):16, 2007. doi:10.1145/1232420.1232423.