Flow-Sensitive Composition of Thread-Modular Abstract Interpretation

Markus Kusano
Virginia Tech
Blacksburg, VA, USA

Chao Wang
University of Southern California
Los Angeles, CA, USA

ABSTRACT

We propose a constraint-based flow-sensitive static analysis for concurrent programs by iteratively composing thread-modular abstract interpreters via the use of a system of lightweight constraints. Our method is compositional in that it first applies sequential abstract interpreters to individual threads and then composes their results. It is flow-sensitive in that the causality ordering of interferences (flow of data from global writes to reads) is modeled by a system of constraints. These interference constraints are lightweight since they only refer to the execution order of program statements as opposed to their numerical properties: they can be decided efficiently using an off-the-shelf Datalog engine. Our new method has the advantage of being more accurate than existing, flow-insensitive, static analyzers while remaining scalable and providing the expected soundness and termination guarantees even for programs with unbounded data. We implemented our method and evaluated it on a large number of benchmarks, demonstrating its effectiveness at increasing the accuracy of thread-modular abstract interpretation.

CCS Concepts

•**Software and its engineering** → *Automated static analysis; Formal software verification;*

Keywords

Concurrency, Abstract interpretation, Invariant generation, Thread-modular reasoning, Interference, Datalog

1. INTRODUCTION

Although abstract interpretation [2] has wide use in the analysis and verification of sequential programs, designing a scalable abstract-interpretation-based analysis for shared-memory concurrent programs remains a difficult task [5, 8, 20–22]. Due to the large concurrent state space, directly applying techniques designed for sequential abstract interpretation to interleaved executions of a concurrent program does not scale. In contrast, recent thread-modular techniques [8, 20–22] drastically *over-approximate* the interactions between threads, allowing a more tractable but less accurate analysis. Their main advantage is that sequential abstract interpreters can be lifted to concurrent ones with minimal effort. However, they consider thread interactions in a *flow-insensitive* manner: given a system of threads $\{A, B, C\}$, for instance, they assume A can observe all combinations of memory modifications from B and C despite that some of these combinations are infeasible, thereby leading to a large number of false alarms even for simple programs.

FSE'16, November 13–18, 2016, Seattle, WA, USA
© 2016 ACM. 978-1-4503-4218-6/16/11...
http://dx.doi.org/10.1145/2950290.2950291

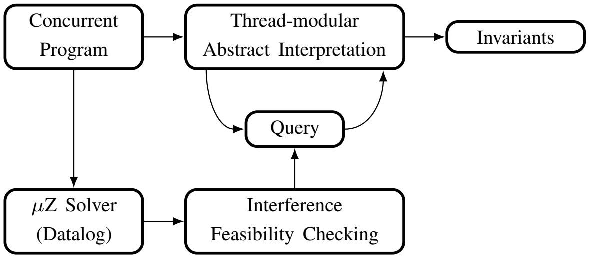
Concurrent Program
Thread-modular Abstract Interpretation
Invariants
Query
μZ Solver (Datalog)
Interference Feasibility Checking

Figure 1: WATTS: Flow-sensitive thread-modular analysis.

In this paper, we propose the first constraint-based flow-sensitive method for composing sequential abstract interpreters to form a more accurate thread-modular analysis. Though desirable, no existing static method is able to maintain inter-thread flow sensitivity with a reasonable cost. The main advantage of our method is that, through the use of a lightweight system of constraints, it can achieve a high degree of flow sensitivity with negligible runtime cost. Here, our goal is to prove the correctness of reachability properties of a program: the properties are embedded assertion statements whose error conditions are relational expressions over program variables at specific thread locations. Another advantage is that our method can be implemented as a flexible composition of existing sequential abstract-interpretation frameworks while retaining the well-known benefits such as soundness and guaranteed termination as well as the freedom to plug in a large number of abstract domains [2, 19].

Figure 1 shows an overview of our new method. Given a concurrent program, our method returns a set of relational and numerical invariants statically computed at each thread location as output. These invariants, in turn, can be used to prove the set of reachability properties of the program. During the thread-modular analysis, we first apply a sequential abstract interpreter to each individual thread and then propagate their results across threads before applying these sequential abstract interpreters again. The iterative process continues until a fix-point is reached over the set of invariants. During each iteration, the abstract interpreter also communicates with a Datalog engine to check if a thread interference, or set of interferences (data flow from global writes to reads), is feasible. If we can statically prove that the interference is infeasible, i.e., it cannot occur in any real execution of the program, we skip it, thereby reducing the analysis time and increasing accuracy.

```
1   bool flag = false;
2   int x = 0;
3   void thread1() {
4     x = 4;
5     x = 5;
6     flag = true;
7   }
```

```
8   void thread2() {
9     bool b1 = flag;
10    if (b1) {
11      int t1 = x;
12      if (t1 != 5)
13        ERROR!;
14  } }
```

Figure 2: Proving the `ERROR!` on l_{13} is not reachable.

In contrast to existing methods in this domain, our analysis is *flow-sensitive* for two reasons. First, we explore the memory interactions between threads individually by propagating their memory-states along data-flow edges without eagerly merging them through join operations as in prior techniques [8,20–22]. Second, we identify and remove the infeasible memory interactions by constructing and solving a system of *lightweight* happens-before constraints. These constraints (Horn clauses in finite domains) capture only the causality ordering of the program's statements as opposed to complex relational/numerical properties. As such, they can be solved by a Datalog engine in polynomial time. These two techniques, together, greatly reduce the number of false alarms caused by over-approximating the global memory state across threads, thereby allowing more properties to be verified compared to prior approaches.

Consider the program in Figure 2, which has two threads communicating through the shared variables `x` and `flag`. Initially `flag` is false and `x` is 0. Thread 1 only performs shared memory writes by setting `x` to 4, and then to 5, before setting `flag` to true. Thread 2 only performs shared memory reads: it reads the value of `flag` and if the value is true, reads the value of `x`. Note that the `ERROR!` (at l_{13}) is unreachable since, for Thread 2 to reach l_{11}, Thread 1 has to set `flag` to true (at l_6) before l_9 is executed; but in such a case, l_5 must have been executed, meaning `x` must have been set to `5`.

Prior static analyzers such as Ferrara [8] and Miné [20–22] would have difficulty because their treatment of inter-thread communication is *flow-insensitive*. That is, if one thread writes to a shared variable at program location l_i and another thread reads the same shared variable at l_j, they model the interaction by adding a data-flow edge from l_i to l_j even if the edge is infeasible or is only feasible in some program executions. For example, in Figure 2, no concrete execution simultaneously allows the flow of `x` from l_4 to l_{11} and the flow of `flag` from l_6 to l_9. In such cases, these prior methods would lose accuracy because their way of modeling the inter-thread data flow cannot differentiate between the feasible and infeasible data-flow combinations.

In contrast, our new method detects and eliminates such infeasible data-flows. For now, it suffices to say that our method would report that the flow of `x` from l_4 to l_{11} *cannot co-exist* with the flow of `flag` from l_6 to l_9. We will provide full details of our constraint-based interference analysis in Section 5.

Our method for checking the feasibility of inter-thread data flows is sound: when it declares a certain combination of interferences as infeasible, the combination is guaranteed to be infeasible. However, for efficiency reasons, our method does not attempt to identify every infeasible combination. This is consistent with the fact that abstract interpretation, in the context of property verification, is generally an over-approximation: it can prove the *absence* of errors but does not aim to guarantee that all unverified properties have real violations. As such, the additional effort we put into our constraint-based interference analysis is a fair trade-off between lower runtime overhead and improved accuracy. This puts our method in a nice middle ground between the more heavyweight model checkers [1] and the more scalable and yet less accurate static analysis techniques [8,20–22].

Another perhaps subtle benefit of our method is that the sequential abstract interpreter only needs a lightweight constraint solver [11] as a black-box to query the feasibility of a set of interferences. As such, it provides a flexible and extensible framework, allowing additional constraints, deduction rules, and decision procedures (e.g., solvers for symbolic-numerical domains) to be plugged in to further reduce the number of false alarms. To make our method more efficient, we also propose several optimizations to our interference feasibility analysis (Section 6): we leverage control and data dependencies to group interferences before checking the feasibility of their combinations, and leverage property-directed pruning to reduce the program's state space.

Our method differs from the DUET concurrent static analyzer of Farzan and Kincaid [5, 6] despite that both methods employ constraint-based analysis, since we aim at solving a different problem. First, our goal is to accurately analyze a concurrent program with a fixed number of threads, whereas their goal is to soundly approximate the behavior of a parameterized program with an unbounded number of thread instances. Second, our method is strictly thread-modular: we iteratively apply a sequential abstract interpreter to a set of control-flow graphs, one per thread, and one at a time. In contrast, they analyze a single monolithic data-flow graph of the entire concurrent program. As a result, their method is significantly less accurate than ours on non-parameterized programs. We illustrate the main difference between these two computational models, i.e., a set of per-thread control-flow graphs versus a monolithic data-flow graph, in Section 2.3.

We implemented our method in a static analysis tool named WATTS, for verifying reachability properties of multithreaded C/C++ programs written using the POSIX thread library. The tool builds upon the LLVM compiler, using the μZ [11] fix-point engine in Z3 [4] to solve Datalog constraints and the Apron library [13] to implement the sequential abstract interpreter over numerical abstract domains. We have evaluated our method on a set of benchmarks with a total of 26,309 lines of code. Our experiments show that WATTS can successfully prove 1,078 reachability properties, compared to 526 properties proved by DUET [5, 6] and 38 properties proved by the prior, flow-insensitive methods [21, 22]. Furthermore, WATTS achieved a 28x increase in the number of verified properties with only a 1.4x increase in the analysis time.

In summary, this paper makes the following contributions:

1. We propose a flow-sensitive method for composing thread-modular abstract interpreters into a more accurate static analysis procedure.
2. We develop a lightweight constraint-based framework for soundly checking the feasibility of inter-thread interferences and combinations of interferences.
3. We develop optimization techniques to improve the efficiency of our analysis by leveraging control and data dependencies and property-directed pruning.
4. We implement and evaluate our method on a large set of benchmarks and demonstrate its advantages over prior works.

2. MOTIVATING EXAMPLES

We present a series of examples showing applications of our new method compared to existing approaches.

2.1 Thread-modular Abstract Interpretation

First, Figure 2 provides an overview of prior works on thread-modular abstract interpretation [8,20–22]. These methods all use the same notion of *interference* between threads: an interference is a value stored into shared memory at some point during the execution of a thread. In Figure 2, there are three interferences, all from Thread 1: the writes to `x` at l_4 and l_5 and the write to `flag` at l_6.

Table 1: Running prior approaches [8,20–22] on Figure 2.

	Thread 1		Thread 2	
Iteration	Reachable	Interference	Reachable	Interference
One	4,5,6	$\mathit{flag} = \{1\}$ $x = \{4, 5\}$	9,10	$\varnothing$
Two	4,5,6	$\mathit{flag} = \{1\}$ $x = \{4, 5\}$	9,10,11 12,13	$\varnothing$

These prior techniques analyze the program by statically computing the over-approximated set of interferences for each thread:

1. Initially, the set of interferences in each thread is empty.
2. Each thread is independently analyzed in the presence of interferences from all other threads.
3. The set of interferences in each thread is recomputed based on the results of the analysis in Step 2.
4. Steps 2–3 are repeated until the interferences stabilize.

During the thread-modular analysis (step 2), each thread keeps track of its own *memory environment* at every thread location. The memory environment is an abstract state mapping program variables to their values. To incorporate inter-thread effects, when a thread performs a shared memory read on some global variable `q`, it reads either the values of `q` in its own memory environment, or the values of `q` from the interferences of all other threads. These techniques rely on a flow-insensitive analysis in that each read may see *all* values ever written by any other thread, even if the flow of data is not feasible in all, or any, of the concrete program executions.

Table 1 shows the results of analyzing Figure 2 with prior thread-modular approaches [8,20–22]. Column 1 shows the two iterations. Columns 2 and 4 show the lines reachable after each iteration in the two threads. Columns 3 and 5 show the interferences generated after each iteration. In the second iteration, the interferences generated during the first iteration are visible: Thread 2 is analyzed in the presence of the interferences generated by Thread 1. After two iterations, the interferences stabilize, which concludes the analysis. Unfortunately, the result in Table 1 shows that Thread 2 can reach l_{12}, where it reads the value of `x` either from its own memory environment (the initial value `0`) or from the interference of Thread 1 (`4`), thereby allowing the `ERROR!` to be reached. This is a false alarm: the property violation is generated because the inter-thread interferences are handled in a flow-insensitive manner.

In this example, to eliminate the false alarm one has to maintain a complex invariant such as $(\mathit{flag} = \mathit{true}) \rightarrow (x = 5)$ which cannot be expressed precisely as a relational invariant even in expensive numerical domains such as convex polyhedra. Additionally, in order to propagate such a relational invariant across threads, as in [22], they need to hold over all states within a thread. Otherwise, interference propagation is inherently non-relational. Specifically, propagating the interferences on a variable `x` first requires a projection on `x`, thus forgetting all relational invariants.

In contrast, our method can eliminate the false alarm even while staying in inexpensive abstract domains such as intervals. In particular, our work shows that eagerly joining over all interference across threads is inaccurate and should be avoided as much as possible.

2.2 Iterative Flow-sensitive Analysis

We propose, instead, to partition the set of interferences from other threads into clusters and then consider combinations of interferences only within these clusters. In this way, we effectively delay the join of interferences and avoid the inaccuracies caused by eagerly joining in existing methods. For example, if we assume the three interferences in Figure 2 fall into one cluster (worst for efficiency but best for accuracy), our analysis of the program would be as follows: in the first iteration, we apply per thread abstract interpretation and then compute the interferences for each thread; these computations remain the same as in the first iteration of Table 1. In the second iteration, however, when analyzing Thread 2 at the point of reading `flag`, there will be six possible cases, due to the Cartesian product of $x = \{0, 4, 5\}$ and $\mathit{flag} = \{0, 1\}$.

Unlike prior approaches, which eagerly join these cases to form $x = \{0, 4, 5\} \wedge \mathit{flag} = \{1, 0\}$, we analyze the impact of each case ρ_1–ρ_6 *individually* as follows:

- ρ_1, corresponding to $(x = 4 \wedge \mathit{flag} = 0)$;
- ρ_2, corresponding to $(x = 5 \wedge \mathit{flag} = 0)$;
- ρ_3, corresponding to $(x = 0 \wedge \mathit{flag} = 0)$;
- ρ_4, corresponding to $(x = 5 \wedge \mathit{flag} = 1)$;
- ρ_5, corresponding to $(x = 4 \wedge \mathit{flag} = 1)$; and
- ρ_6, corresponding to $(x = 0 \wedge \mathit{flag} = 1)$.

This leads to enough accuracy to prove the `ERROR!` is not reachable. First, when $\mathit{flag} = 0$ (ρ_1, ρ_2, and ρ_3) the `ERROR!` cannot be reached since the branch at l_{10} will not be taken (`b1` is `false`). Second, in the case of ρ_4, the first branch at l_{10} will be taken but the branch guarding `ERROR!` will not, since $x = 5$, meaning `t1` is also 5. For the two remaining cases (ρ_5 and ρ_6) our constraint-based interference analysis (Section 5) would show it is impossible to have both $x = 4 \wedge \mathit{flag} = 1$, or $x = 0 \wedge \mathit{flag} = 1$.

The intuition behind the analysis is that infeasible data flows cause a contradiction between program-order constraints and data-flow edges. Specifically, examining ρ_5, if Line 9 reads `flag` as `1` and Line 11 reads `x` as `4`, then:

- Line 6 is executed before Line 9 (`b1==true`),
- Line 9 is executed before Line 11 (program order),
- Line 11 is executed before Line 5 (`t1==4`), and
- Line 5 is executed before Line 6 (program order).

This leads to a contradiction since the above must-happen-before relationship forms a cycle, meaning the combination cannot happen. Similarly, ρ_6 is infeasible since the write of 1 to `flag` implies the updates to `x` have already occurred, meaning `x`'s initial value, 0, is not visible to Thread 2. At this point, the only feasible interferences do not cause an `ERROR!` — the program is verified.

To obtain the aforementioned accuracy, we leverage the statically computed control and data dependencies to partition the set of interferences into clusters. This can significantly reduce the number of cases considered during our thread-modular analysis. For example, when a load of `y` is *independent* of the subsequent load of `x`, e.g., the value loaded from `y` has no effect on the load of `x`, the thread would have two unconnected subgraphs in its program dependence graph [7]. Unconnected subgraphs create a natural partition of loads into clusters, thereby significantly reducing the complexity of our interference-feasibility checking. This is because we only need to consider combinations of interferences *within* each subgraph. We will show details of this optimization in Section 6.

2.3 Control-flow versus Data-flow Graphs

Our method also differs from DUET, a concurrent static analyzer for parametric programs [5, 6]. Although DUET also employs a constraint-based analysis, its verification problem is significantly different. First, it is designed for soundly analyzing parameterized concurrent programs, where each thread routine may have an unbounded number of instances. In contrast, our method is designed to analyze programs with a fixed number of threads with the goal of obtaining more accurate analysis results.

Second, DUET relies on running an abstract interpreter over a single data-flow graph of the entire program, whereas our method relies on running abstract interpreters over a set of thread-local control-flow graphs. The difference between using a set of thread-local control-flow graphs and a single monolithic data-flow graph can be illustrated by the following two-threaded program: `{x++;} ||`

`{tmp=x;}`. In the monolithic data-flow graph representation [5,6], there would be cyclic data-flow edges between the read and write of `x` across threads as well as an edge from the write of `x` to itself. As a result, applying a standard abstract interpretation based analysis would lead to the inclusion of $tmp = \infty$ as a possible value, despite that in any concrete execution of the program, the end result is either $tmp = 1$ or $tmp = 0$ (assume that $x = 0$ initially). Our method, in contrast, can correctly handle this program.

3. BACKGROUND

We provide a brief review of abstract interpretation based static analysis for sequential and concurrent programs. For a thorough treatment, refer to Nielson and Nielson [23] and Miné [20–22].

3.1 Sequential Abstract Interpretation

An abstract interpretation based static analysis is a fix-point computation in some abstract domain over a program's *control-flow graph* (CFG). The CFG consists of nodes representing program statements and edges indicating transfer of control between nodes. Due to their one-to-one mapping we interchangeably use the term statement and node. We assume the graph has a unique entry.

The analysis is parameterized by an *abstract domain* defining the representation of *environments* in the program. An environment is an abstract memory state. The purpose of restricting the representation of memory states to an abstract domain is to reduce computational overhead and guarantee termination. For example, in the interval domain [2], each variable has an upper and lower bound. For a program with two variables `x` and `y`, an example environment is $x = [0, 5] \wedge y = [10, 20]$. With properly defined meet ($\sqcap$) and join ($\sqcup$) operators, a partial order ($\sqsupseteq$), as well as the top ($\top$) and bottom ($\bot$) elements, the set of all possible environments in the program forms a lattice. In the interval domain, for example, we have $[0, 5] \sqcup [10, 20] = [0, 20]$ and $[0, 5] \sqsupseteq [0, 2]$.

Each statement in the program is associated with a *transfer function*, taking an environment as input and returning a new environment as output. The transfer function of statement st for some input environment e returns a new environment e', which is the result of applying st in e. Consider the above example of interval domain for x and y again. The result of executing the statement `x=x+y` in the above example environment would be the new environment $x = [10, 25] \wedge y = [10, 20]$.

For brevity, we will not define all the transfer functions for a programming language explicitly since the main contributions of this work are language-agnostic. As an example, however, consider the statement `t=load x`, which copies a value from memory to a variable. Its transfer function can be represented as $\lambda e.e[t = x]$, where $e[st]$ is the result of evaluating st in the environment e. Conceptually, it takes an input environment and returns a new environment where t is assigned the current value of x.

Algorithm 1 Sequential abstract interpretation.

```
1:  function SEQABSINT( $G$ : the control-flow graph )
2:    $Env(n)$ is initialized to $\top$ if $n \in$ ENTRY($G$), else to $\bot$
3:    $WL \leftarrow$ ENTRY($G$)
4:    while $\exists n \in WL$
5:      $WL \leftarrow WL \setminus \{n\}$
6:      $e \leftarrow$ TRANSFER($n, Env(n)$)
7:      for all $n' \in$ SUCCS($G, n$) such that $e \not\sqsubseteq Env(n')$
8:        $Env(n') \leftarrow Env(n') \sqcup e$
9:        $WL \leftarrow WL \cup \{n'\}$
10: return $Env$
```

The standard work-list implementation of an abstract-interpretation based analysis [23] is shown in Algorithm 1. The input is a control-flow graph G, where ENTRY(G) is the entry node and SUCCS(G, n) is the set of successors of node n. Env is a function mapping each node n to an environment immediately before n is executed. The initial environment $\top$ associated with the entry node means that all program variables can take arbitrary values, e.g., $x = y = \cdots = [-\infty, \infty]$ for integer variables. The initial environments for all other nodes are set to $\bot$ (the absence of values).

The work-list, WL, is initially populated only with the entry node of the control-flow graph. The fix-point computation in Algorithm 1 is performed in the while-loop: a node $n \in WL$ is removed and has its transfer function executed, resulting in the new environment e. The function TRANSFER takes a node n and the environment $Env(n)$ as input and returns the new environment e (result of executing n in $Env(n)$) as output. If a successor of the node n has a current environment with less information than e (as determined by $\not\sqsubseteq$), then it is added to the work-list and its environment is expanded to include the new information (Lines 7-9). The process proceeds until the work-list is empty, i.e., all the environments have stabilized. Standard widening and narrowing operators [2] may be used at Line 8 to guarantee termination and ensure speedy convergence.

3.2 Thread-modular Abstract Interpretation

Next, we review thread-modular abstract interpretation: an iterative application of a sequential abstract interpreter on each thread in the presence of a joined set of interferences from all other threads. Since a thread-modular analysis never constructs the *product* graph of all threads in the program, it avoids the state space explosion encountered by non-thread-modular methods [12].

First, we make a slight modification to the previously described sequential abstract interpretation (Algorithm 1); the *per-thread* abstract interpretation must consider both the thread-local environment and the *interferences* from other threads. Here, an interference is an environment resulting from executing a shared memory write. Let SEQABSINT-MODIFIED(G, i) be the modified abstract analyzer, which takes an additional environment i as input. The environment i represents a joined set of interferences from all the other threads. We also modify the transfer function TRANSFER($n, Env(n)$) of shared memory read as follows: for `t=load x`, where `x` is a shared variable, we allow `t` to read either from the thread-local environment $Env(n)$ or from i, the interference parameter. For example, if the thread-local environment before the load statement contains $x = [10, 15]$ and the interference parameter contains $x = [50, 60]$, we would have $t = [10, 15] \sqcup [50, 60] = [10, 60]$.

Algorithm 2 Thread-modular abstract interpretation.

```
function THREADMODABSINT( $Gs$ : the set of CFGs )
  $TE \leftarrow \varnothing$
  $I \leftarrow \varnothing$
  repeat
    $I' \leftarrow I$
    for all $g \in Gs$
      $i \leftarrow \bigsqcup\{e \mid e \in I(g'), g' \in Gs, \text{ and } g' \neq g\}$    ▷ Sec. 3.2
      $Env \leftarrow$ SEQABSINT-MODIFIED($g, i$)
      $TE \leftarrow TE \uplus Env$
    for all $(n, e) \in TE$
      if $n$ is a shared memory write in $g \in Gs$
        $I(g) \leftarrow I(g) \sqcup$ TRANSFER($n, e$)
  until $I = I'$
  return $TE$
```

Algorithm 2 shows the thread-modular analysis procedure. The input is the set Gs of control-flow graphs, one per thread. The output, TE, is a function mapping the thread nodes (nodes in all threads) to environments. During the analysis, each thread-local CFG g has an associated interference environment $I(g)$: the environment is the join of all environments produced by shared memory writes in the thread g. Due to their one-to-one correspondence, we will use thread and its (control-flow) graph interchangeably.

Inside the thread-modular analysis procedure, both TE and I are initially empty. Then, the sequential abstract interpretation procedure is invoked to analyze each thread $g \in Gs$. The environment i (Line 7) is the join of all interfering environments from other threads. The sequential analysis result, Env, is a function mapping nodes in g to their corresponding environments. With a slight change of notation, we use $TE \uplus Env$ (Line 9) to denote the join of environments from TE and Env on their matching nodes. Let A and B be sets of pairs of the form $\{(n, e), \ldots\}$; then $A \uplus B$ denotes the join of environments on the matching nodes.

After analyzing all the threads (Lines 6–9), we take the results (TE) and compute the new interferences: for each thread g, the new environment $I(g)$ is the join of all environments produced by the shared memory writes (Lines 10–12). The analysis repeats until the interferences stabilize ($I = I'$), meaning that environments in all node (TE) also stabilize. Again, standard widening and narrowing operators [2] may be used to ensure speedy convergence. Overall, the thread-modular analysis is an additional fix-point computation on the set of interferences relative to sequential analysis, with the same termination and soundness guarantees [22].

4. FLOW-SENSITIVE THREAD-MODULAR ANALYSIS

In this section, we present our new method for flow-sensitive thread-modular analysis. For ease of comprehension, we shall postpone the presentation of the constraint-based feasibility checking until Section 5, while focusing on explaining our method for maintaining inter-thread flow-sensitivity during thread-modular analysis.

4.1 The New Algorithm

Before diving into the new algorithm, notice that the reason why Algorithm 2 is flow-insensitive is because all environments from interfering stores of other threads are joined (Line 7) prior to the thread-modular analysis. Furthermore, within the thread-modular analysis routine, SEQABSINT-MODIFIED, the combined interfering environment, i, is joined again with the thread-local environment during the application of the transfer function at each CFG node. Such eager join operations are the main sources of inaccuracy in existing methods. First, inaccuracy arises from the join operation itself: it tends to introduce additional behaviors, e.g., $[0, 0] \sqcup [10, 10] = [0, 10]$. Second, a thread is allowed to see *any* combination of interfering stores even if some of them are obviously infeasible (e.g., Section 2, Figure 2).

To avoid such drastic losses in accuracy, we need to make fundamental changes to the thread-modular analysis procedure.

- For each thread $g \in Gs$, instead of defining its interference as a single environment, we use a set of pairs (n, e) where n is a CFG node of a shared memory write and e is the environment after n.
- For each shared variable read, instead of it reading from the eagerly joined set of environments, we maintain a set, $LIs(l) = \{(n, e), \ldots\}$, where each (n, e) represents an interfering store and the store's interfering environment.
- For each thread $g \in Gs$, instead of representing the interferences from all other threads as the join of the interfering environments (Line 7, Algorithm 2), we represent them as a set I_c of *interference combinations*: each $i_c \in I_c$ is a distinct combination of the store-to-load flows for all $l \in$ LOADS(g).

Algorithm 3 shows our new analysis: in the remainder of this section, we shall compare it with Algorithm 2 and highlight their differences. There are two main differences. First, the interferences are represented as a set of pairs of store statements and their associated environment (Line 13). We modify $\uplus$ to be the join of environments of pairs with matching nodes across two sets. Recall that if A and B are sets of pairs of the form $\{(n, e), \ldots\}$, then $A \uplus B$ denotes the join of environments on the matching nodes. Second, we compute the set I_c of feasible and non-redundant interference combinations (store-to-load flows) for a thread (Line 7) and analyze a thread in the presence of each combination individually (Lines 8–10). That is, for each call to the sequential abstract interpreter SEQABSINT-MODIFIED2, as the second parameter, instead of passing the join of interferences from all other threads, we pass each $i_c \in I_c$ to map every load to an interfering store individually.

Algorithm 3 Flow-sensitive thread-modular analysis.

```
 1: function ThreadModAbsInt-Flow($Gs$: the set of CFGs)
 2:   $TE \leftarrow \varnothing$
 3:   $I \leftarrow \varnothing$
 4:   repeat
 5:     $I' \leftarrow I$
 6:     for all $g \in Gs$
 7:       $I_c \leftarrow$ InterferenceComboFeasible$(g, I)$
 8:       for all $i_c \in I_c$                                    ▷ Sec. 4
 9:         $Env \leftarrow$ SeqAbsInt-Modified2$(g, i_c)$
10:         $TE \leftarrow TE \uplus Env$
11:     for all $(n, e) \in TE$
12:       if $n$ is a shared memory write in $g \in Gs$
13:         $I(g) \leftarrow I(g) \uplus \{$Transfer$(n, e)\}$
14:   until $I = I'$
15:   return $TE$
16:
17: function InterferenceComboFeasible($g$, $I$)
18:   $I_c \leftarrow \varnothing$
19:   $VEs \leftarrow \{(n, e) \mid (n, e) \in I(g'),\ g' \in Gs,$ and $g' \neq g\}$
20:   for all $l \in$ Loads$(g)$
21:     $LIs(l) \leftarrow \{(s_{dummy}, e_{self})\}$
22:     if $l$ is not self-reachable
23:       for all $(n, e) \in VEs$
24:         if LoadVar$(l)$ = StoreVar$(n)$
25:           $LIs(l) \leftarrow LIs(l) \cup \{(n, e)\}$
26:     else                                      ▷ Handling loads in loops
27:       for all $(n, e) \in VEs$
28:         if (LoadVar$(l)$ = StoreVar$(n)$)
                 $\wedge\ \neg$MustHappenBefore$(l, n)$
29:           $LIs(l) \leftarrow LIs(l) \uplus \{(s_{dummy}, e)\}$
30:   $Es \leftarrow$ CartesianProduct$(LIs)$                          ▷ Sec. 6
31:   for all $i_c \in Es$
32:     if Query.IsFeasible$(i_c)$                                ▷ Sec. 5
33:       $I_c \leftarrow I_c \cup \{i_c\}$
34:   return $I_c$
```

4.2 The Interference Combinations

Inside INTERFERENCECOMBOFEASIBLE(g, I), we compute the set I_c of feasible interference combinations. Here, LOADS(g) is the set of shared variable reads in thread g, LOADVAR(l) is the variable used in the load instruction l, and STOREVAR(s) is the variable stored-to in the store instruction s.

We first compute the set VEs of interferences from other threads (Line 19); each pair $(n, e) \in VEs$ is a store and environment from a thread other than g. Then, we pair each load $l \in$ LOADS(g) with any corresponding store in VEs (Lines 20–29); the result is stored in LIs which maps each load instruction l to a set of stores in the form of (n, e) pairs. The special pair (s_{dummy}, e_{self}) indicates the thread should read from its intra-thread environment. For now, ignore Lines 26–29 since they are related to the handling of loops — we discuss how loops are handled during the computation of interference combinations in the next subsection.

Next, the function CARTESIANPRODUCT takes LIs as input and returns the complete set of interference combinations from

$LIs(l_1) \times \cdots \times LIs(l_k)$. To make what we have explained so far clearer, consider an example program with two threads: g_1 and g_2. Thread g_1 has two loads, $\textsc{Loads}(g_1) = \{l_1, l_2\}$ such that $\textsc{LoadVar}(l_1) = \texttt{x}$ and $\textsc{LoadVar}(l_2) = \texttt{y}$. Thread g_2 has three interfering environments: two on x, s_1 and s_2, with associated environments e_1 and e_2, respectively; and another, s_3, on y, with environment e_3. Assume we are currently analyzing g_1 in the presence of interferences from g_2.

We first use the set I of interferences to collect the interferences from g_2 in *VEs*: $\{(s_1, e_1), (s_2, e_2), (s_3, e_3)\}$. Next, we compute *LIs* for the two loads $\{l_1, l_2\}$ in thread g_1. We pair l_1 with the two interferences on x from s_1 and s_2, and pair l_2 with the single interference on y from s_3. Using $[\cdots]$ to denote a list of items, we represent the result as $LIs(l_1) = [(s_1, e_1), (s_2, e_2), (s_{dummy}, e_{self})]$ and $LIs(l_2) = [(s_3, e_3), (s_{dummy}, e_{self})]$. Without any optimizations, the resulting Cartesian product $Es = LIs(l_1) \times LIs(l_2)$ would contain the following items:

$$
\begin{aligned}
i_{c_1} &= \{\langle l_1, (s_1, e_1)\rangle, & &\langle l_2, (s_3, e_3)\rangle\},\\
i_{c_2} &= \{\langle l_1, (s_2, e_2)\rangle, & &\langle l_2, (s_3, e_3)\rangle\},\\
i_{c_3} &= \{\langle l_1, (s_{dummy}, e_{self})\rangle, & &\langle l_2, (s_3, e_3)\rangle\},\\
i_{c_4} &= \{\langle l_1, (s_1, e_1)\rangle, & &\langle l_2, (s_{dummy}, e_{self})\rangle\},\\
i_{c_5} &= \{\langle l_1, (s_2, e_2)\rangle, & &\langle l_2, (s_{dummy}, e_{self})\rangle\},\\
i_{c_6} &= \{\langle l_1, (s_{dummy}, e_{self})\rangle, & &\langle l_2, (s_{dummy}, e_{self})\rangle\}.
\end{aligned}
$$

For each combination $i_c \in Es$, we check if it is feasible (Lines 31–33): the infeasible combinations will be filtered out, and the result, I_c, is returned. We discuss how we determine the feasibility of an interference (Line 32) in Section 5.

Continuing with the algorithm's description, on Line 9 the sequential abstract interpretation, $\textsc{SeqAbsInt-Modified2}$, takes g and each $i_c \in I_c$ as input and returns a node-to-environment map, *Env*, as output. During this per-thread analysis, the transfer function of a load uses only i_c to determine the environment to use. When a load l_1 is being executed, if the special item $\langle l_1, (s_{dummy}, e_{self})\rangle$ is in i_c, the load reads from its own thread-local environment at l_1; if the remote store environment $\langle l_1, (s, e)\rangle$ is in i_c, the load also reads from the remote environment e.

At this point, we have improved the prior work (Algorithm 2) to avoid inaccuracies from over-approximations caused by the eager join over all interferences. The cost for this accuracy is explicitly testing each of the combinations of potential interferences. However, we have not presented our methods for clustering and pruning (Section 6) as well as checking if any of the combinations are *infeasible* (Section 5). By applying such optimization techniques, we cannot only drastically reduce the overhead of running the abstract interpretation subroutine but also increase the accuracy.

4.3 Handling Loops

Since a load within a loop may execute many times, the number of stores it could read from may be infinite. To guarantee termination, we join all the interfering stores that *may* affect a load in a loop with the environment within the thread at the time of the load. By doing this, we conservatively treat all these feasible interferences in a flow-insensitive manner for loads within loops.

Specifically, Lines 26–29 perform the join of interferences for loads within a loop. For a given load, all stores on the same variable that must-not-happen after the load are considered (we will further discuss the happens-before constraints in Section 5). For these conflicting stores, all of the environments are joined together on a single dummy node (s_{dummy}). In the end, each self-reachable load has a single (joined) environment. Consequently, during the Cartesian product computation, it will have a single interference. Within the sequential abstract interpreter, the load merges the thread-local environment and this single interfering environment.

However, even in such case, our new method is more accurate than the prior work. Consider the example in Figure 3. Thread 1 executes a load in a while-loop running an arbitrary number of times concurrently with thread 2 before creating thread 3. Because of the thread creation, there is a must-happen-before edge between the load in thread 1 (Line 5) and the write in thread 3 to x. When constructing the interference combinations for the load in thread 1 (l), there are three potential stores: s_{10}, s_{11}, and s_{14} for the writes to x on Lines 10, 11, and 14, respectively.

```
 1  int x = 0;                     9  void thread2() {
 2  void thread1() {              10    x = 1;
 3    create(thread2);            11    x = 2;
 4    while (*) {                 12  }
 5      int t1 = x;               13
 6    }                           14  void thread3() {
 7    create(thread3);            15    x = 10;
 8  }                             16  }
```

Figure 3: Example: handling loops in thread-modular analysis.

When considering s_{10}, the condition on Line 28 of our new algorithm is true since s_{10} does not always happen after l (and similarly for s_{11}). Therefore, $LIs(l)$ is assigned $\{s_{dummy}, e_{10}\}$ initially, where e_{10} is the environment at s_{10}. Next, $LIs(l)$ is assigned $\{s_{dummy}, e_{10} \sqcup e_{11}\}$. Finally, for s_{14}, since it must happen after l, it is not added to *LIs*. When computing the Cartesian product, there is only a single load with a single location-store pair, so there is only one interference combination.

For this example, the analysis results in t1 being 0, 1, or 2. The value of 10 written by thread 3 is excluded using the must-happen-before constraint. So, although multiple interfering stores are merged for the single load within the loop, the accuracy of the analysis is still higher than prior flow-insensitive analyses.

4.4 Correctness

Our method in Algorithm 3 is a form of semantic reduction [3,25] of the interferences allowed by the prior flow-insensitive approach in Algorithm 2. Specifically, the input environment to a load instruction in Algorithm 2 is the join of the set $S = \{\rho, \rho_1, \ldots, \rho_n\}$ where ρ is the intra-thread environment and $\rho_1, \ldots, \rho_n$ are environments from interfering stores. The semantic-reduction operator we use in Algorithm 3 is to apply the transfer function of the load to each element of S individually relative to all other loads (i.e., the Cartesian product). Therefore, the correctness of our algorithm directly follows the correctness argument in [3,25]. Additionally, we remove infeasible interferences combinations (Lines 31-33), which does not affect the soundness of the algorithm.

In the case of loops, the transfer function of a load can be executed more than once: each execution of the transfer function may use a different interference, so, using the same semantic-reduction operator would have resulted in a potentially infinite number of interference combinations. In this case, we conservatively merge all the *feasible* interferences into a single value. Correctness of this treatment directly follows the correctness of Algorithm 2.

In the case of aliasing, our algorithm can be lifted to use the output of any (sound) alias analysis by considering each alias-set as a single variable – it is a standard technique to handle aliasing in static analysis. In such case, our algorithm would operate on these alias-sets instead of on the individual program variables.

5. CONSTRAINT-BASED FEASIBILITY

We now present our procedure for eliminating infeasible combinations of interferences. We revisit Algorithm 3 to show its integration with our new thread-modular analysis procedure.

Removing infeasible interferences from the thread-modular analysis significantly reduces computational overhead and increases accuracy. However, the main problem is that the feasibility checking has to be conducted efficiently for such an optimization to be useful. Therefore, our goal is to make the checking both *sound* and *efficient*. By sound, we mean that if the procedure determines a combination is infeasible then it is truly infeasible. By efficient, we mean that the procedure relies on constructing and solving a system of *lightweight* constraints, i.e., Horn clauses in finite domains, which can be decided using a Datalog engine in polynomial time.

Algorithm 4 Constraint-based feasibility checking.

```
$POs \leftarrow$ PROGRAMORDER-CONSTRAINTS($Gs$)
QUERY.ADD($POs$)
function QUERY.ISFEASIBLE($i_c$: permutation of interferences)
    $Cs \leftarrow$ READSFROM-CONSTRAINTS($i_c$)
    QUERY.ADD($Cs$)
    $res \leftarrow$ QUERY.SATISFIABLE()
    QUERY.REMOVE($Cs$)
    return $res$
```

Algorithm 4 shows the high-level flow of our feasibility analysis procedure. Initially, we traverse the set Gs of control-flow graphs to compute a set POs of constraints representing the order between statements which must hold on all possible executions of the program. We initialize the constraint system with these orderings by calling QUERY.ADD(POs).

During the execution of Algorithm 3 (Lines 31–33), for each $i_c \in I_c$, we compute a set Cs of *reads-from* constraints, which must be enforced in order to realize the interference combination i_c. We add them to the system as well by calling QUERY.ADD(Cs).

Our constraint analysis then, using a set of deduction rules, expands upon these input constraints to generate more constraints. We invoke QUERY.SATISFIABLE to check if the constraint system is satisfiable. The deduction rules are designed such that, if the system is not satisfiable, then i_c is guaranteed to be infeasible. In the remainder of this section, we go into each of these steps in detail.

5.1 The Program-order and the Reads-from Constraints

To check the simultaneous feasibility of POs and Cs, we first compute the *dominators* on a thread's CFG. Given two nodes m and n in a graph g, m dominates n if all paths from the entry of g to n go through m. Then, we define the following relations:

- DOMINATES is the dominance relation on a thread's CFG: $(m, n) \in$ DOMINATES means m dominates n.
- NOTREACHABLEFROM is reachability on a thread's CFG: $(m, n) \in$ NOTREACHABLEFROM means node m can not be reached from node n.
- THCREATES is a parent–child relation over threads: $(p, n_{sta}) \in$ THCREATES if p is thread creation point and n_{sta} is the child thread's start node.
- THJOINS is a parent–child relation over threads: $(p, n_{end}) \in$ THJOINS means p is a thread join on a child thread with node n_{end} as exit.
- $(l, v) \in$ ISLOAD means l is a load of variable v.
- $(s, v) \in$ ISSTORE means s is a store to variable v.
- READSFROM is obtained from the combination i_c under test: $(l, s) \in$ READSFROM if the load l is reading from the store s.

All these relations can be computed from the given set Gs of control-flow graphs efficiently [18]. Furthermore, they are defined over finite domains (sets of nodes or variables), which means constraints built upon these relations are efficiently decidable.

5.2 Deduction Rules for Checking Feasibility

Figure 4 shows the deduction rules underlying our feasibility analysis. If a contradiction is reached after applying the rules to the input constraints, the interference combination is guaranteed to be infeasible. For brevity, we only present the intuition behind these rules. Detailed proofs can be found in our supplementary material.

$$\frac{(m,n) \in \textsc{Dominates} \wedge (m,n) \in \textsc{NotReachableFrom}}{(m,n) \in \text{MHB}} \qquad (1)$$

$$\frac{(m,n_{sta}) \in \textsc{ThCreates}}{(m,n_{sta}) \in \text{MHB}} \qquad \frac{(m,n_{end}) \in \textsc{ThJoins}}{(n_{end},m) \in \text{MHB}} \qquad (2)$$

$$\frac{\begin{array}{c}(l,s_1) \in \textsc{ReadsFrom} \wedge (s_1,s_2) \in \text{MHB} \\ \wedge\ (l,v) \in \textsc{IsLoad} \wedge (s_1,v) \in \textsc{IsStore} \\ \wedge\ (s_2,v) \in \textsc{IsStore}\end{array}}{(l,s_2) \in \text{MHB}} \qquad (3)$$

$$\frac{(a,b) \in \text{MHB} \wedge (b,c) \in \text{MHB}}{(a,c) \in \text{MHB}} \qquad (4)$$

$$\frac{(a,b) \in \text{MHB}}{(a,b) \in \textsc{MustNotReadFrom}} \qquad (5)$$

$$\frac{\begin{array}{c}(l_1,s_1) \in \textsc{ReadsFrom} \wedge (l_1,s_2) \in \text{MHB} \wedge (s_2,l_2) \in \text{MHB} \\ \wedge\ (l_1,v) \in \textsc{IsLoad} \wedge (l_2,v) \in \textsc{IsLoad} \wedge (s_2,v) \in \textsc{IsStore}\end{array}}{(l_2,s_1) \in \textsc{MustNotReadFrom}} \qquad (6)$$

Figure 4: Rules used by our interference feasibility analysis.

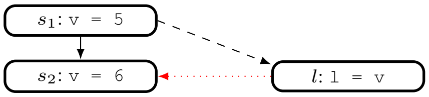

Figure 5: Example: application of Rule 3.

Rules 1, 2, and 3 create the *must-happen-before* relation, MHB, where $(m, n) \in$ MHB means node m must happen before node n under the current interference combination i_c. Rule 4 is simply the transitive property for the must-happen-before relation.

First, if m dominates n in a CFG, since m occurs before n on *all* program paths, m must happen before n (Rule 1). We check if n can reach m to ensure that even if m dominates n, m can never subsequently occur after n (e.g., if n is in a loop). Similarly, since a thread cannot execute before it is created, or after it terminates, THCREATES and THJOINS also map directly to MHB (Rule 2).

Rule 3 captures the scenario of two stores overwriting each other as shown in Figure 5. Here, one thread has stores s_1 and s_2, and a second thread has one load l. READSFROM(l, s_1) is represented by the dashed edge (flow of data) from s_1 to l. MHB(s_1, s_2) is represented by the solid edge from s_1 to s_2. Given the two previous relations, the rule deduces the relation MHB(l, s_2), represented by the red dotted edge. The implication is that for load l to read from the first store s_1, l must happen before the second store s_2.

The intuition behind this rule is that if s_2 executes before l, then s_2 would overwrite the value of s_1, making it impossible for l to read the value of s_1. Note that this must-happen-before constraint is *only* considered for i_c, the current combination of interferences: it does *not* hold globally across all executions of the program.

Rule 5 introduces the MUSTNOTREADFROM relation. For a load store pair $(l, s) \in$ MUSTNOTREADFROM if in the current interference combination l cannot read from s.

Rule 6 prevents a thread from reading an interference after it has been over-written, shown in Figure 6. The first thread has a store s_1, and the second thread has load l_1, store s_2, and then load l_2. Again,

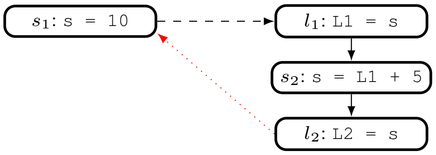

Figure 6: Example: application of Rule 6.

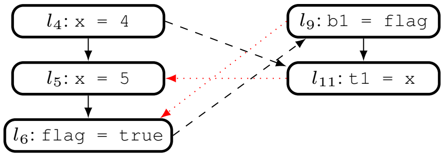

Figure 7: Input and implied constraints for Figure 2.

```
1   int x = 0;
2   void thr(int v) {
3     int t1 = 5 * v;
4     int t2 = x;
5     x = t1 + t2;
6     if (t1 < 0)
7       ERROR!;
8   }
```

```
9   int main() {
10    thread_create(thr,5);
11    thread_create(thr,10);
12    x = 1;
13    thread_exit(0);
14  }
```

Figure 8: Example: property directed redundancy pruning.

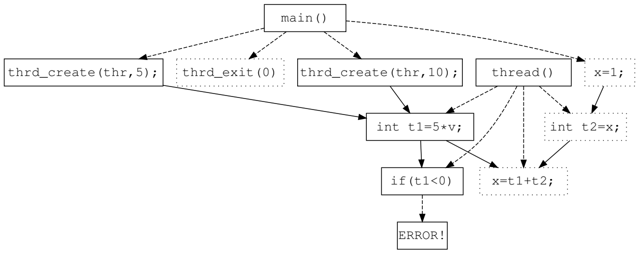

Figure 9: The program dependence graph for Figure 8.

MHB relations are represented by solid edges, READSFROM(l_1, s_1) is represented by the dashed edge, and MUSTNOTREADFROM(l_2, s_1) is represented by the red dotted edge.

Conceptually, the rule captures the situation when a value is read from an interference (l_1: L1=s), followed by a modification of the same memory location that was loaded (s_2: s=L1+5), followed by a load of the same location (l_2: L2=s). Intuitively, since the interfering value was just overwritten, it cannot be loaded again. Therefore, the pair (l_2, s_1) is added to MUSTNOTREADFROM.

Finally, our constraint analysis does not try to identify all infeasible combinations for efficiency reasons. However, the framework is generic enough to allow new rules and other types of constraint solvers to be plugged in easily to refine the approximation.

5.3 The Running Example

We revisit the example in Figure 2 to illustrate our feasibility checking for one interference combination (Figure 7). Our goal is to decide if READSFROM(l_9, l_6) and READSFROM(l_{11}, l_4) can co-exist. Initially, our constraint system would have the solid edges from the MHB relations, which represent the program-order constraints, and the dashed edges from the READSFROM relations, which represent the current interference combination i_c.

First, we can deduce MHB(l_{11}, l_5) by applying Rule 3: if l_{11} does not happen before l_5, l_5 would overwrite the value of x, preventing l_{11} from reading from l_4. This deduced MHB relation is represented by the red dotted edge in the figure.

Next, we can deduce a must-happen-before relation between l_9 and l_6 by applying Rule 4 twice. That is, MHB$(l_9, l_{11}) \wedge$ MHB(l_{11}, l_5) implies MHB(l_9, l_5), followed by MHB$(l_9, l_5) \wedge$ MHB(l_5, l_6) implies MHB(l_9, l_6). The result is represented by the red dotted edge from l_9 to l_6.

At this point, we have a contradiction: since b1=flag must-happen-before flag=true, b1 cannot read the value of true (Rule 5). So, this interference combination is proved to be infeasible. (There are more implied edges in Figure 7; for clarity, we show only those relevant to the check.)

6. OPTIMIZATIONS WITH CLUSTERING AND PRUNING

To reduce the number of interference combinations, we apply dependency-based clustering analysis and property-directed pruning. Consider the program in Figure 8: the main thread creates two children in the function thr with arguments 5 and 10, respectively. The thr function performs a store to x (Line 5) based on the value passed as an argument (v). At the load of x in the thr function, the value may come from the initial value 0, from the main thread (Line 12), or from the other thread thr (Line 5). This results in three combinations of loads in thr to be tested on every iteration.

However, the reachability of ERROR! does *not* depend on the value loaded from x, since the error condition (t1 $<$ 0) only depends on the argument passed to thr. As such, the load of x is immaterial to the property. We can formally capture this notion of immateriality using *control and data dependencies* [7].

Intuitively, a statement s is data dependent on t if the value of t may affect the computation of s. For example, in Figure 8, the statement t1=5*v is data dependent on the input parameter v. On the other hand, a statement l is control dependent on m if the execution of m affects the reachability of l. For example, the ERROR! statement in Figure 8 is control-dependent on the evaluation of the predicate t1<0.

The composition of the control- and data-dependency relations is the *program dependence graph* [7]. Note that in concurrent programs, the dependency graph may span across multiple threads, due to the flow of data from shared memory writes to reads.

Next, we show two applications of the program dependence graph for optimizing our overall algorithm.

6.1 Property-guided Pruning

First, we create the *backward slice* on every property in the program. The backward slice with respect to a property s contains all the statements involved in the computation of s (Theorem 2.2 [15]). As an example, the program dependence graph for Figure 8 is shown in Figure 9. Dashed edges are control dependencies and solid edges are data dependencies. The backward slice on the ERROR! statement is also shown: the dotted nodes are nodes not contained in the slice. All computations involving x can be ignored, since the slice shows that they are irrelevant to the property being verified.

During our analysis, the transfer function of a statement not on the backward slice is the *identity*. And any load not on the backward slice is ignored when computing interference combinations.

6.2 Dependency-guided Clustering

Second, during the generation of combinations of interferences, we do not always consider the Cartesian product across all sets of loads. Instead, we group loads together to form *cluster* and only

```
1  int x = 0;
2  int y = 0;
3  void thread1() {
4    x = 1;
5    y = 1;
6  }
```

```
7  void thread2() {
8    int t1 = x;
9    int t2 = y;
10   assert(x >= 0);
11   assert(y >= 0);
12 }
```

Figure 10: Example: dependency guided clustering.

generate interference combinations within each cluster.

Consider the program in Figure 10. Initially, x and y are zero; the first thread sets them to one, and the second thread checks the property that they are both greater than or equal to zero. The backward slice on assert(x>=0) contains lines 4, 8, and 10. The backward slice on assert(y>=0) contains lines 5, 9, and 11. Without optimization, in Algorithm 3, the loads on x and y both have two potential environments to read from: the interfering store and the environment within the thread. In total, there are $2 * 2 = 4$ combinations leading to four abstract interpreter executions.

The backward slices on properties in the program form disjoint subgraphs; e.g., a graph with the operations on x and those on y. The interference combinations in the subgraphs can be considered independently requiring only $max(2, 2) = 2$ interpreter executions.

7. EXPERIMENTS

We implemented our method in a software tool named WATTS, designed for verifying multithreaded programs represented in the LLVM intermediate language. All experiments were performed on C programs written using POSIX threads. We used the Apron library [13] for implementing the sequential analyzer over interval and octagon abstract domains, and the Datalog solver in Z3 (μZ [11]) for solving the causality constraints.

We evaluated WATTS on two sets of benchmark programs. The first set consists of some multithreaded programs from SVCOMP [28]. The second set consists of Linux device drivers from [29] and [5]. In all benchmark programs, the reachability properties are expressed in the form of embedded assertions. Table 2 shows the characteristics of these programs, including the name, the number of lines of code (LoC), the number of threads, and the number of assertions. In total, our benchmarks have 26,309 lines of code and 10,078 assertions. For the device driver benchmarks, in particular, since assertions are not included in the original source code, we manually added these assertions. We performed all experiments on a computer with 8 GB RAM and a 2.60 GHz CPU.

Although we used the benchmarks from [5], the verification problem targeted by our method is significantly different. DUET assumes each device driver is a parametric program, whereas our method analyzes programs with a finite number of threads. As shown in Section 2, our method, using a set of control-flow graphs as opposed to a monolithic data-flow graph, is often more accurate. During experiments, we ran both WATTS and DUET on all benchmarks with our assertions. WATTS verified 548 more properties than DUET, whereas DUET did not verify any property not verified by WATTS. The result shows that DUET's abstraction for infinite threads leads to loss of precision. Therefore, in the remainder of this section, we do not directly compare WATTS with DUET.

Instead, we focus on comparing our method with the prior thread-modular approaches [8, 20–22]. For evaluation purposes, we implemented both methods in WATTS: the flow-insensitive analysis of Algorithm 3 and the flow-insensitive analysis of Algorithm 2.

Table 3 shows the results of comparing Algorithm 3 and Algorithm 2 in the interval abstract domain. Column 1 shows the name of each benchmark. Columns 2–3 show the result of running Algorithm 2. Columns 4–5 show the result of running Algorithm 3 without using the feasibility checking. Columns 6–7 show the result with the feasibility checking. Columns 8–9 show the result with clustering/pruning optimizations. For each test case, Tm. is the run time in seconds and Verif. is the number of verified properties. The last row shows the sum of all columns.

Table 2: Statistics of the benchmarks in our experiments.

Name	LoC	Threads	Properties	Source
thread01	29	3	1	created
create01	24	2	1	created
create02	28	2	1	created
sync01	38	3	1	[28]
sync02	36	3	1	[28]
intra01	41	3	1	created
dekker1	65	3	1	[28]
fk2012	88	3	1	[5], added asserts
keybISR	62	3	2	[29]
ib700_01	346	3	1	[5], added asserts
ib700_02	466	23	1	[5], added asserts
ib700_03	587	41	81	[5], added asserts
i8xxtco_01	735	3	1	[5], added asserts
i8xxtco_02	901	22	1	[5], added asserts
i8xxtco_03	1027	42	103	[5], added asserts
machz_01	667	8	1	[5], added asserts
machz_02	795	29	1	[5], added asserts
machz_03	881	41	83	[5], added asserts
mix_01	457	12	1	[5], added asserts
mix_02	580	31	62	[5], added asserts
pcwd_01	1197	8	1	[5], added asserts
pcwd_02	1405	41	81	[5], added asserts
sbc_01	686	24	1	[5], added asserts
sc1200_01	715	24	1	[5], added asserts
sc1200_02	768	31	93	[5], added asserts
smsc_01	904	12	1	[5], added asserts
smsc_02	931	12	24	[5], added asserts
sc520_01	806	4	1	[5], added asserts
sc520_02	880	41	81	[5], added asserts
wfwdt_01	777	4	1	[5], added asserts
wfwdt_02	907	51	101	[5], added asserts
wdt	1023	31	1	[5], added asserts
wdt977_01	867	16	1	[5], added asserts
wdt977_02	877	31	92	[5], added asserts
wdt_pci	1133	31	1	[5], added asserts
wdt_pci02	1165	31	122	[5], added asserts
pcwdpci_01	1363	64	128	[5], added asserts

Compared to the flow-insensitive approach (Columns 2–3), our baseline flow-sensitive method (Columns 4–5) can already achieve a 12x increase in the number of verified properties (from 38 to 452) without employing the lightweight constraint-based feasibility checking. This demonstrates the benefits of delaying the join operation across threads. Furthermore, the significant increase in accuracy comes at the modest 1.5x increase in run time.

With the constraint-based feasibility checking, a more significant improvement can be observed (Columns 6–7): there is a 28x increase in the number of verified properties (from 38 to 1,078) compared to the prior flow-insensitive approach. Furthermore, the large increase in accuracy comes with only an 1.6x increase in run time.

Finally, with the optimizations from Section 6, our method improves further (Columns 8–9). Compared to the prior flow-insensitive approach (Columns 2–3), our method only has a 1.4x increase in the runtime overhead but with a 28x increase in number of verified properties. Compared to the version of our method without optimizations (Columns 6–7), the version with optimization finishes the entire analysis 1.4x faster. Additionally, the optimized version finishes slightly *faster* than the non-constraint based approach (Columns 4–5) while able to verify 2.4x as many properties.

Note that across all experiments, the number of verified properties are strictly increasing: e.g., the flow-sensitive approach with optimizations verifies all the properties of the flow-insensitive approach

Table 3: Experimental results in the interval domain.

	Flow-insensitive		Flow-sensitive		F.-s. + Const.		F.-s. + Opt.	
Name	Tm. (s)	Verif.	Tm. (s)	Verif.	Tm. (s)	Verif.	Tm. (s)	Verif.
thread01	0.03	0	0.05	0	0.05	1	0.09	1
create01	0.02	0	0.03	0	0.04	1	0.07	1
create02	0.03	0	0.03	0	0.03	1	0.07	1
sync01	0.04	0	0.05	1	0.06	1	0.07	1
sync02	0.04	0	0.06	0	0.07	1	0.07	1
intra01	0.03	0	0.03	0	0.03	1	0.08	1
dekker1	0.14	0	9.81	0	2.10	1	0.75	1
fk2012	0.10	0	0.25	0	0.25	1	0.18	1
keybISR	0.05	0	0.15	0	0.14	2	0.12	2
ib700_01	0.09	0	0.10	0	0.10	1	0.13	1
ib700_02	1.17	0	0.88	0	0.95	1	1.03	1
ib700_03	33.46	0	40.95	40	36.95	81	37.81	81
i8xxtco_01	0.15	0	0.13	0	0.13	1	0.22	1
i8xxtco_02	1.34	0	0.96	0	1.02	1	1.24	1
i8xxtco_03	38.07	18	50.78	61	47.90	103	55.24	103
machz_01	0.21	0	0.18	0	0.18	1	0.29	1
machz_02	0.97	0	0.69	0	0.76	1	0.94	1
machz_03	41.30	0	74.32	42	153.50	83	118.25	83
mix_01	0.24	0	0.19	0	0.20	1	0.29	1
mix_02	12.42	1	15.22	31	13.24	62	15.28	62
pcwd_01	0.25	0	0.21	0	0.21	1	0.32	1
pcwd_02	33.12	0	41.57	40	33.77	81	38.23	81
sbc_01	0.60	0	0.73	0	1.09	1	0.57	1
sc1200_01	0.53	0	0.62	0	0.47	1	0.54	1
sc1200_02	70.46	0	119.24	62	161.00	93	122.48	93
smsc_01	0.35	0	0.32	0	0.40	1	0.51	1
smsc_02	3.73	0	7.27	1	15.39	24	6.12	24
sc520_01	0.64	0	1.23	0	0.73	1	0.72	1
sc520_02	50.87	0	81.27	39	65.95	81	46.95	81
wfwdt_01	0.61	0	1.20	0	0.71	1	0.70	1
wfwdt_02	94.54	0	148.32	0	118.39	101	83.91	101
wdt	0.71	0	0.49	0	0.55	1	0.69	1
wdt977_01	0.57	0	0.44	0	0.49	1	0.65	1
wdt977_02	51.86	0	58.92	32	86.16	93	92.01	93
wdt_pci	0.79	0	0.55	0	0.61	1	0.77	1
wdt_pci02	75.14	1	114.55	31	100.12	122	110.33	122
pcwdpci_01	91.10	18	115.82	72	136.13	128	109.02	128
Total	605.77	38	887.61	452	979.87	1,078	846.74	1,078

and more. At most we were able to verify 1,078 properties. Those we missed largely were due to cross-thread synchronization which was not captured by our constraint analysis.

In addition to the results in Table 3, we also performed experiments using the octagon abstract domain. We observed little increase in accuracy as a result of this change, indicating that the properties being verified are mostly on inter-thread concurrency control behavior, and therefore a more sophisticated representation of numerical relations over the program variables does not offer more advantages. For brevity, we omit the result table for the octagon domain.

In the past, introducing flow-sensitivity to static analysis often results in scalability issues (e.g., [12]); however, this is not the case for our method. Figure 11 shows our experiments on a parametrized program named *i8xx_tco*, where the run time of our method grows only moderately with the increase in program size. Here, the x-axis is the number of threads of the program and the y-axis is the run time. The optimized method has slightly *lower* runtime than the least accurate flow-insensitive approach. Furthermore, our method enjoys an almost linear growth in the execution time, indicating it is more scalable than the other methods.

8. RELATED WORK

There is a large body of work on the static analysis and formal verification of multithreaded programs, but none of these existing methods can obtain flow sensitivity in thread-modular analysis with a reasonable run-time cost. For brevity, we discuss only those that are most relevant to our new method. The interested reader can see Rinard [24] for a survey of early work.

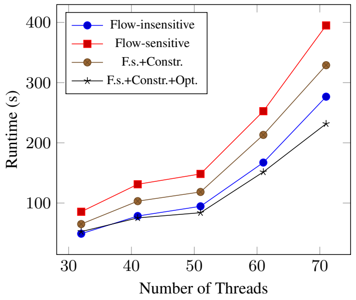

Figure 11: Runtime overhead versus number of threads.

Thread-modular abstract interpretation was introduced by Ferrara [8] and Miné [20, 21]. As shown, their approaches eagerly joined interferences and considered them flow-insensitively, thus introducing inaccuracies. Our method avoids such drawbacks. Ferrara [8] also introduced models designed specific for the Java memory model to remove certain types of infeasible interferences in an *ad hoc* fashion. Our constraint-based feasibility checking, in contrast, is more general and systematic, and can handle transitive must-happen-before constraints as well as other constraints both within and across threads.

Miné [22] introduced an extension to their prior thread-modular analysis to compute *relational* interferences. This allows for relations between variables to be maintained across threads, thereby bringing more accuracy than using *non-relational* interferences. However, as we have explained earlier, this technique is orthogonal and complementary to our new method.

Farzan and Kincaid [5] introduced a method to iteratively construct a monolithic data-flow graph for a concurrent program. However, their technique, as well as similar methods designed for parametric programs [14, 17], targets the problem of verifying properties in a concurrent program with an unbounded number of threads. As we have shown earlier, our new method is often significantly more accurate than these existing methods.

Thread-modular approaches have been applied to model checking [9, 10] and symbolic analysis [26, 27]. There are also works on verifying concurrent software using abstraction and stateless model checking [16, 30–32]. However, these approaches in general are either heavyweight or under-approximative, and therefore are complementary to our abstract-interpretation based approach.

9. CONCLUSIONS

We have presented a flow-sensitive method for composing standard abstract interpreters to form a more accurate thread-modular analysis procedure for concurrent programs. Our method relies on constructing and solving a system of happens-before constraints to decide the feasibility of inter-thread interference combinations. We also use clustering and pruning to reduce the run-time overhead of our analysis. We have implemented our method in a software tool and evaluated it on a large set of multithreaded C programs. Our experimental results show that the new method can significantly increase the accuracy of the thread-modular analysis while maintaining a modest run-time overhead.

10. ACKNOWLEDGMENTS

This work was primarily supported by the NSF under grants CCF-1149454, CCF-1405697, and CCF-1500024. Partial support was provided by the ONR under grant N00014-13-1-0527.

11. REFERENCES

[1] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 168–176, 2004.

[2] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[3] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 269–282, 1979.

[4] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 337–340, 2008.

[5] A. Farzan and Z. Kincaid. Verification of parameterized concurrent programs by modular reasoning about data and control. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 297–308, 2012.

[6] A. Farzan and Z. Kincaid. Duet: Static analysis for unbounded parallelism. In *International Conference on Computer Aided Verification*, pages 191–196, 2013.

[7] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, July 1987.

[8] P. Ferrara. Static analysis via abstract interpretation of the happens-before memory model. In *Tests and Proofs*, pages 116–133. 2008.

[9] C. Flanagan and S. Qadeer. Thread-modular model checking. In *Proceedings of the 10th International Conference on Model Checking Software*, pages 213–224, Berlin, Heidelberg, 2003. Springer-Verlag.

[10] T. A. Henzinger, R. Jhala, R. Majumdar, and S. Qadeer. Thread-modular abstraction refinement. In *International Conference on Computer Aided Verification*, pages 262–274, 2003.

[11] K. Hoder, N. Bjørner, and L. de Moura. muZ - an efficient engine for fixed points with constraints. In *International Conference on Computer Aided Verification*, pages 457–462, 2011.

[12] B. Jeannet. Relational interprocedural verification of concurrent programs. *Software and Systems Modeling*, 12(2):285–306, 2013.

[13] B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. In A. Bouajjani and O. Maler, editors, *International Conference on Computer Aided Verification*, pages 661–667. 2009.

[14] A. Kaiser, D. Kroening, and T. Wahl. Dynamic cutoff detection in parameterized concurrent programs. In *International Conference on Computer Aided Verification*, pages 645–659, 2010.

[15] K. Kennedy and J. R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[16] M. Kusano and C. Wang. Assertion guided abstraction: a cooperative optimization for dynamic partial order reduction. In *IEEE/ACM International Conference On Automated Software Engineering*, pages 175–186, 2014.

[17] S. La Torre, P. Madhusudan, and G. Parlato. Model-checking parameterized concurrent programs using linear interfaces. In *International Conference on Computer Aided Verification*, pages 629–644, 2010.

[18] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, Jan. 1979.

[19] A. Miné. The octagon abstract domain. *Higher Order Symbol. Comput.*, 19(1):31–100, Mar. 2006.

[20] A. Miné. Static analysis of run-time errors in embedded critical parallel c programs. In G. Barthe, editor, *Programming Languages and Systems*, pages 398–418. 2011.

[21] A. Miné. Static analysis by abstract interpretation of sequential and multi-thread programs. In *Proc. of the 10th School of Modelling and Verifying Parallel Processes (MOVEP 2012)*, pages 35–48, 3–7 Dec. 2012.

[22] A. Miné. Relational thread-modular static value analysis by abstract interpretation. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 39–58, 2014.

[23] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[24] M. Rinard. Analysis of multithreaded programs. In *Static Analysis*, pages 1–19. 2001.

[25] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 105–118, 1999.

[26] N. Sinha and C. Wang. Staged concurrent program analysis. In *ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 47–56, 2010.

[27] N. Sinha and C. Wang. On interference abstractions. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 423–434, 2011.

[28] SVCOMP. International competition on software verification. `http://sv-comp.sosy-lab.org/2015/benchmarks.php`, Accessed: 2015-05-06.

[29] TLDP. Interrupt handlers: Linux kernel module programming guide. `http://www.tldp.org/LDP/lkmpg/2.6/html/x1256.html`, Accessed: 2015-05-06.

[30] C. Wang, M. Said, and A. Gupta. Coverage guided systematic concurrency testing. In *International Conference on Software Engineering*, pages 221–230, 2011.

[31] C. Wang, Y. Yang, A. Gupta, and G. Gopalakrishnan. Dynamic model checking with property driven pruning to detect race conditions. In *International Symposium on Automated Technology for Verification and Analysis*, pages 126–140, 2008.

[32] Y. Yang, X. Chen, G. Gopalakrishnan, and C. Wang. Automatic discovery of transition symmetry in multithreaded programs using dynamic analysis. In *International SPIN workshop on Model Checking Software*, pages 279–295, 2009.