The Journal of Systems & Software 181 (2021) 111042

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

A parallel worklist algorithm and its exploration heuristics for static modular analyses☆

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

Software Languages Lab, Vrije Universiteit Brussel, Belgium

ARTICLE INFO

Article history:
Received 5 March 2021
Received in revised form 29 May 2021
Accepted 2 July 2021
Available online 16 July 2021

Keywords:
Static programme analysis
Modular analysis
Parallelism
Concurrency
Dynamic Languages

ABSTRACT

One way to speed up static programme 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: analyses of individual ModConc components can themselves be parallel, resulting in a doubly-parallel exploration. Finally, we present several heuristics for the exploration order of the analysis and discuss how they can impact its performance.

The parallel worklist algorithm and the exploration heuristics are implemented for and integrated into MAF, a framework for modular programme analysis. On a set of Scheme benchmarks for ModF, we observe speedups between 3× and 8× when using 4 workers, and speedups between 8× and 32× when using 16 workers, with a maximum speedup of 333× using 128 workers. For ModConc, we achieve a maximum speedup of 37× with 32 workers. We observe that on a ModF analysis, among 11 exploration heuristics, the heuristics prioritising either components with smaller environments or with less dependencies result in consistent speedups that can reach 20× those of a random exploration strategy. We find a clear correlation between the mean number of dependencies in a programme and the speedup obtained by this heuristic.

1. 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 (Shivers, 1991; Gilray et al., 2018; Smaragdakis et al., 2011) or using a more precise abstract domain (Cousot and Cousot, 1977; Stiévenart et al., 2017). 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 (Johnson et al., 2013b; Sadowski et al., 2018; Christakis and Bird, 2016).

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

☆ Editor: W.K. Chan.

* Corresponding author.
E-mail addresses: quentin.stievenart@vub.be (Q. Stiévenart), noah.van.es@vub.be (N. Van Es), jens.van.der.plas@vub.be (J. Van der Plas), coen.de.roover@vub.be (C. De Roover).

https://doi.org/10.1016/j.jss.2021.111042
0164-1212/

target rather static languages such as C (Monniaux, 2005; Xie and Aiken, 2007; McPeak et al., 2013; Kim et al., 2020; Albarghouthi et al., 2012). An advantage when parallelising the analysis for such languages is that the control-flow dependencies of the analysed programme are almost entirely known beforehand (i.e., the *inter-procedural call graph* is available a priori). For instance, the Saturn analyser (Xie and Aiken, 2007) 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× 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 programme and dependencies between analysis results are not known beforehand, and only discovered during the analysis itself. Dewey et al. (2015) parallelise JSAI, an abstract interpreter for JavaScript, by exploring multiple independent programme 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 programme 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 programme.

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 (Cousot and Cousot, 2002), a programme is split up into different components (such as the functions in the programme), 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 (Nicolay et al., 2019) (referred to as ModF) and for the inter-process analysis of concurrent programmes (Stiévenart et al., 2019) (referred to 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 (Xie and Aiken, 2007), these analyses are not bottom-up, but rather *top-down* modular analyses, as the control-flow dependencies of the programme 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 further explore its combination with different heuristics that determine the exploration order of the underlying analysis.
- In order to demonstrate its general applicability, 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× and 32× when using 16 workers for ModF. For ModConc, we achieve speedups up to 37× with 32 workers.

This is an extended version of previous work (Van Es et al., 2020b). This journal article extends the conference publication in multiple ways.

- We have improved and optimised the baseline analysis, greatly reducing the base sequential analysis time. This is to ensure the comparison of the parallel worklist algorithm to the baseline sequential worklist algorithm is fair. As the baseline analysis has already been optimised, real-world speedups are more difficult to achieve through parallelism. The additions made to our implementation pertain to language support, and improvements to running times are the results of various refactorings, with no specific root cause. These improvements have been applied both to the sequential and parallel versions of the analyses to ensure no bias is introduced in our evaluation.
- Because of the improvements to the baseline analysis (both in terms of performance as well as language support), our empirical evaluation could be expanded to additional benchmark programmes. We have therefore extended the evaluation (Section 5) as follows:
  - Our evaluation on ModF without context sensitivity is conducted on 28 programmes instead of 7.
  - We have included an evaluation with 1-CFA context sensitivity, which was not present in the original paper.
  - Our evaluation on ModF with 2-CFA context sensitivity is performed on 15 programmes instead of 7.
  - Our evaluation on ModConc is conducted on 16 programmes instead of 8.
- We have reworked the evaluation and now include a new research question where we evaluate the impact of different heuristics for the exploration order of an analysis on the speedup achieved through parallelisation. The original paper used a fixed exploration strategy that prioritises the analysis of components corresponding to deeper function calls; we now provide an evaluation of 11 different heuristics and changed the default heuristic to one that assigns random priorities to components.

[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.

- We provide more details about our implementation in Section 4 and provide a replication package for reproducing our evaluation.[2]

2. Background: Modular analysis

A modular static programme analysis splits up a programme into several *components*. Ideally, this enables a "divide-and-conquer" approach to programme analysis: all components of a given programme 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 programme. The exact definition of a component can be chosen depending on the given programme and the goal of the analysis, and typically represents an abstraction of some run-time entities in the programme (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 standard worklist algorithm (Cousot and Cousot, 2002; Fecht and Seidl, 1999) to compute a (least) fixed point. 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 known 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 2.1) and MODCONC (Section 2.2). The inter-component algorithm discussed here, however, is the same for any of these modular analyses.

We assume that some initial component c_0 represents the entry point of the programme. For instance, if components represent function calls, then c_0 would represent the initial call to the `main` function of the programme. The worklist algorithm maintains the following iteration variables (line 1):

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)
```

- 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 programme. 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.[3]

[2] https://github.com/softwarelanguageslab/maf/tree/parallel-evaluation.

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

- 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).

In order for the analysis to terminate, we must assume that only a finite number of components can be discovered for a given programme. Usually, this is ensured by approximating the actual run-time entities (e.g., function calls) by components holding both the corresponding lexical programme 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 programme 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* $\sqcup$ that is monotone, such that $\forall a, b \in L : a \sqsubseteq a \sqcup b$. This is for example the case when $(L, \sqsubseteq, \sqcup)$ forms a join semi-lattice. The choice of L depends on the target analysis: an abstract value in L can approximate a set of real values, or encode other programme 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 $l_0 \sqsubseteq l_1 \sqsubseteq l_2 \sqsubseteq \ldots$ 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,[4] 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 $\sigma_1 \sqsubseteq \sigma_2$.[5] 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.

2.1. MODF: *Inter-Procedural Modular Analysis*

One instantiation of this modular analysis framework is MODF (Nicolay et al., 2019). 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.

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.

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

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

```
(define (even? n)
  (if (= n 0)
      #t
      (odd? (- n 1))))
(define (odd? n)
  (if (= n 0)
      #f
      (even? (- n 1))))
(define (main)
  (even? 42))
```

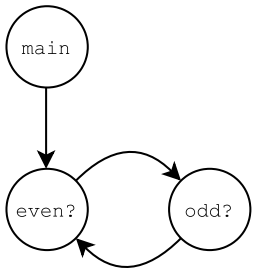

Fig. 1. A MODF analysis example (left: the Scheme programme under analysis; right: the call graph computed by a context-insensitive MODF analysis).

Consider the snippet of Scheme code shown in Fig. 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 programme, that means our analysis will only discover 3 components. Assuming that the `main` function is the entry point of our programme, the analysis will start with $c_0 =$ `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, and the argument to `even?` is store allocated at an address $a_{\texttt{even?}-0}$. As this is the only function call in `main`, the resulting set C for ANALYSE is {`even?`}. In the analysis of `main`, the return value of the call to `even?` is immediately looked up in the store σ at an address $a_{\texttt{even?}}$ dedicated to the return value of the `even?` component. This value is initially $\bot$ because `even?` has not yet been analysed. Due to this store lookup, a new dependency is registered for this dedicated address. Hence, the set R returned by the analysis of `main` is $\{a_{\texttt{even?}}\}$. Finally, at the end of the analysis of `main`, the current result is stored at location $a_{\texttt{main}}$. The set T returned by the analysis of `main` therefore is $\{a_{\texttt{main}}, a_{\texttt{even?}-0}\}$, which are the two addresses that have been written to in the analysis of `main`.

The inter-component analysis then adds `even?` to the worklist W, and proceeds by analysing the next (and only) component in the worklist: `even?`. Due to its dependency on address $a_{\texttt{even?}}$, the `main` component will be re-analysed if (after analysing `even?`) the return value of the `even?` component stored at $a_{\texttt{even?}}$ is updated.

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 Fig. 1.

2.2. MODCONC: *Inter-Process Modular Analysis*

Another instantiation of the modular analysis framework is MODCONC (Stiévenart et al., 2019), which can be used to analyse concurrent programmes 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.

```
(define (fib n)
  (if (< n 2)
      n
      (let
        ((f1
          (fork
           (fib (- n 1))))
         (f2
           (fib (- n 2))))
        (+ (join f1) f2))))
(define (main)
  (fib 10))
```

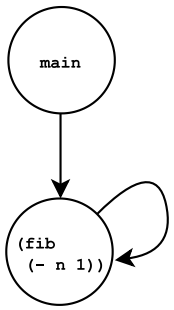

Fig. 2. A ModConc analysis example (left: the Scheme programme under analysis; right: a graph computed by a context-insensitive ModConc analysis, showing how threads fork other threads in the programme).

We again use an example to explain how ModConc works (Fig. 2). Without additional context for components, the analysis uses one component per `fork` expression in the programme, 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. Hence, the set C returned by ANALYSE for the analysis of the main thread is $\{(\texttt{fib}(-\,\texttt{n}\,1))\}$, i.e., the expression that is evaluated in the created thread. This new thread will spawn new threads that are all approximated by the same component, hence the self-loop in Fig. 2. Dependencies in ModConc are the same as in ModF: they are addresses. In the case of ModConc, a thread that completes writes its resulting value in the store at a specific address, and the `join` operation corresponding to that thread reads that value. Hence, the analysis of component `(fib (- n 1))` results in $T = \{a_{(\texttt{fib}(-\,\texttt{n}\,1))}\}$ as it writes its result to the corresponding address, and $R \supset \{a_{(\texttt{fib}(-\,\texttt{n}\,1))}\}$ as it depends on its own return value (as well as other addresses corresponding to variables, not discussed here).

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 programme, 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.

3. 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 3.1. Then, we propose some optimisations that can be applied to this algorithm to further increase its parallel efficiency (Section 3.2). Finally, we discuss its application to ModF and ModConc (Section 3.3).

3.1. 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

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 = {c_0}, V = {c_0}, σ = σ_0, 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(c_0)
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}
```

track all updates to the analysis state after an intra-component analysis.

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

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. A set S of components 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 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),

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

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).

Thread safety. We now turn to the thread safety of this algorithm. A possible data race can occur at the level of the store: it could be the case that the store used for its analysis (σ_{local}, line 4) is out of sync with the current version of the store (σ). If it is the case, then the analysis of the component may have produced results based on an outdated store. The addresses of σ_{local} that the worker depends upon are registered as dependencies in the analysis. Once the worker has finished, we check for each registered dependency if the value at the address it depends on in the store has changed since the worker fetched its version of the store (line 23 of Algorithm 2), comparing the store used by the intra-component analysis (σ_{local}) with the current store (σ). Should any addition to the store possibly impact the outcome in the analysis of a component, the component will therefore have been rescheduled for analysis. Hence, such data races are harmless.

It should be noted that in an actual implementation (such as in Section 4), it is necessary to make σ a `volatile` variable. Only the coordinator thread modifies σ, while worker threads can 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.

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.

3.2. 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 and Aiken (2007) analyse callees in parallel before their callers are analysed, and other existing work (Helm et al., 2020) on parallel analysis has also confirmed the importance of prioritised scheduling to improve parallel efficiency. Multiple exploration heuristic to prioritise certain components can be expressed, and this is a question that we address at the end of this section.

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 (Shivers, 1991; Johnson et al., 2013a).

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).

3.3. 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 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.

3.4. Exploration heuristics

We mentioned that the order in which components are scheduled may impact performance, although it has no impact on the final analysis results. We now turn to the possible exploration heuristics that can be defined for both MODF and MODCONC. We therefore propose a number of heuristics for prioritising components. These heuristics are either inspired by related work (Xie and Aiken, 2007; Lyde and Might, 2015), or they aim at prioritising components based on the characteristics of their dependencies. We evaluate the impact of each of these heuristics on the running time of a MODF analysis in Section 5.4.

- **Rand**. This heuristic assigns a random priority to each component.
- **CD**. This heuristic implements an idea originally proposed by Xie and Aiken (2007): it assigns a higher priority to callees than to callers in the case of MODF. To that end, it tracks which component calls which other components, and the priority of a component is its call depth. This heuristic is also applicable to MODCONC, by assigning a higher priority to "creator" threads than created threads.
- **V-**. This heuristic assigns a higher priority to components that have been visited the least so far. The idea is that components that have been visited the least will contribute the most to the analysis state, and this might result in a faster convergence.
- **V+**. This heuristic assigns a higher priority to components that have been visited the most. It is therefore dual to the previous one, and might result in a slower convergence.
- **Ex+**. This heuristic assigns a higher priority to components that contain expressions that are deeper in the AST. Lyde and Might (Lyde and Might, 2015) have shown that this has a positive impact on the convergence speed of context-sensitive state-based analyses.
- **Ex-**. This heuristic is the opposite of the previous one: it assigns a higher priority to components that contain expressions that are higher in the AST.
- **D+**. This heuristic assigns a higher priority to components that have the most dependencies. Components with more dependencies will be re-triggered more often.
- **D-**. This heuristic assigns a higher priority to components that have fewer dependencies, and is the opposite of the previous one. Components with fewer dependencies may be triggered less often, and scheduling them before other components may result in fewer reanalyses.
- **DR**. This heuristic assigns a priority as $T/(T+R)$ where R is the number of dependencies of the component, and T is the number of times the component triggers dependencies. This prioritises components that have a higher ratio of triggering dependencies, with the aim of discovering components that will need to be reanalysed more quickly.
- **E+** This heuristic assigns a priority based on the size of the environment of a component, as a proxy for the size of the actual component. The advantage of measuring component size through environments is that a component may span across many lines of code, yet only have few reachable addresses. Hence, the impact of such a component on the state space will remain small. The idea is that larger components may have a larger impact on the global analysis state, therefore triggering dependencies which lead to the analysis state converging more quickly.

- **E-** This heuristic is the opposite of the previous one: components with a smaller environment are prioritised. The rationale is that components that have a smaller environment may stabilise faster than components with larger environments.

4. Implementation

We have implemented the parallel worklist algorithm presented in this paper for the MAF static analysis framework (Van Es et al., 2020a) and made it available online.[7] In this section, we detail our Scala implementation of the algorithm in order to illustrate the technicalities of implementing such an algorithm in a real-world analysis framework. This section can be skipped for the reader not interested in reproducing such an implementation.

MAF is a framework for implementing modular analyses for dynamic, higher-order languages. The framework provides a set of reusable building blocks to define a modular analysis as a composition of Scala traits. We defined the following trait for the parallel worklist algorithm:

```
trait ParallelWorklistAlgorithm[Expr <: Expression]
  extends ModAnalysis[Expr]
     with GlobalStore[Expr]
     with PriorityQueueWorklistAlgorithm[Expr] {
    ...
}
```

This trait extends `ModAnalysis`, which is the root class for all modular analyses in MAF. In addition, it extends the following two traits:

- `GlobalStore`, which expresses the requirement that a parallel analysis uses a global store. The parallel worklist algorithm can easily be generalised to support other kinds of global analysis state in addition to a global store. We only require such global state to form a join semi-lattice and to grow monotonically during the analysis, through the joining of results computed by intra-component analyses.
- `PriorityQueueWorklistAlgorithm`, which represents worklist algorithms that prioritise the analysis of certain components (i.e., using a priority queue) based on their priority. The definition of a component's priority is left open as a parameter, allowing us to plug in different heuristics for the exploration order of components.

We now cover the implementation of this trait before showing how it can be used to render a modular analysis parallel.

4.1. Intra-component analysis

The intra-component analysis performs the analysis of a single component. Afterwards, the computed result is used to update the global analysis state, which is composed of the global store, a timestamp for every dependency (as described in Section 3.2), a map of dependencies to the set of components that depend on it, and the set of visited components:

```
1  type GlobalState = (Map[Addr, Value], // <- store
2                      Map[Dependency, Int], // <- depVersion
3                      Map[Dependency, Set[Component]], // <- deps
4                      Set[Component] // <- visited
5  )
6  @volatile var latest: GlobalState = _
7
8  // keep track for every dependency of its "version number"
9  var depVersion = Map[Dependency, Int]().withDefaultValue(0)
```

[7] https://github.com/softwarelanguageslab/maf/tree/parallel-evaluation.

The *latest* global state is stored in a volatile variable `latest`. This is necessary because all workers should retrieve the global state when the intra-component analysis is performed, rather than when the component is scheduled for analysis, and because accesses to the global state need to be made in one atomic read operation to avoid race conditions when accessed from different threads.

The intra-component analysis then simply overrides helper functions from the intra-component analysis defined within the `GlobalStore` trait. Upon a write to an address in the global store (line 6 in the code below), the timestamp of the dependency is increased if the value written changed something in the store. Upon the registration of a dependency (line 13), the dependency is marked for inspection, and if it is the first time that the dependency is registered, it is added to the dependencies of the current component. Upon the creation of a new component (line 19), the new component is added to the set of components created, in case it has not yet been visited. Finally, the intra-component analysis is only considered finished (line 24) when, after analysing the component, all dependencies marked for inspection (i.e., those that were read during the intra-analysis) have the same timestamp locally (in `intra.depVersion`) as in the global analysis state (in `inter.depVersion`). Otherwise, the computed result is not up-to-date with respect to the latest global analysis state, and the component will need to be analysed again using the latest global analysis state.

```
1  def intraAnalysis(component: Component): ParallelIntra with
   GlobalStoreIntra
2  trait ParallelIntra extends IntraAnalysis with GlobalStoreIntra
   { intra =>
3    val (latestStore, depVersion, deps, visited) = latest
4    store = latestStore
5    var toCheck = Set[Dependency]()
6    override def doWrite(dep: Dependency): Boolean =
7      if (super.doWrite(dep)) {
8        inter.depVersion += dep -> (inter.depVersion(dep) + 1)
9        true
10     } else {
11       false
12     }
13   override def register(dep: Dependency): Unit = {
14     toCheck += dep
15     if (!deps(dep)(component)) {
16       R += dep
17     }
18   }
19   override def spawn(cmp: Component): Unit =
20     if (!visited(cmp)) {
21       C += cmp
22     }
23     def isDone = toCheck.forall(dep =>
24       inter.depVersion(dep) == intra.depVersion(dep))
25 }
```

4.2. Worker thread

The code below concerns the worker threads. The number of workers used by the analysis, which determines how much parallelism can be exploited, is defined in the `workers` variable (line 1). The variable `currentTimeout` (line 2) is the global timeout of the analysis: in case it is exceeded, all workers have to terminate. The use of the (global) worklist in worker threads is protected by a monitor (line 4). A worker is a regular thread (line 14), which continuously takes the next component from the worklist, analyses it and then sends back the result (using `pushResult`). The helper function `spawnWorker` (line 29) simply creates a new worker and starts it.

```
def workers: Int = Runtime.getRuntime.availableProcessors()
var currentTimeout: Timeout.T = _

object WorkListMonitor
def popWorklist(): Component = WorkListMonitor.synchronized {
  while (worklist.isEmpty)
    WorkListMonitor.wait()
  worklist.dequeue()
}
def pushWorklist(cmp: Component) = WorkListMonitor.synchronized {
  worklist += cmp
  WorkListMonitor.notify()
}
class Worker(i: Int) extends Thread(s"worker-thread-$i") {
  override def run(): Unit = try while (true) {
    val cmp = popWorklist()
    val intra = intraAnalysis(cmp)
    intra.analyzeWithTimeout(currentTimeout)
    if (currentTimeout.reached) {
      pushResult(TimedOut(cmp))
    } else {
      pushResult(Completed(intra))
    }
  } catch {
    case _: InterruptedException => ()
  }
}

def spawnWorker(i: Int) = {
  val worker = new Worker(i)
  worker.start()
  worker
}
```

4.3. Analysis results

The `Result` trait (line 1) represents the result of an intra-component analysis, which can either have completed successfully or have timed out. A successful analysis result contains the corresponding `ParallelIntra` object. Results are stored in a priority queue (line 8), which is protected by a monitor (line 7) as it will be used by multiple workers concurrently through the `pushResult` function. The `popResult` function will be used by the coordinator to extract analysis results.

```
sealed trait Result { def cmp: Component }
case class Completed(intra: ParallelIntra) extends Result {
  def cmp = intra.component
}
case class TimedOut(cmp: Component) extends Result

object ResultsMonitor
val results: PriorityQueue[Result] = PriorityQueue.empty

def popResult(): Result = ResultsMonitor.synchronized {
  while (results.isEmpty) ResultsMonitor.wait()
  results.dequeue()
}
def pushResult(res: Result) = ResultsMonitor.synchronized {
  results += res
  ResultsMonitor.notify()
}
```

4.4. Analysis coordination

The coordination of the worker threads relies on three elements: a list of worker threads that have been spawned (line 1), the set of components that need to be analysed (line 2), and the set of components that are currently queued in the worklist (line 3).

```
var workerThreads: List[Worker] = Nil
var todo: Set[Component] = Set(initialComponent)
var queued: Set[Component] = Set.empty
```

The `addToWorkList` function inherited by the `ModAnalysis` class is implemented (line 1) to only push a component on the worklist if it has not already been queued.

```
override def addToWorkList(cmp: Component): Unit =
  if (!queued.contains(cmp)) {
    queued += cmp
    pushWorklist(cmp)
  }
```

Two helper functions are defined to deal with the result of intra-component analyses. The `processTimeout` method is called when an intra-component analysis reaches its timeout: in this case, the component is removed from the set of queued components and will not be scheduled for analysis at that moment, but it is added back to the `todo` set to track that it has not been successfully analysed yet. The `processTerminated` method is called upon successful termination of an intra-component analysis. It first calls the `commit` method of the intra-component analysis (line 6), which will schedule newly created components and trigger dependencies to reschedule components that may depend on dependencies modified by the current intra-component analysis. Then, the global analysis state is updated (line 7). If the intra-component analysis does not need to be rescheduled due to its state being out of sync with the latest global state, the corresponding component is removed from the queued components (line 9), otherwise it is pushed on the worklist (line 11) as it needs to be analysed again to account for a newer version of the global analysis state.

```
private def processTimeout(cmp: Component): Unit = {
  todo += cmp
  queued -= cmp
}
private def processTerminated(intra: ParallelIntra): Unit = {
  intra.commit()
  latest = (store, depVersion, deps, visited)
  if (intra.isDone) {
    queued -= intra.component
  } else {
    pushWorklist(intra.component)
  }
}
```

The main method of an analysis is the `analyzeWithTimeout` method. It first initialises the timeout and the global state (lines 4 and 5), spawns all worker threads (line 7) and populates the worklist (line 9). As long as there are components queued for analysis (line 12), the next result is extracted (line 13), which possibly waits for the termination of an intra-component analysis, and calls the appropriate helper method depending on whether the analysis succeeded or timed out. Finally, once there are no more components to analyse, all worker threads are terminated (line 18).

```
override def analyzeWithTimeout(timeout: Timeout.T): Unit =
  if (!finished()) {
    // initialize timeout and initial analysis state
    currentTimeout = timeout
    latest = (store, depVersion, deps, visited)
    // spawn the workers
    workerThreads = List.tabulate(this.workers)(spawnWorker)
    // fill the worklist with initial items
    todo.foreach(addToWorkList)
    todo = Set.empty
    // main workflow: continuously commit analysis results
    while (queued.nonEmpty)
      popResult() match {
        case Completed(intra) => processTerminated(intra)
        case TimedOut(cmp)    => processTimeout(cmp)
      }
    // wait for all workers to finish
    workerThreads.foreach { t =>
      t.interrupt()
      t.join()
    }
  }
```

4.5. Prioritisation heuristics

As explained previously, the order in which the components are scheduled for analysis may have an impact on the convergence speed of the analysis. In order to evaluate this impact in the case of the parallel worklist algorithm, we have implemented all heuristics described in Section 3.4. Each heuristic is defined by implementing an ordering on components (`Ordering[Component]`), which will be used by the `PriorityQueueWorklistAlgorithm`.

Note that with the exception of the **Ex+** and **Ex-** heuristics, the priorities of components cannot be computed before the analysis is run. Therefore, these heuristics update the priorities as more components and dependencies are discovered, registered, and triggered.

4.6. Rendering an analysis parallel

Any `ModAnalysis` in MAF can be rendered parallel with this worklist algorithm. As an example, we illustrate how one can render a *k*-CFA analysis parallel. The following code shows how a sequential analysis is instantiated, based on the traits already provided by MAF, such as the main trait for an analysis of Scheme programmes (`SimpleSchemeModFAnalysis`), the context sensitivity (`SchemeModFKCallSiteSensitivity`), the abstract domain (`SchemeConstantPropagationDomain`), and the worklist algorithm (`RandomWorklistAlgorithm`).

```
def kCFAAnalysis(prg: SchemeExp, kcfa: Int) =
  new ModAnalysis(prg)
    with SimpleSchemeModFAnalysis
    with RandomWorklistAlgorithm[SchemeExp]
    with SchemeModFKCallSiteSensitivity
    with SchemeConstantPropagationDomain {
    val k = kcfa
  }
```

To instantiate a parallel version of this analysis, only a few lines have to be adapted, as shown in the next snippet. Rather than the sequential worklist algorithm, we rely on the parallel one (`ParallelWorklistAlgorithm`), which is instantiated with a random priority heuristic (`RandomPriorityWorklistAlgorithm`). The number of workers is defined, and the `intraAnalysis` method has to be redefined to mix in the intra-component analysis provided by the parallel worklist algorithm.

```
def parallelKCFAAnalysis(prg: SchemeExp, n: Int, kcfa: Int) =
  new ModAnalysis(prg)
    with SimpleSchemeModFAnalysis
    with RandomPriorityWorklistAlgorithm[SchemeExp]
    with ParallelWorklistAlgorithm[SchemeExp]
    with SchemeModFKCallSiteSensitivity
    with SchemeConstantPropagationDomain {
    val k = kcfa
    override def workers = n
    override def intraAnalysis(cmp: Component) =
      new IntraAnalysis(cmp) with BigStepModFIntra with
    ParallelIntra
  }
```

5. Evaluation

Our parallel worklist algorithm, including the optimisations discussed in Section 3.2, has been integrated into the MAF framework (Van Es et al., 2020a) as described in the previous section, where it is made available as a Scala trait that can directly be used with any existing modular analysis. We set up our experiments using this implementation and answer the three following research questions as part of our evaluation.

- **RQ1**: Does the proposed approach result in speedups when applied to the MODF modular analysis, and this for all context-insensitive and context-sensitive configurations of the analysis?
- **RQ2**: Does the proposed approach result in speedups when applied to a different modular analysis such as MODCONC?
- **RQ3**: What is the impact of the various exploration heuristics discussed in the previous section on the speedups?

We cover each of these research questions in the remainder of this section, after describing our evaluation setup. We have also verified the correctness of the parallel implementation by running the built-in soundness tests of MAF and by verifying that the analysis results are the same as those of the sequential implementation for all benchmarks. Our implementation, the setup for our experiments, and our benchmark programmes are available in a replication package.[8]

5.1. Setup

Execution environment. All experiments were conducted on a server using a AMD Ryzen Threadripper 3990X processor with 64 cores at 2.9 GHz, enabling running 128 threads simultaneously in total. The server uses Java 14.0.2 (OpenJDK) and Scala 2.13.3, and we configured the JVM with a fixed heap size of 64 GB. Each experiment was preceded by up to 10 warm-up runs, with a total warm-up timeout of 1 min. The reported results are the mean over 20 measurements, and we report on the standard deviation as well.

Benchmarks. We use a total of 40 Scheme benchmarks from the built-in benchmark suite of MAF, 24 for MODF (cf. Table 1) and 16 for MODCONC (cf. Table 2). The MODCONC benchmarks are written in an extended version of R5RS Scheme that includes special forms and primitives to support concurrency. In contrast to some of the existing work on parallel analysis (Dewey et al., 2015), our approach is not dependent on the choice of context-sensitivity: for MODF, we perform an evaluation on a context-insensitive configuration of the analysis as well two context-sensitive ones, using 1-CFA and 2-CFA (Shivers, 1991). For MODCONC, we run all analyses with high context-sensitivity (5-CFA), since otherwise they terminated too quickly to produce relevant results. For our evaluation, the abstract values come from a constant propagation domain (Cousot and Cousot, 1977), allowing the analysis to derive the type and, if possible, the constant value of every expression in the programme. Values such as functions and pointers are themselves approximated by a set-based lattice.

Sequential implementation. As advocated by Dewey et al. (2015), we report speedups relative to the sequential implementation (which is not always the case for speedups reported in other related work Lee and Ryder, 1992; Méndez-Lojo et al., 2010; Monniaux, 2005; Weeks et al., 1994). This is done for three reasons. First, our parallel algorithm has no sequential baseline: its instantiation with a single worker uses two threads (one for the worker, one for the coordinator), hence this would skew the results from an initial speedup that would not be reported in the results. Second, the work of Dewey et al. (2015) being the most closely related to ours, this enables comparing results. Finally, reporting speedups in such a way corresponds to the usual notion of speedups and leads to a more natural interpretation of the results: achieving a $n\times$ speedup means that replacing the sequential implementation of an analyser by the parallel one results in the analysis being $n\times$ faster.

For a fair comparison, unless specified otherwise, both the sequential and the parallel implementations use a heuristic that

[8] https://github.com/softwarelanguageslab/maf/tree/parallel-evaluation.

Table 1
An overview of the ModF benchmarks. **LOC** indicates lines of code (as calculated by `cloc`). The **0-CFA**, **1-CFA**, **2-CFA** columns indicate the time it takes for the sequential ModF implementation to analyse a benchmark with the given context sensitivity. ∞ indicates that the analysis timed out after 60 min.

Benchmark	LOC	0-CFA	1-CFA	2-CFA
`scp`	2504	1 m 28 s ± 47 s	10 s ± 2 s	13 s ± 2 s
`scheme`	767	1 m 21 s ± 26 s	∞	∞
`eceval`	774	1 m 1 s ± 55 s	1 m 0 s ± 18 s	4 m 26 s ± 1 m 54 s
`sboyer`	632	31 s ± 1 m 0 s	18 s ± 43 s	24 s ± 7 s
`peval`	497	35 s ± 19 s	∞	∞
`multiple-dwelling`	402	17 s ± 11 s	∞	∞
`decompose`	412	16 s ± 10 s	∞	∞
`dynamic`	1520	12 s ± 4 s	∞	∞
`prime-sum-pair`	394	8 s ± 4 s	∞	∞
`ambeval`	379	8 s ± 4 s	∞	∞
`meta-circ`	374	6 s ± 2 s	2 m 1 s ± 35 s	∞
`boyer`	577	4 s ± 2 s	9 s ± 4 s	∞
`nboyer`	625	3 s ± 4 s	6 s ± 11 s	29 s ± 1 m 15 s
`SICP-compiler`	402	3 s ± 1 s	1 s ± 0 s	6 s ± 3 s
`compiler`	466	3 s ± 1 s	1 s ± 0 s	3 s ± 0 s
`ad`	604	1 s ± 0 s	0 ± 0 s	1 s ± 0 s
`leval`	334	1 s ± 0 s	3 s ± 1 s	23 s ± 8 s
`aeval`	300	1 s ± 0 s	2 s ± 1 s	1 m 47 s ± 29 s
`earley`	469	0 s ± 0 s	5 s ± 1 s	∞
`graphs`	483	0 s ± 0 s	0 s ± 0 s	12 m 31 s ± 1 m 26 s
`nbody`	1250	0 s ± 0 s	2 s ± 0 s	11 s ± 3 s
`matrix`	617	0 s ± 0 s	0 s ± 0 s	1 m 2 s ± 21 s
`browse`	161	0 s ± 0 s	2 s ± 1 s	∞
`regsim`	396	0 s ± 0 s	0 s ± 0 s	0 s ± 0 s

Table 2
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 5-CFA ModConc implementation.

Benchmark	LOC	Sequential (5-CFA)
`minimax`	96	5 m 40 s ± 6 s
`matmul`	111	2 m 58 s ± 4 s
`actors`	105	1 m 28 s ± 1 s
`stm`	130	57 s ± 1 s
`abp`	72	23 s ± 0 s
`msort`	38	13 s ± 2 s
`randomness`	26	7 s ± 0 s
`crypt`	163	5 s ± 0 s
`sieve`	51	3 s ± 0 s
`sudoku`	84	3 s ± 0 s
`life`	124	2 s ± 0 s
`nbody`	136	1 s ± 0 s
`pps`	69	1 s ± 0 s
`phild`	46	1 s ± 0 s
`pp`	33	0 s ± 0 s
`atoms`	45	0 s ± 0 s

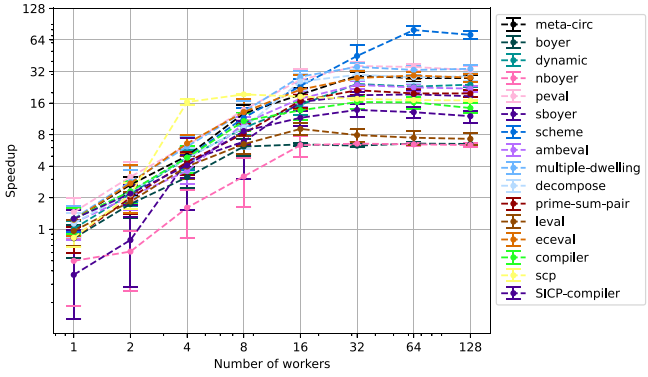

Fig. 3. Speedups for the context-insensitive ModF analysis.

randomly picks components from the worklist. The total running time of the sequential ModF and ModConc implementations are included in Tables 1 and 2, respectively. Using a random ordering for the exploration of components, together with the fact that we average our results over 20 runs of the analysis and report on the standard deviation, enables comparing the analyses' performance independently of the exploration order picked by one actual run of the analysis.

Although one could expect that increasing context sensitivity will lead to an increased analysis time because the theoretical complexity of the analysis is worsened, in practice we sometimes observe the opposite. For example, `sboyer` and `scp` see their analysis time reduced with a higher context sensitivity. This is an observation shared by other static analyses of both static and dynamic languages (Shapiro and Horwitz, 1997; Kashyap et al., 2014). In our case, we can track down this improvement to the use of *sets of abstract addresses* to encode data structures such as lists. Where an imprecise analysis will collate many abstract addresses into a set of abstract values, a more precise analysis will – thanks to its increased precision – not contain many of the superfluous abstract addresses in this set, therefore resulting in a reduced analysis time. In addition, when the data-flow information computed by the analysis impacts the control-flow behaviour, increased precision reduces the number of (spurious) behaviours that have to be analysed further.

5.2. RQ1: Speedups on context-insensitive and context-sensitive ModF analyses

The speedups for a context-insensitive ModF analysis are given in Fig. 3, and for context-sensitive ModF analyses, respectively 1-CFA and 2-CFA, in Figs. 4 and 5. These graphs only report speedups for benchmarks where the base analysis terminates within 60 min and takes more than one second. Each data point contains a vertical line indicating the standard deviation for that measurement.

In each case, we observe rather consistent speedups up to 16 workers: using 4 workers, most benchmarks achieve a speedup between 3× and 8×; using 16 workers, most benchmarks achieve a speedup between 8× and 32×. When using more than 16 workers, performance gains are mostly dependent on the benchmark under analysis.

In general, longer-running benchmarks such as `scheme` (0-CFA), `meta-circ` (1-CFA), `eceval` (2-CFA), and `graphs` (2-CFA)

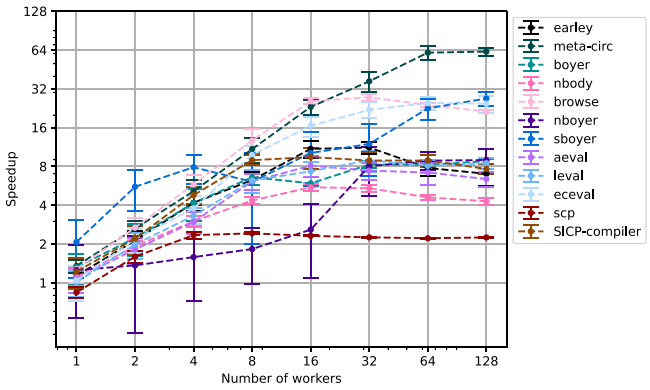

Fig. 4. Speedups for the context-sensitive 1-CFA ModF analysis.

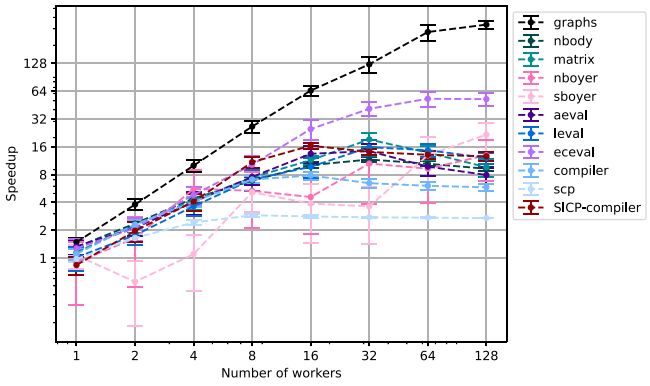

Fig. 5. Speedups for the context-sensitive 2-CFA ModF analysis.

can benefit more from the additional parallelism, achieving high speedups for 64 workers (80×, 60×, 53×, and 277× respectively). Other benchmarks improve only slightly (or not at all) when increasing the number of workers further. In particular, the `compiler` benchmark (2-CFA) appears to slow down when too many workers are added. One reason for this slowdown could be that the sequential implementation only takes 3 s to analyse this programme; using 8 workers reduces this to 0.4 s, 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 more limited 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, although with a reduced improvement as the number of workers grows.

The standard deviation of the speedups is particularly high for `sboyer` and `nboyer`. This is also something we observe in the performance of the sequential version of our algorithm, and indicates that the running time for these benchmarks are highly dependent on the exploration order of the analysis. However, we clearly observe for these benchmarks a speedup that increases with a standard deviation that decreases in all configurations of the analysis.

Interestingly, we observe some speedups that are better than ideal (greater than $N\times$ with N threads), which are sometimes called "superlinear" speedups in related work. First, it should 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 up to 2× when using just a single worker. Reporting speedups relative to the sequential implementation is advocated by Dewey et al. (2015), in order to adhere with the usual definition of speedup.

Another major factor contributing to such better than ideal speedups 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 3.1, 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 high speedups. For instance, the `scheme` benchmark (0-CFA) performs 53 489 intra-component analyses in total using the sequential implementation, whereas on a sample run of the parallel implementation with 8 workers, only 14 095 intra-component analyses are required. This means that for that benchmark, the intra-component analysis workload is reduced by approximately a factor of 4, which, combined with the parallelism of using 8 workers, explains why we are able to achieve a 40× speedup. Our work confirms findings reported in related work (Dewey et al., 2015; Albarghouthi et al., 2012; Edvinsson et al., 2011; Lee and Ryder, 1992) that often attributes better than ideal or unexpected speedups to the exploration order.

We can therefore answer our research question: the proposed approach does indeed result in impressive speedups when applied to a ModF analysis, often reaching a plateau after 64 workers. Some speedups are better than ideal due to the way the parallel algorithm exploits the monotonicity of the analysis to speed up its convergence. These results are similar across different context sensitivities. On average, our approach yields respectively a 15.4×, 10.6×, and 17.3× speedup on a 0-CFA, 1-CFA, and 2-CFA analysis with 16 threads.

5.3. RQ2: Speedups for a ModConc analysis

As discussed in Section 3.3, 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 Fig. 6, 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 $m * n$.

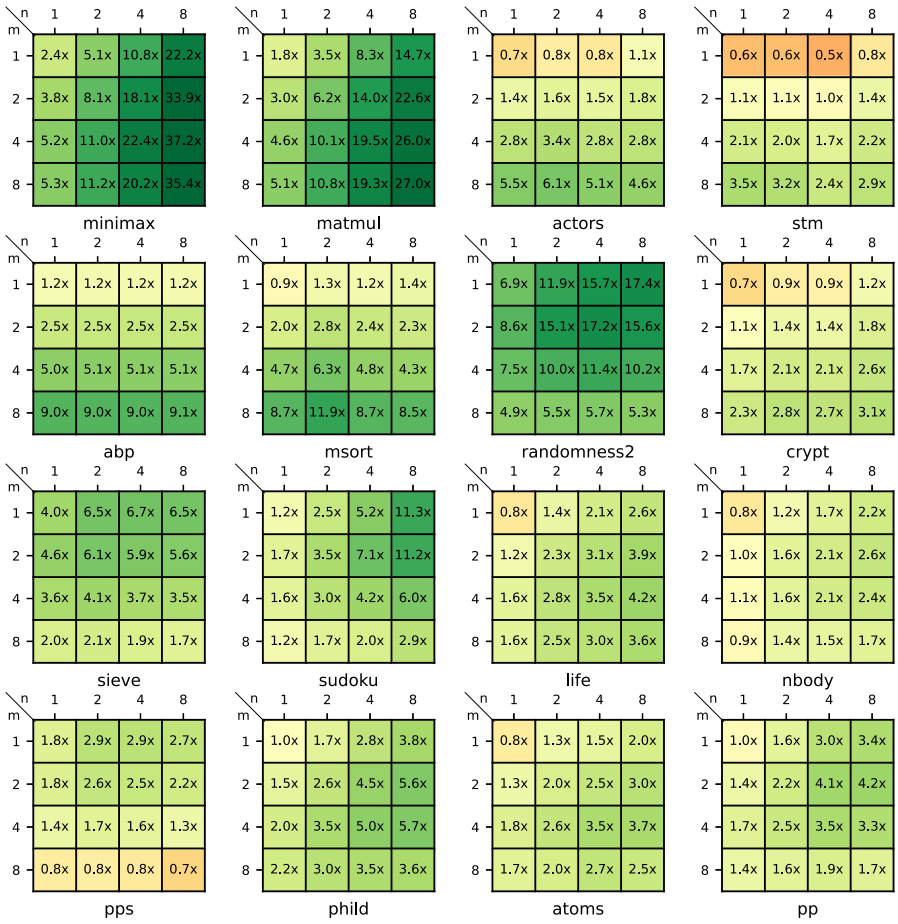

Fig. 6. Speedups for the context-sensitive 5-CFA ModConc analysis, relative to the sequential implementation. Horizontally (left-to-right), we increase the number of workers for ModConc inter-component analysis. Vertically (top-to-bottom), we increase the number of workers per intra-component (ModF) analysis. Colour is used to emphasise the magnitude of the speedup (or in some cases, slowdown). (For interpretation of the references to colour in this figure legend, the reader is referred to the web version of this article.)

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 `randomness`) clearly benefit from analysing multiple processes in parallel; others (such as `crypt`, `nbody`, or `sieve`) fail to achieve large speedups when using the same configuration, sometimes even slowing down compared to the sequential implementation. A reason for this could be 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 many 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 some programmes (e.g., `sudoku`, `life`, `nbody`, `pps`, `sieve`, `phild`, `pp`, `atoms`) do not leave much room for speedups, as their base analysis time is 2 s or less. This explains why the speedups observed for these programmes are lower. In contrast, longer-running benchmarks such as `matmul` and `minimax` achieve more significant speedups up to 27× and 37×, respectively.

We can therefore answer our research question: when applied to a ModConc analysis, our parallel worklist algorithm also results in decent speedups, although less impressive ones than when applied to a ModF analysis. This is mostly due to the more complicated orchestration of the analysis workers.

5.4. RQ3: Exploration strategies and their effects on speedups

We have introduced multiple exploration heuristics (Section 3.4), each of which could potentially result in further speedups. So far, our evaluation has only been concerned with the random exploration heuristic (**Rand**). To investigate the impact of the other heuristics on the speedups, we have run the ModF analyses with a fixed number of workers ($n = 4$), for various context sensitivities (0-CFA, 1-CFA, and 2-CFA), with each of the proposed heuristics. We report on the speedup of each heuristic compared to the time taken by the random exploration heuristic in the same configuration, in Tables 3 (0-CFA), 4 (1-CFA), and 5 (2-CFA). We only include results for configurations for which **Rand** takes at least 1 s and less than 60 min.

We notice that some heuristics only have a minor impact on performance: **Ex+**, **Ex-**, **V+**, **V-** achieve running times that are generally within 10% of the running time of an analysis using **Rand**.

Table 3
Speedups of various exploration heuristics relative to the random exploration heuristic for a parallel 0-CFA analysis with 4 workers. The background colour indicates whether the result is a speedup (green), or a slowdown (red).

Benchmark	Rand	CD	DR	D+	D-	Ex+	Ex-	V+	V-	E+	E-
scheme	18s	2.79	0.43	0.66	3.00	0.97	0.98	1.19	0.88	0.23	6.31
eceval	14s	2.95	0.92	0.34	1.51	0.78	1.11	1.32	0.75	0.23	7.92
scp	5s	1.07	1.01	0.80	1.31	1.01	1.01	0.98	0.99	0.88	1.14
peval	5s	1.72	0.48	0.72	2.76	1.06	0.76	0.96	1.10	0.63	1.21
sboyer	3s	0.27	5.17	0.21	5.73	1.11	0.95	0.99	0.97	0.21	5.29
decompose	3s	1.17	1.57	0.46	1.22	1.02	1.01	0.97	1.09	0.91	1.11
multiple-dwelling	3s	1.36	2.24	0.40	0.84	0.89	1.12	0.89	1.23	0.80	2.56
dynamic	2s	1.44	0.91	0.48	2.09	1.06	0.95	1.02	1.01	0.39	3.56
Prime-sum-pair	2s	1.12	2.17	0.45	0.77	1.05	0.97	1.14	0.90	0.88	2.80
ambeval	2s	0.65	2.72	0.74	0.74	0.98	1.09	0.88	1.07	0.70	3.20
nboyer	1s	0.63	3.50	0.28	4.46	1.13	1.01	0.94	1.07	0.11	8.27
meta-circ	1s	2.60	0.56	0.45	2.42	1.09	0.95	1.02	0.91	0.53	1.58
boyer	1s	0.61	1.72	0.77	1.37	0.95	1.05	1.00	0.98	1.27	0.98
compiler	1s	0.85	2.73	0.33	2.65	1.08	0.91	1.03	1.07	0.35	1.53
SICP-compiler	1s	0.90	2.33	0.33	2.84	0.98	0.91	0.99	1.10	0.41	1.52
ad	1s	1.03	1.01	0.98	1.01	1.02	1.00	0.99	1.02	0.95	1.02

Table 4
Speedups of various exploration heuristics relative to the random exploration heuristic for a parallel 1-CFA analysis with 4 workers. The background colour indicates whether the result is a speedup (green), or a slowdown (red).

Benchmark	Rand	CD	DR	D+	D-	Ex+	Ex-	V+	V-	E+	E-
meta-circ	22s	2.28	0.53	0.61	3.10	1.03	1.08	0.98	1.00	0.46	1.20
eceval	14s	0.96	2.47	0.30	1.55	1.09	1.02	1.09	1.13	0.48	2.48
nboyer	7s	0.57	9.93	0.33	4.17	1.02	0.94	1.04	0.94	0.20	6.51
sboyer	5s	0.25	13.87	0.12	6.89	1.08	0.95	0.94	0.76	0.28	10.38
scp	4s	0.86	1.01	0.84	1.28	0.99	1.02	0.99	1.02	0.87	1.05
boyer	2s	0.87	1.39	0.71	1.29	1.00	1.06	1.07	0.99	1.06	1.19
earley	1s	1.22	0.91	0.70	1.50	0.98	1.12	0.93	1.09	1.27	0.99
leval	1s	1.18	0.91	0.62	2.26	1.05	1.19	0.87	0.95	0.56	1.37
nbody	1s	1.32	1.07	0.52	1.81	1.07	1.02	1.00	0.95	0.71	0.84
aeval	1s	1.23	1.04	0.60	1.47	1.15	1.14	0.80	1.19	0.57	1.08

Table 5
Speedups of various exploration heuristics relative to the random exploration heuristic for a parallel 2-CFA analysis with 4 workers. The background colour indicates whether the result is a speedup (green), or a slowdown (red).

Benchmark	Rand	CD	DR	D+	D-	Ex+	Ex-	V+	V-	E+	E-
browse	5m31s	0.89	0.66	1.04	4.11	1.01	1.06	1.06	0.90	1.59	3.51
graphs	1m14s	0.89	1.57	0.69	2.65	0.95	0.97	0.91	1.14	1.16	0.38
eceval	56s	1.47	2.49	0.16	2.47	1.48	0.91	1.05	0.84	0.35	5.54
aeval	33s	2.12	1.14	0.34	1.90	0.95	1.03	0.98	1.04	0.62	1.02
sboyer	25s	0.55	19.63	0.40	0.97	1.60	0.65	1.42	0.85	0.75	3.74
matrix	17s	4.23	0.31	0.76	2.16	1.16	0.73	1.36	0.67	3.09	0.30
leval	7s	0.49	2.85	0.42	2.78	0.89	0.88	1.61	1.00	0.42	1.07
scp	6s	0.94	1.08	0.72	1.44	0.99	1.02	0.98	1.01	0.78	1.19
nbody	3s	1.53	1.40	0.30	2.35	1.10	1.01	0.86	1.11	0.60	0.85
nboyer	2s	0.09	15.55	0.56	1.45	0.74	1.18	1.00	0.96	0.88	2.27
SICP-compiler	1s	1.06	1.23	0.48	2.74	1.11	1.06	0.83	1.10	0.32	1.29
compiler	1s	0.45	2.25	0.87	1.17	1.08	1.13	0.83	1.11	0.58	1.02

Two heuristics (**E-** and **D-**) are often consistently better than the random heuristic, while their opposite heuristics (**E+** and **D+**) are consistently worse. On a context-insensitive analysis, **E-** and **D-** outperform **Rand** most of the time, or are only slightly slower. On context-sensitive analyses, **V-** consistently outperforms **Rand**, achieving an extra speedup of up to 6.89× (`sboyer`, 1-CFA), while **E-** is less consistent and can result in some slowdowns (e.g., 0.3× on `matrix`, 2-CFA), but also in better speedups (e.g., 10.38× on `sboyer`, 1-CFA).

The idea behind the **D-** and **E-** heuristics is to prioritise smaller components first, where for **D-** the number of dependencies is used as a proxy for component size, and for **E-** the size of the environment is used as a proxy for component size. Components that are larger will not only require more time to analyse due to their increased size, but may also be more frequently scheduled for reanalysis due to having more dependencies that may be triggered. Hence, prioritising smaller components enables the analysis to stabilise portions of its global state before analysing the larger components, thereby requiring less reanalyses of the larger components, and reducing the total analysis time. We have instrumented the 0-CFA analysis to log the number of dependencies of each analysed component, and we find a clear correlation between the speedups of **D-** and of **E-** with the mean number of dependencies of the components encountered during the analysis ($\rho = 0.466$, p-value $= 0.021$ for **D-**, and $\rho = 0.672$, p-value $= 0.0003$ for **E-**): the more dependencies the average component has, the higher the speedup obtained by these heuristics.

The other heuristics (**CD** and **DR**) do not perform consistently. **DR** is the heuristic that achieves the highest speedup (19.63× on `sboyer`, 2-CFA). However, for consistent results, we would recommend the use of either **D-** or **E-** as the default heuristic for context-insensitive analyses, and of **D-** for context-sensitive analyses.

5.5. 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. (2000).

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 programmes 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 text (Abelson and Sussman, 1996) and other well-known benchmark suites[9] for Scheme (Gabriel, 1985).

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 report the mean and standard deviation of our results, both for the sequential implementation and the speedups obtained by the our parallel algorithm. Although some programmes see a high standard deviation when analysed sequentially or with few workers, we observed that the standard deviation decreased with more workers, indicating more stable analysis times. Moreover, we have evaluated in RQ3 whether other exploration strategies lead to improvements in running time when compared to a random strategy, and also repeated measurements for this evaluation 20 times.

6. 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 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 or more workers.

A second improvement would be to optimise thread usage in the multi-level parallelism that we introduce in ModConc.

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

Currently, our approach requires specifying a number n of workers for the analysis of different threads, and a number m of workers for the analysis of different functions applications within a thread. Each of the n workers has to manage the creation, coordination, and termination of its m workers, which might incur a overhead. This could be avoided by designing an approach that solely requires one parameter, where workers can deal with the analysis of both threads and function applications, without having to manage other workers. As a result, one may reach speedups closer to the speedups we achieved for ModF.

7. Related work

Our approach falls within the domain of modular analyses, initially proposed by Cousot and Cousot (2002). In particular, we apply our approach to function-modular analyses (Nicolay et al., 2019) and process-modular analyses (Stiévenart et al., 2019). The sequential algorithm for the inter-component analysis (Algorithm 1) can be seen as an instantiation of Kildall's worklist algorithm (Kildall, 1973; Fecht and Seidl, 1999) for a system of equations where variables and their dependencies are discovered dynamically. In turn, our parallel variant of this algorithm (Algorithm 2) could also be formulated more generically for a general class of systems of equations. We now discuss the extensive existing work on the parallelisation of static programme analyses.

7.1. Parallelisation of classical dataflow problems

Classical dataflow problems have been parallelised by Lee and Ryder (1992), achieving a speedup of 7.5× on 8 cores, and by Kramer et al. (1994), computing an ideal achievable speedup of up to 5.4×. These are applicable when the control-flow graphs of the programme 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.

7.2. Parallelisation with a static call graph

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

C analysers. Monniaux (2005) described the parallel implementation of the Astrée static analyser (Cousot et al., 2005), achieving a speedup of around 2× on 5 cores. However, Cousot et al. (2009) observed that beyond 4 cores, "the cost of synchronisation out-weights the speedup of parallel execution" for Astrée. The Saturn programme analysis framework (Xie and Aiken, 2007) achieved a high parallelisation by performing bottom-up modular analysis, with speedups of up to 29× on 80 cores. McPeak et al. (2013) 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× on 8 cores. Recently, Kim et al. (2020) have revisited Bourdoncle's algorithm (Bourdoncle, 1993) with parallelisation, achieving a speedup of up to 11× 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. (2012) parallelised a top-down interprocedural modular analysis by relying on MapReduce-style parallelism, achieving a speedup of up to 7.4× 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 (2011) presented a parallelisation of IFDS (Reps et al., 1995) 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× on 8 cores. Edvinsson et al. (2011) parallelised *independent nodes* (e.g., independent control-flow branches or targets with different context-sensitivities), achieving a speedup of 4.4× on 8 cores. Again, in our approach we do not require control-flow and call graphs to be known before the analysis.

7.3. 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. (1994) 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× on 16 threads.

More recently, Dewey et al. (2015) parallelised an abstract interpreter for JavaScript, by partitioning states per context. The speedups they report range from 2× to 4× on 12 threads, with a few outliers up to 37×. In contrast, our approach does not need to partition components per context, and is therefore also applicable to context-insensitive analyses.

7.4. 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.

Méndez-Lojo et al. (2010) implemented parallel inclusion-based points-to analyses on GPUs (Méndez-Lojo et al., 2012), achieving a speedup of up to 3× on 8 cores, and of 7× on a GPU. Similarly, Prabhu et al. (2011) 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×.

Venet and Brat (2004) presented the first distributed static analysis implementation, focused on detecting out-of-bounds errors in embedded C programmes. It relies on a relational database to store the analysis data, achieving speedups up to 2.7× 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.

8. 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. We also define 11 exploration heuristics that prioritise the components based on their properties.

Our implementation in the MAF framework reveals significant speedups. For MODF, using 16 workers usually delivers speedups between 8× and 32×, with speedups up to 564×. On programmes that have small environments, our heuristic that prioritises components with small environments can lead to a further speedup that can reach 20×. For MODCONC, we observe speedups up to 37×. 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 programme 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.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

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

References

Abelson, H., Sussman, G.J., 1996. Structure and Interpretation of Computer Programs, Second Edition. MIT Press.

Albarghouthi, A., Kumar, R., Nori, A.V., Rajamani, S.K., 2012. Parallelizing top-down interprocedural analyses. In: ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI. pp. 217–228. http://dx.doi.org/10.1145/2254064.2254091.

Bourdoncle, F., 1993. Efficient chaotic iteration strategies with widenings. In: Formal Methods in Programming and their Applications, International Conference. pp. 128–141. http://dx.doi.org/10.1007/BFb0039704.

Christakis, M., Bird, C., 2016. What developers want and need from program analysis: an empirical study. In: Lo, D., Apel, S., Khurshid, S. (Eds.), Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016. ACM, pp. 332–343. http://dx.doi.org/10.1145/2970276.2970347.

Cousot, P., Cousot, R., 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Graham, R.M., Harrison, M.A., Sethi, R. (Eds.), Conference Record of the Fourth ACM Symposium on Principles of Programming Languages. ACM, pp. 238–252. http://dx.doi.org/10.1145/512950.512973.

Cousot, P., Cousot, R., 2002. Modular static program analysis. In: Horspool, R.N. (Ed.), Compiler Construction, 11th International Conference, CC 2002, Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002. In: Lecture Notes in Computer Science, vol. 2304, Springer, pp. 159–178. http://dx.doi.org/10.1007/3-540-45937-5_13.

Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X., 2005. The ASTREÉ analyzer. In: Programming Languages and Systems, 14th European Symposium on Programming, ESOP 2005.. pp. 21–30. http://dx.doi.org/10.1007/978-3-540-31987-0_3.

Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Rival, X., 2009. Why does Astrée scale up? Formal Methods Syst. Des. 35 (3), 229–264. http://dx.doi.org/10.1007/s10703-009-0089-6.

Dewey, K., Kashyap, V., Hardekopf, B., 2015. A parallel abstract interpreter for JavaScript. In: Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO. pp. 34–45. http://dx.doi.org/10.1109/CGO.2015.7054185.

Edvinsson, M., Lundberg, J., Löwe, W., 2011. Parallel points-to analysis for multicore machines. In: High Performance Embedded Architectures and Compilers, 6th International Conference, HiPEAC. pp. 45–54. http://dx.doi.org/10.1145/1944862.1944872.

Fecht, C., Seidl, H., 1999. A faster solver for general systems of equations. Sci. Comput. Program. 35 (2–3), 137–161.

Gabriel, R.P., 1985. Performance and Evaluation of LISP Systems, Computer Systems, Vol. 263. MIT Press, Cambridge, Mass..

Gilray, T., Adams, M.D., Might, M., 2018. Abstract allocation as a unified approach to polyvariance in control-flow analyses. J. Funct. Programming 28, e18. http://dx.doi.org/10.1017/S0956796818000138.

Helm, D., Kübler, F., Kölzer, J.T., Haller, P., Eichberg, M., Salvaneschi, G., Mezini, M., 2020. A programming model for semi-implicit parallelization of static analyses. In: Khurshid, S., Pasareanu, C.S. (Eds.), ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis. ACM, pp. 428–439. http://dx.doi.org/10.1145/3395363.3397367.

Johnson, J.I., Labich, N., Might, M., Van Horn, D., 2013a. Optimizing abstract abstract machines. In: Morrisett, G., Uustalu, T. (Eds.), ACM SIGPLAN International Conference on Functional Programming, ICFP'13. ACM, pp. 443–454. http://dx.doi.org/10.1145/2500365.2500604.

Johnson, B., Song, Y., Murphy-Hill, E.R., Bowdidge, R.W., 2013b. Why don't software developers use static analysis tools to find bugs? In: Notkin, D., Cheng, B.H.C., Pohl, K. (Eds.), 35th International Conference on Software Engineering, ICSE '13. IEEE Computer Society, pp. 672–681. http://dx.doi.org/10.1109/ICSE.2013.6606613.

Kashyap, V., Dewey, K., Kuefner, E.A., Wagner, J., Gibbons, K., Sarracino, J., Wiedermann, B., Hardekopf, B., 2014. JSAI: Designing a sound, configurable, and efficient static analyzer for javascript. CoRR abs/1403.3996. arXiv:1403.3996. URL http://arxiv.org/abs/1403.3996.

Kildall, G.A., 1973. A unified approach to global program optimization. In: Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. pp. 194–206.

Kim, S.K., Venet, A.J., Thakur, A.V., 2020. Deterministic parallel fixpoint computation. Proc. ACM Program. Lang. 4 (POPL), 14:1–14:33. http://dx.doi.org/10.1145/3371082.

Kramer, R., Gupta, R., Soffa, M.L., 1994. The combining DAG: A technique for parallel data flow analysis. IEEE Trans. Parallel Distrib. Syst. 5 (8), 805–813.

Lee, Y., Ryder, B.G., 1992. A comprehensive approach to parallel data flow analysis. In: Proceedings of the 6th International Conference on Supercomputing, ICS. pp. 236–247. http://dx.doi.org/10.1145/143369.143415.

Lyde, S., Might, M., 2015. State exploration choices in a small-step abstract interpreter. In: Scheme and Functional Programming Workshop.

McPeak, S., Gros, C., Ramanathan, M.K., 2013. 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. pp. 554–564. http://dx.doi.org/10.1145/2491411.2501854.

Méndez-Lojo, M., Burtscher, M., Pingali, K., 2012. 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. pp. 107–116. http://dx.doi.org/10.1145/2145816.2145831.

Méndez-Lojo, M., Mathew, A., Pingali, K., 2010. Parallel inclusion-based points-to analysis. In: Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA. pp. 428–443. http://dx.doi.org/10.1145/1869459.1869495.

Monniaux, D., 2005. The parallel implementation of the Astrée static analyzer. In: Programming Languages and Systems, Third Asian Symposium, APLAS 2005. pp. 86–96. http://dx.doi.org/10.1007/11575467_7.

Nicolay, J., Stiévenart, Q., De Meuter, W., De Roover, C., 2019. Effect-driven flow analysis. In: Enea, C., Piskac, R. (Eds.), Verification, Model Checking, and Abstract Interpretation - 20th International Conference, VMCAI 2019. In: Lecture Notes in Computer Science, vol. 11388, Springer, pp. 247–274. http://dx.doi.org/10.1007/978-3-030-11245-5_12.

Prabhu, T., Ramalingam, S., Might, M., Hall, M.W., 2011. EigenCFA: accelerating flow analysis with GPUs. In: Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL. pp. 511–522. http://dx.doi.org/10.1145/1926385.1926445.

Reps, T.W., Horwitz, S., Sagiv, S., 1995. Precise interprocedural dataflow analysis via graph reachability. In: Conference Record of POPL'95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. pp. 49–61. http://dx.doi.org/10.1145/199448.199462.

Rodriguez, J., Lhoták, O., 2011. Actor-based parallel dataflow analysis. In: Compiler Construction - 20th International Conference, CC 2011. pp. 179–197. http://dx.doi.org/10.1007/978-3-642-19861-8_11.

Sadowski, C., Aftandilian, E., Eagle, A., Miller-Cushon, L., Jaspan, C., 2018. Lessons from building static analysis tools at Google. Commun. ACM 61 (4), 58–66. http://dx.doi.org/10.1145/3188720.

Shapiro, M., Horwitz, S., 1997. The effects of the precision of pointer analysis. In: Hentenryck, P.V. (Ed.), Static Analysis, 4th International Symposium, SAS '97. In: Lecture Notes in Computer Science, vol. 1302, Springer, pp. 16–34. http://dx.doi.org/10.1007/BFb0032731.

Shivers, O., 1991. Control-Flow Analysis of Higher-Order Languages (Ph.D. thesis). Carnegie Mellon University.

Smaragdakis, Y., Bravenboer, M., Lhoták, O., 2011. Pick your contexts well: understanding object-sensitivity. In: Ball, T., Sagiv, M. (Eds.), Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011. ACM, pp. 17–30. http://dx.doi.org/10.1145/1926385.1926390.

Stiévenart, Q., Nicolay, J., De Meuter, W., De Roover, C., 2017. Mailbox abstractions for static analysis of actor programs. In: Müller, P. (Ed.), 31st European Conference on Object-Oriented Programming, ECOOP 2017. In: LIPIcs, vol. 74, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 25:1–25:30. http://dx.doi.org/10.4230/LIPIcs.ECOOP.2017.25.

Stiévenart, Q., Nicolay, J., De Meuter, W., De Roover, C., 2019. A general method for rendering static analyses for diverse concurrency models modular. J. Syst. Softw. 147, 17–45. http://dx.doi.org/10.1016/j.jss.2018.10.001.

Van Es, N., Van der Plas, J., Stiévenart, Q., De Roover, C., 2020a. 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. IEEE Computer Society.

Van Es, N., Stiévenart, Q., Van der Plas, J., De Roover, C., 2020b. A parallel worklist algorithm for modular analyses. In: 20th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2020. pp. 1–12. http://dx.doi.org/10.1109/SCAM51674.2020.00006.

Venet, A., Brat, G.P., 2004. 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. pp. 231–242. http://dx.doi.org/10.1145/996841.996869.

Weeks, S., Jagannathan, S., Philbin, J., 1994. A concurrent abstract interpreter. LISP Symb. Comput. 7 (2–3), 173–193.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., 2000. Experimentation in Software Engineering - An Introduction. In: The Kluwer International Series in Software Engineering, vol. 6, Kluwer, http://dx.doi.org/10.1007/978-1-4615-4625-2.

Xie, Y., Aiken, A., 2007. Saturn: A scalable framework for error detection using boolean satisfiability. ACM Trans. Program. Lang. Syst. 29 (3), 16. http://dx.doi.org/10.1145/1232420.1232423.

Quentin Stiévenart is a postdoctoral researcher at the Software Languages Lab of the Vrije Universiteit Brussel in Belgium. He obtained his Ph.D. from the Vrije Universiteit Brussel in 2018 with a thesis entitled "Scalable Designs for Interpretation of Concurrent Programs: Application to Actors and Shared-Memory Multi-Threading". His research focuses on programme analysis and concurrency with a focus on security.

Noah Van Es is a Ph.D. candidate at the Software Languages Lab (SOFT) of the Vrije Universiteit Brussel (VUB), where he obtained his M.Sc. degree in Computer Science in 2017. His research is centred around programme analyses, specifically on rendering analyses more efficient through the use of adaptive programme analysis.

Jens Van der Plas is a Ph.D candidate at the Software Languages Lab (SOFT) of the Vrije Universiteit Brussel (VUB), where he obtained his M.Sc. degree in Computer Science in 2019. His current research is focused on techniques for the incrementalisation of modular static analyses.

Coen De Roover is an associate professor at the Software Languages Lab (SOFT) of the Vrije Universiteit Brussel (VUB). The central theme of his research is the design of programme analyses, and their application to problems in software quality. He has published over 90 peer-reviewed articles in the domain, and he is actively involved in collaborative research projects of a fundamental, strategic, or applied nature. He frequently serves on the programme committee for conferences such as ASE, MSR, ICSME, SANER, and SCAM.