


Parallelizing the Top-Down Solver for Faster Static Analysis

Parallelisierung des top-down Solvers für schnellere statische Analyse

Felix Kraye 

Technical University of Munich

 felix.kraye@tum.de

October 27, 2024

Abstract — Abstract interpreters get slow for larger programs since solving large constraint systems through fix-point iteration is time-consuming. In this report, we aim to accelerate the static analysis of programs by parallelizing a top-down fix-point algorithm. We implemented three different approaches of parallel solvers in the GOBLINT static analyzer. The first solver spawns multiple threads at the beginning that independently search for variables to iterate. The other two rely on a modification of the constraint system that explicitly indicates which parts can likely be solved in parallel. These two solvers differ in how they share data: one solver uses a shared data structure, while the other one employs local data structures for each thread that are merged in the end.

Benchmarks of the three solvers compared to a baseline show, that the speedup depends on the analyzed program. This is due to how we modified the constraint system to give parallelization hints. Overall we achieved a satisfying speedup for certain programs, where parallelization hints were well-placed, with the best-performing solver reaching a speedup of around 2.0 when using two worker threads for a number of programs. Additionally, another solver provided a slight speedup for two other programs. We were not able to find a correlation between the size of the analyzed program and the speedup achieved.

Our work provides a foundation for further research in this area, focusing on improving the parallelization hints or developing strategies to select a solver based on the program to be analyzed.

1 Introduction

Static analyzers examine the source code of a program to find semantic properties without having to execute it. To gain information about a program, many analyzers —particularly abstract-interpreters— compute abstract values for the different program points that over-approximate the set of possible concrete states. This is done by generating a system of constraints, where the variables are program points, possibly to-

gether with some calling context. The instructions in the code give rise to constraints describing how they affect the abstract state. When the system of constraints is generated, the analyzer computes a solution through fix-point iteration. This means that from an initial non-satisfying assignment of values to variables, these values are updated according to the constraints until a fix-point is reached, and all constraints are satisfied. Well-known algorithms for fix-point iteration are round-robin iteration and the work-list algorithm. Round-robin algorithms can, however, only be used for finite constraint systems and the work-list algorithm usually requires static dependencies. In contrast, the top-down solver recursively explores a possibly infinite constraint system on demand and tracks dynamic dependencies on the fly. It aims to find solutions for a defined set of interesting variables, e.g., the end node of the main program. For that, it recursively computes stable values for variables that are needed to calculate the values for the set of interesting ones. If the value for a stable variable is needed, it can just be looked up and does not have to be calculated again. However, if a new value for a variable is computed and updated during the solving process, it is necessary to destabilize all variables that depend on this updated variable. This means that the values for the now destabilized variables have to be evaluated again as they have to factor in the new value of the updated variable. These dependencies of variables are detected dynamically during the solving process and are saved in a data structure. Further data structures are used to keep track of stable variables and their values. The time of analysis grows together with program size. For example, an analysis of SQL lite 3 takes multiple hours with the GOBLINT static analyzer. Calculating a solution for the constraint system takes up a significant part of the overall analysis time. Thus, improving the performance of a solver concerning computation time seems a promising way to improve the speed of the whole analysis. An idea to approach this issue is to equip the top-down solver with a way to execute tasks in parallel and find steps of the solving process, where

a speedup can likely be achieved through parallelization. For example, a variable can depend on multiple other variables, e.g., when a thread is created and the thread function has to be analyzed as well as the rest of the program. In this case, one can imagine, that stable values for the variables corresponding to the thread function could be computed parallel to the variables corresponding to the main program, and the computation time could be reduced. As our main contribution, we aim to implement a parallelized top-down solver for the GOBLINT analyzer.

This report is structured as follows: In section 2 we introduce side-effecting constraint systems as they are used in GOBLINT and explain how a single-threaded top-down solver works. In section 3 we present our contributions. We explain how we adapt the constraint system to indicate for which variables parallelized solving can be useful. Furthermore, we implement a thread-safe data structure in this section. In subsection 3.3 we present three different concepts of parallelized solvers and implement them in GOBLINT as our main contribution. We evaluate our implementation by comparing the runtimes of the solvers in section 4. Lastly we give our conclusions and ideas for future work in section 6

2 Background

GOBLINT is a static analyzer for C programs for detecting bugs and errors. Since it computes an over-approximation of the program behavior, it is also suited to show correctness. Thus, if GOBLINT concludes that a program does not contain certain bugs, we can be sure that this holds. To prove certain properties about the program to be analyzed, it computes abstract states for each program point. These states describe all possible concrete states, the program can be in at that point in a sound manner. GOBLINT finds the abstract states by solving a constraint system over the program points. This constraint system is generated from the control-flow graph (CFG) according to the specifications of the type of analysis to be performed. The solution of the constraint system is a mapping from program points to abstract states, from which properties about bugs, errors, and correctness can be deduced. GOBLINT uses a fix-point solver to compute a satisfying solution.

2.1 Side-effecting constraint systems

In this section, we describe the constraint systems as they are generated and solved in GOBLINT in more

detail. The variables of the constraint system correspond to program points of the CFG. If the program is analyzed context sensitively, each variable additionally has a context from an arbitrary set of contexts and thus, multiple variables can exist for the same program point with different contexts. This allows to analyze function calls from different calling contexts separately for increased precision. However, the additional context potentially makes the space of variables for the constraint system infinitely large as the context domain might be infinite. Even though this sounds bad at first, since a non-trivial solution to an infinite constraint system cannot be computed in finite time, this is not an issue. In the case of program analysis, the solution to the constraint system does not have to include a satisfying mapping to all variables, but only to those reachable from a set of one or more *interesting* variables. In the case of program analysis in GOBLINT this is usually the variable corresponding to the program point at the end of the main function. The value domain of the constraint system is a set of abstract states, that varies according to the specification of the analysis to be performed. In any case, it has to be a complete lattice with a well-defined partial order, meet and join operations as well as a largest (top) and a least (bottom) value. Furthermore, to ensure termination of the constraint solver, widen and narrow operations are required. The constraint system now defines a constraint function for each variable. We refer to this function as the “right-hand side (rhs)” of a variable. The constraint system requires the variable to hold a value that is greater or equal to the value computed by the rhs with respect to the partial order of the value lattice. The rhs of a function can depend on other variables in the constraint system. We say, it *queries* another variable for its value and uses that for its calculations. A peculiarity of the GOBLINT analyzer is that it uses *side-effecting* constraint systems. This means that some rhs trigger *side-effects* to other variables. When a side-effect is triggered, it posts a value to another variable in the constraint system, i.e., the variable that receives the side-effect has to hold a value, that is greater or equal to its previous value and the value received through the side-effect. In GOBLINT this is for example used for function calls to post the state from the caller to the variable at the start of the called function. For GOBLINT it is ensured, that all variables that receive side-effects only have a trivial rhs, i.e., one that is only a constant initial value. We can think of the constraint system as a graph, where the program points are the nodes and the edges rep-

represent queries and side-effects from the **rhs**, i.e., if the **rhs** of variable x queries variable y , we draw a **query** edge from x to y . Similarly, a side-effect from x to y results in an **side-effect** edge from x to y . The **query** edges span a graph very similar to the CFG with flipped edges since the **rhs** for a variable usually queries the variable corresponding to the program point before it and applies the abstract effect of the instruction between the two program points to the queried value. We note here, that certain **rhss** in GOBLINT query a variable but discard the result, i.e., the value from the queried value is not used in the calculation of this **rhs**. This is done to force that a satisfying solution for this and all variables it depends on has to be calculated. Furthermore, this forces side-effects from the **rhss** of these variables to be triggered at some point. Concretely, queries whose results are discarded are used in the constraints for thread creations. When a thread is created at the CFG edge from variable x to y , the **rhs** of y queries not only x but also the return variable that corresponds to the program point, where the function of the created thread returns. The value of this return variable is not used to compute the **rhs**-value of y , but it forces the computation of a satisfying solution for the variables corresponding to the thread. We can define a sub-system of the constraint system originating at the return variable of the created thread. Besides the return variable, the sub-system of the thread contains all variables that are queried directly or indirectly in the **rhs** evaluation of the return variable. In general, this sub-system is disjunct from the rest of the constraint system with respect to **query** edges. However, there is a special case if a function is called from a created thread and another location in the program. If additionally the contexts of these calls are identical, two or more sub-systems can overlap in the variables of the function call with the same context. In the case of context-insensitive analysis, the contexts are always identical. Note that in any case side-effects happen across sub-systems.

2.2 Top-down solver

GOBLINT mainly employs top-down solvers (TDs) to solve its constraint systems. The most optimized of these implements many improvements and additional features, including but not limited to solving with widening and narrowing phases and incremental analysis. In this section, we introduce a simplified version of this solver that is the basis for the parallelized solvers we introduce in subsection 3.3. The task of the solver is to compute a solution to the constraint sys-

tem. Since the domain of the constraints is a complete lattice, this can be achieved through fix-point iteration. This means that all variables are assigned the bottom value of the lattice, after which the variables are repeatedly *iterated* until a fix-point is reached. In each iteration of a variable, its **rhs** is evaluated for a new value. If the new value is not the same as their current one, the variable is assigned a value greater than or equal to both the current and the new value. Once all variables have a that is at least as great as their **rhs**, a fix-point is reached. Different solvers employ different strategies to decide which variable should be iterated next. Before explaining the strategy of the TD we introduce some important properties the solver tracks for the constraint variables:

- **value:** The current value assigned to this variable from the value domain
- **called:** A called variable is currently in the stack of variables to be iterated.
- **stable:** A stable variable currently satisfies its constraints. This holds as long as none of the variables it depends on change their value.
- **influences:** A transitive relation between two variables x and y . “ x influences y ” means that the value of variable y depends on the value of x , because the **rhs** of y queries x .
- **wpoint:** If a variable is updated while it is a widening-point (wpoint), widening is applied when computing the new value. This is necessary to ensure the termination of the solver.

The TD starts from one variable from the set of interesting variables, for which it is supposed to compute a value and starts iterating it. An iteration of a variable means, first setting it called and stable, then evaluating its **rhs** and updating its value if necessary and lastly removing it from the set of called variables. If during the evaluation of the **rhs** of a variable x , a value from another variable y is queried that is neither called nor stable, the iteration of x is paused and y is iterated until a stable value is found for it. If y is called or stable, the query just returns its current value which is then used in the **rhs** evaluation of x . Furthermore, for such a query it is tracked that y influences x . This results in the solver recursively following the **query** edges of the constraint graph until the solver reaches a variable, whose **rhs** does not query any other variable, or a variable already marked as *called*. From this variable,

the solver backtracks the **query** edge. In each backtracking step, the current value of the queried variable is used in the **rhs** evaluation of the querying variable. The querying variable is then updated with the result of the **rhs** before taking the next backtracking step. When the backtracking reaches the initial variable from which the solving process was started, this variable is stable. This process is repeated with the other variables from the set of interesting variables until all have a stable solution. When the value of a variable is changed during an iteration, the variables that depend on this changed variable have to be destabilized. This is necessary as these variables need to take the new value of the changed variable into account. The destabilization is done by recursively following the influences-relation of the updated variable and setting all encountered variables to *not stable* while removing the used paths from the influences-relation to avoid infinite loops during destabilization. Destabilizing is necessary because the influenced variables have to be iterated again since their **rhs** might change as it queries the changed variable. If the solver encounters a *called* variable while exploring the **query** edges, this variable is marked as a *wpoint*. This happens for program loops since the variable at the loop head queries the last variable inside the loop next to the one before the loop. A variable being a *wpoint* means, that when this variable has to be updated during an iteration, i.e., the value from the **rhs** is not equal to the current value, widening is applied. This means that the current value is not joined with the new one, but widened by it, which ensures the termination of the solver. Lastly, we note that side-effects are computed as soon as they are encountered during the evaluation of an **rhs**. If a variable receiving a side-effect has to be updated, all variables influenced by it are destabilized in the same manner as described above.

3 Methodology

Before explaining our contributions in detail, we note here that the framework of the GOBLINT analyzer had to be adapted so that the parallelized solvers do not run into concurrency issues. We protect certain functions with a mutex such that they cannot be called multiple times in parallel. Amongst other examples, this is necessary for functions related to printing output as otherwise multiple messages can get mixed together. Furthermore, we make certain data structures thread local, meaning, that the worker threads do not share these data structures, but instead, each thread has a

local copy of it. For example, this applies to the data structure responsible for timing different parts of the analysis.

3.1 Constraint systems with create edges

We want to give the solvers hints, which variables can likely be iterated in parallel. As we explained in subsection 2.1, in GOBLINT a variable can have a **rhs** that queries a variable but discards the result. This is a suitable location for parallelization, as a variable whose value is not needed anyway can be iterated in parallel to the iterations of the main solver. A sequential solver has to pause the current **rhs** evaluation to compute a to-be-discarded solution for the queried variable, while a parallel solver can continue the **rhs** evaluation while computing a solution for the queried variable in parallel. To integrate this idea into our notion of a constraint system, we introduce **create** edges. **Create** edges have a target variable x . Similar to **query** edges, they require the solver, to solve the target variable for a satisfying solution. However, they also tell the solver, that the variable that is the origin of this **create** edge does not depend on the result. Concretely, we introduce **create** edges at points in the program, where a thread is created. The target variable is the return point of the created thread. Thus, it is ensured that a solution for the sub-system of the thread is computed.

3.2 Lockable Hash-table

During the solving process, the solvers store and update a mapping of variables to values and track further information about the variables. For this, the existing single-threaded top-down solver uses hash-tables, since it allows for fast access and updates. However, for a multithreaded solver, the access to these data structures has to be guarded by a mutex to allow only a single thread to read and write the information belonging to a variable during a critical section. A straightforward idea is to lock the whole hash-table with a single mutex. This however takes away much potential for parallel work, since large parts of the solver are spent in critical sections. We propose a *lockable hash-table* that splits the mapping it stores into a set number of n buckets, where each of the buckets can be locked with a mutex. The bucket, in which the value for a variable is stored is determined by a hash function, e.g. the mapping for variable x is stored in bucket $\text{hash}(x) \bmod n$. This allows for parallel work

on multiple variables without waiting, as long as these variables are placed in different buckets.

3.3 Parallelized top-down solvers

In this section, we describe the workings of three different parallel solvers as we implemented them in the GOBLINT analyzer. The concepts for these solvers were given to us and discussed in private communication with Helmut Seidl, Michael Schwarz and Ali Rasim Kocal [5]. The basis for all three solvers is the TD we described in subsection 2.2. We note here that the stealing TD does not use the `create` edges we introduced in subsection 3.1, while the shared-memory TD and the disjunct TD need these edges to perform work in parallel.

3.3.1 Stealing TD

The idea of the stealing TD is to start multiple TDs in parallel. The hope is that they find different spots in the constraint system where multiple iterations are needed. The parallel solvers work independently of each other but have a shared data structure for the value of the variables and for tracking which variables are called and which are stable. If one solver at some point queries a variable that was already iterated by another one, it can use the result from that solver. These variables are then *stolen* from the solver that worked on them previously. For this concept to work, the solvers are organized in a hierarchy. Thus, each solver has a distinct priority (`prio`) with higher prioritized solvers stealing variables from lower prioritized solvers. Furthermore, the concept of calledness and stability changes for the stealing TD. A variable is no longer either called or uncalled, but it is assigned a called-`prio` coinciding with the `prio` of the solver that *owns* it, with a special value indicating, that a variable is not owned at all. Solvers take ownership of a variable when they first encounter it during a query and it is not owned by a higher prioritized solver. If the variable is assigned a lower called-`prio`, the solver *steals* it, if not, the solver marks the variable as a `wpoint` and uses its current value. After finding a stable solution for a variable, the solver gives up ownership of that variable. It is not given back to the solver from which it was stolen, instead, it is not owned by any solver at all. Similarly, variables are assigned a stable-`prio` according to the solver that computed the stable value. Solvers see variables with their own or a higher stable-`prio` as stable, but variables with a lower stable-`prio` as unstable and iterate those again.

If a variable gets stolen from a solver during a `rhs` evaluation, the solver does not update the mapping of the variable. The current value, calledness and stability are tracked in a shared data structure, for which a *lockable hash-table* we introduced in subsection 3.2 is used. The solvers track the *influences*-relation and the set of *wpoints* independently, i.e., each parallel solver has a private data structure to store this information. Destabilizing works similarly to the single-threaded TD, i.e., the influences-relation is followed recursively and encountered variables are set to unstable—in this case the special value denoting that no solver sees the variable as stable.

Revival We note here, that lower prioritized solvers often lose all their owned variables before the whole solving process is completed. In that case, they terminate when they do not find any from their view uncalled variables to work on. This is especially an issue for larger programs, where after a while the most prioritized solver has stolen all variables and works like a single-threaded solver after that point. To combat this issue, we introduce revival and work-finding for terminated solvers. We still start all solvers with their respective priorities. However, when a solver returns because it is unable to find an uncalled variable, it enters a work-finding phase instead of terminating instantly. During this phase, the solver traverses the *query* edges of the constraint system until it finds a variable that is not owned or been set stable by a higher prioritized solver. This traversal is randomized in a way that it alternates between a depth-first and a breadth-first approach non-deterministically. The randomization is supposed to prevent the solvers from getting caught up in the same area of the constraint system. It then uses this variable as an entry point to iterate it and all variables it depends on for a stable solution. The solvers alternate between the work-finding phase and solving an unstable variable for a solution. Once the highest prioritized solver finishes, a stable solution has been found and the other solvers are notified to terminate.

3.3.2 Shared-Memory TD

In contrast to the previously introduced stealing TD, the shared-memory TD does not start multiple parallel solvers in the beginning but uses a thread-pool instead. The shared-memory TD adds a task to solve for a variable x every time a `create` edge for variable x is encountered in a `rhs` evaluation. These tasks are then

executed in parallel. This solver tracks all information about the variables in a shared data structure for all tasks. Concretely a *lockable hash-table* introduced in subsection 3.2 is used. The shared-memory TD essentially works like the single-threaded solver. However, when the solver encounters a **create** edge for the target y in the **rhs** of a variable x , it adds a new task solving for variable y and continues with its current iteration of x . Recall, that we introduced **create** edges at the points in the constraint system, where a thread is created. We note here, that the single-threaded solver would first find a stable solution for y before continuing working on x . Additionally, it would track that y influences x because a **create** edge is like a **query** edge for the single-threaded solver. If a variable in the sub-system of a thread receives a side-effect that changes its value, this variable and the ones it influences are destabilized. For the single-threaded solver this results in a destabilization across the **create** edge. This is not the case for the shared-memory TD. To ensure that the sub-system of the thread is reiterated and all variables have a stable solution in the end, the target variable of a **create** edge is marked as a **root** variable. In the case, that a root variable is destabilized, a task is added that reiterates the corresponding sub-system for a stable solution.

3.3.3 Disjunct TD

The disjunct TD also uses a thread-pool to which it adds tasks when encountering a **create** edge. Unlike the previous solvers, all solver data is stored in separate data structures per task. This includes the mapping of variables to current values, meaning that different tasks can have different values for the same variable. However, the need for locking when accessing these data structures is removed. We recall an observation from subsection 2.1, that the sub-systems originating from the return points of threads are in general disjunct with respect to **query** edges. An exception to this rule is the call to a function in the same context from different sub-systems. Since we placed **create** edges exactly at the return points of threads, this means, that in general the tasks from the disjunct TD in general solve disjunct sub-systems. If this is the case, the set of variables, that the tasks iterate is disjunct as well. Otherwise, multiple tasks iterate the same variables independently of each other, creating overhead in the form of duplicate work and reducing the effectiveness of parallelization. In any case, side-effects can update values across these sub-systems. To handle these, we implemented a dedicated **Side** structure. All tasks

post their new side-effects to this structure and obtain side-effects of other tasks from it. The **Side** structure contains a list of all side-effects. This list contains the target variable and the value of the side-effect. When a task now encounters a side-effect, it only adds the target variable and the value to the list. Side-effects are only processed by a task, right before it evaluates a **rhs**, or right before it terminates. In these cases, the task checks the **Side** structure whether there are new side-effects it has not processed yet. For this, each task tracks the number of side-effects it already processed and compares this to the length of the list of side-effects. Since the list is only expanded and never altered in any other way, the task can easily find the unprocessed side-effects and handle them. This possibly causes destabilization of variables that the task then has to reiterate again. Access to the **Side** data structure has to be locked to avoid race conditions. If a task is done, i.e., it has found a stable solution for its whole sub-system, it is suspended. This means, that all of its data is stored and it can be revived easily. Whenever a new side-effect is registered to the **Side** structure, all suspended tasks are revived, since they need to process the new side-effect. The whole disjunct TD only terminates once all tasks are suspended. After all tasks have terminated, the different mappings from variables to values of the separate tasks have to be combined to a single mapping. In the case, that the sub-systems are actually disjunct, this is not an issue. The mappings can just be combined without conflicts since the mappings only share side-effected variables that have trivial **rhss** as mentioned in subsection 2.1. Since all tasks processed the same side-effects, the values of these variables are the same for all. In the case that sub-systems overlap, there can however be conflicts. Since the order in which side-effects are observed between iterations is not deterministic, and it can differ for the different tasks, one task might detect a widening point one or more iterations earlier or later than another task. Thus, conflicts can arise, when one task applies widening for a certain variable, while another task does not widen for the same variable. Generally, it suffices to compute the meet of conflicting values. In some cases, however, the result no longer satisfies all constraints. We consider a variable x that was only iterated by a single task t , but its **rhs** queries a different variable y , for which multiple tasks computed different values. The value of x satisfies its **rhs** for the value of y that was computed by the task t . After the meet of the different values for y is computed, the result might be larger than the y value

from task t . However, when the `rhs` of x is evaluated with the meet of all y s, the result might be larger than the value of x . In this case, x does no longer satisfy the constraints. This issue is still to be solved.

Sharing of side-effected variables In the following, we introduce a variant of the previously described disjunct TD that uses a shared data structure to store the values of exactly those variables that receive side-effects. For this variant, we make use of the fact that in GOBLINT all variables that receive side-effects only have a constant initial value as their `rhs`. We call these variables *side-effected variables*. These variables are only ever updated by side-effects and never through an iteration. Thus, we can have a *lockable hash-table* inside the `Side` structure that directly contains the mapping from the side-effected variables to their value. The tasks of the solver do not contain this part of the mapping. When a side-effected variable is queried, the value is looked up in the central `Side` structure instead of the local solver mapping. When a side-effect is encountered by a task, it updates the shared mapping in the `Side` structure if the new value is not less or equal to the current one. If this is the case, it also adds the target variable to the list of side-effects. In this variant of the disjunct TD, the values of the side-effects are not tracked in this list. This is because the other tasks only need to notice, which side-effected variables are updated but not the values of the updates. When a task checks the `Side` structure before evaluating a `rhs` or before terminating, it looks up which side-effected variables were updated since it last checked and destabilizes those variables and all variables they influence. The main difference to the original disjunct TD is, that the processing of side-effects is moved from the individual tasks into the central `Side` structure. This avoids all tasks performing the updates to side-effected variables individually. Ideally, the sub-systems solved by the different tasks are disjunct, and thus each task only ever queries a subset of all side-effected variables. With the shared mapping in the `Side` structure, the tasks do not need to process and track the values of side-effected variables they never query.

4 Evaluation

We want to evaluate and compare our implementations of the three different parallel solvers in GOBLINT. Before comparing their performance with respect to evaluation time, we investigate if the parallelization

results in a loss of precision. The source repository of GOBLINT contains more than 1400 regression tests. These are small programs focused on testing various edge cases of the different features of the analyzer. We executed all regression tests five times with each of the parallelized solvers from subsection 3.3. From the results, we observe that for less than 0.5% of the tests, the parallelized solvers were less precise than the single-threaded solver we introduced in subsection 2.2. This loss of precision was non-deterministic, i.e., in some runs, there was no loss of precision observed. We do not notice a difference in the number of failing tests for the three different parallelized solvers. Furthermore, when analyzing real-world programs as we do in the following sections, no loss of precision was observed in the warnings GOBLINT produced. These warnings include but are not limited to notifications about thread-(un)save memory locations and dead lines of code. Thus, we believe that the parallelized solvers can be considered as precise as the single-threaded solver we compared them against. We note here that GOBLINT typically uses an improved version of the single-threaded TD from this report and thus is even more precise in 32 of the regression tests.

4.1 Analysis speed

To evaluate the speed of the different solvers, we analyze several programs and compare the time of the analysis. We use two groups of programs for this evaluation. The first is a selection of the GNU core utility programs (`coreutils`) [3]. Before analysis, these programs were combined, i.e., a single source code file was produced, that contains the original program together with the dependencies of other included files. The exact code files used can be found in the GOBLINT benchmark repository [4] that contains benchmark programs for the analyzer. The second group consists of programs from the `pthread` folder of the GOBLINT benchmark repository. These are programs that use threads more extensively than the `coreutil` programs, i.e., they use threads slightly more often and the spawned threads perform much more work. We included these programs since the shared memory TD and the disjunct TD depend on threads being spawned in the analyzed program to add tasks that are solved in parallel.

For comparison, we analyze each program once with the single-threaded TD from subsection 2.2 as a baseline and twice with each of the parallelized TDs from subsection 3.3, where one run was done with only the main thread working and one run with two

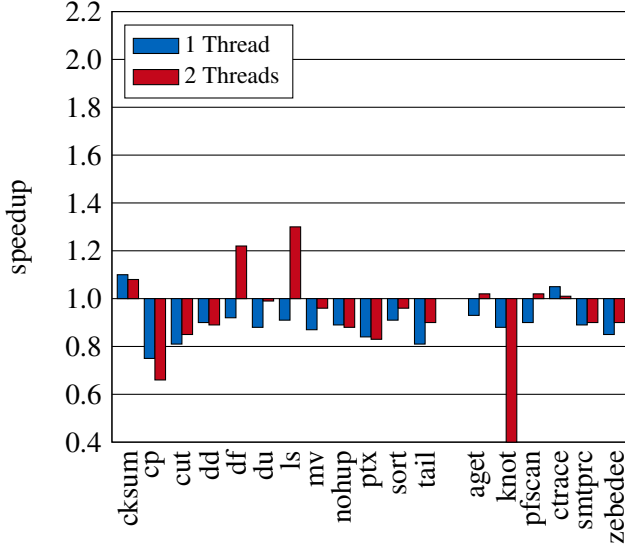


Figure 1 Speedup of the stealing TD. Runs with a single worker thread are shown in blue, while runs with two threads are represented by red bars. The twelve leftmost programs make up the coreutil group, while the six rightmost programs belong to the pthread group of programs. The run with two threads for the `knot` program resulted in a timeout of 900s

worker threads. We do this, so we can compare the analysis time between the single-threaded TD and a certain parallelized TD in more detail, e.g., attribute differences in analysis time to the inherently different algorithms of the solvers or to the parallelization. Since the programs are different in size and general analysis time, we mainly focus on the *speedup* as a metric. We calculate the speedup of the parallelized solvers with the following formula:

$$\text{speedup}(\text{parallel solver}) = \frac{\text{time}(\text{baseline solver})}{\text{time}(\text{parallel solver})}$$

With this, we get a value larger than 1, if the parallelized solver is faster and a value smaller than 1 if it is slower. The raw timing values are listed in the appendix in Table 1.

When investigating the results for the stealing TD from Figure 1, we notice, that in general, this solver is slower or just as fast as the baseline. Furthermore, the difference between running this solver single-threaded or with two worker threads is minimal. Interesting are the programs `df` and `ls` where the stealing TD achieved a speedup of 1.2 and 1.3 respectively. However, for the program `cp` it was significantly slower than the baseline. Unfortunately, we are not able to explain the timeout for the `knot` program, as it terminated in a reasonable time when analyzed with a debugging configuration. We note here, that we only

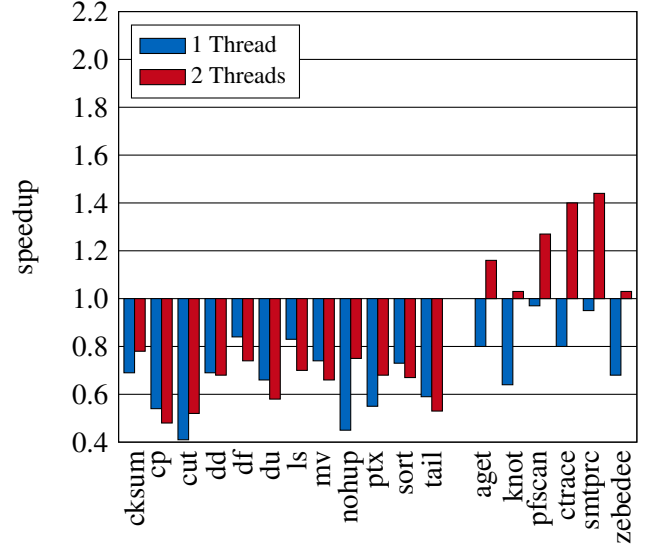


Figure 2 Speedup of the shared memory TD. Runs with a single worker thread are shown in blue, while runs with two threads are represented by red bars. The twelve leftmost programs make up the coreutil group, while the six rightmost programs belong to the pthread group of programs

evaluate the stealing TD with the addition of revival, since the stealing TD is strictly faster with revival.

Figure 2 shows the speedup results for the shared memory TD. We see this solver being significantly slower for all the coreutil programs, with the double-threaded configuration only outperforming the single-threaded one in three cases. However, the pthread group of programs paints a different picture. As expected, the single-threaded configuration is slower than the baseline for these programs. When allowing the shared memory TD to use a second worker thread for parallel work, it achieves significant speedup for four of the six programs compared to the baseline. This speedup even reaches 1.4 for two of these.

Through our benchmarks, we noticed that the variant of the disjunct TD where side-effected variables are shared is slower when analyzing the coreutil programs but faster for the pthread programs compared to the original disjunct TD. The original variant is approximately as fast as the baseline for the coreutil programs but achieves a speedup of around 1.8 for three of the pthread programs. Like the previously discussed shared memory TD, the disjunct TD is built around the `create` edges we inserted into the constraint system. Therefore, we are most interested in the performance on the pthread programs. In the following, we focus on the speedups for the variant that shares side-effected variables, since it performed better for the pthread programs. The speedups for the different programs are

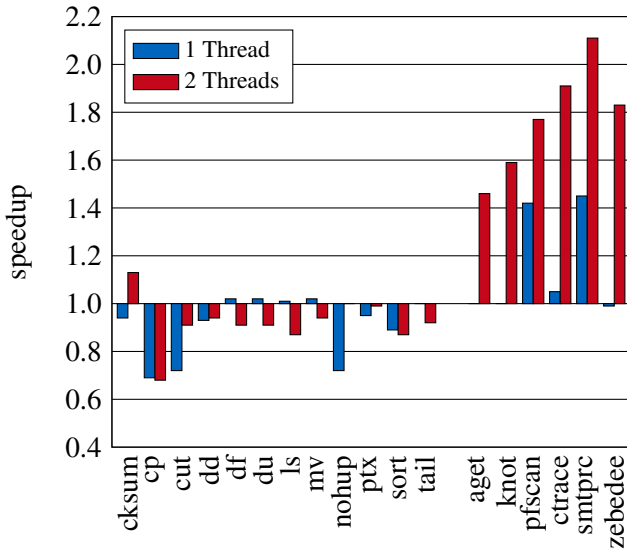


Figure 3 Speedup of the disjunct TD with sharing of side-effected variables. Runs with a single worker thread are shown in blue, while runs with two threads are represented by red bars. The twelve leftmost programs make up the coreutil group, while the six rightmost programs belong to the pthread group of programs

illustrated in Figure 3. As mentioned, for the coreutil programs we see the disjunct TD being slower than or about as fast as the baseline. For the pthread group of programs, however, this solver provides significant speedup when run with two worker threads. Interestingly enough, the speedup for the `smtprc` program exceeds 2.0, which is in theory the highest possible speedup that can be achieved by doubling the number of worker threads. We believe the reason for this peculiarity is the fact, that the solver data is distributed over multiple tasks, reducing the size of the individual data structures. The frequent accesses to these smaller data structures are likely quicker. Thus, it is plausible that the disjunct TD can achieve a speedup higher than 2.0 when having two threads available. The solver data being distributed in smaller data structures would also explain, how the disjunct TD can be faster than the baseline even when it is just run with a single worker thread like it is the case for the `pfscan` and the `smtprc` program.

To investigate if there is a correlation between program size and speedup, we computed the Pearson correlation coefficient of the speedup a solver achieves with two worker threads and the number of live lines of code in the program. These are the lines that correspond to program points that are reached during the solving process. For the stealing TD the correlation coefficient amounts to 0.03. To compute this coefficient,

we included all programs except for `knot`, where the solver timed out. For the other two solvers, we only included the pthread group of programs, since these are the programs, where the `create` edges are suitably placed for these solvers. The correlation coefficient is 0.04 for the shared memory TD and 0.12 for the disjunct TD. This means, there is little to no correlation between program size and speedup achieved. Interestingly enough, when we compute the correlation coefficient with respect to the coreutil programs, we get coefficients of -0.29 for the shared memory TD and -0.68 for the disjunct td. This means, that there is a small but noticeable negative correlation for the programs, that are not suitable to be analyzed with these solvers. This indicates that these solvers scale worse than the baseline for larger programs that make little or no use of threads.

Overall, we see the disjunct TD achieving a significant speedup on the pthread group programs. While the shared memory TD also provides a speedup for these programs, the disjunct TD outperforms it in almost every case. The stealing TD, implementing a very different approach that does not depend on `create` edges in the constraint system, does not provide a speedup for the pthread programs, but it is the only one of the three solvers that achieves a noticeable speedup for any of the coreutil programs, i.e., the `df` and the `ls` program.

5 Related Work

In this section, we discuss a few approaches other researchers proposed to parallelize fix-point algorithms or static analysis in general. Blaß and Philippsen [2] propose a fix-point algorithm, where a GPU is used to propagate predicates between CFG nodes in parallel. Each node is assigned to one of the usually more than 1000 GPU threads. The nodes repeatedly query the states of their predecessors in the CFG and use the result to update their own state. The algorithm does not prevent data races through synchronization but detects and repairs inconsistencies caused by races. A heuristic is used to decide when to return to the CPU and check if a fix-point has been reached. If not, the GPU continues to work on a solution. Though this approach is promising and achieves a good speedup, it requires dedicated hardware to run. Furthermore, the variables of the constraint system have to be known beforehand, making it hard to apply this algorithm to context-sensitive analysis as it is implemented in GOBLINT. Aiken et al. [1] present the Saturn program anal-

ysis system. Saturn is summary-based which means, that summaries of individual functions are computed. A caller then uses this summary of a function to compute the effect of a call. This approach, where functions are analyzed separately without factoring in a calling context is easily parallelizable, being only limited by the dependencies between functions. Aiken et al. use this fact to analyze functions in parallel with great efficiency. However, to analyze whole programs like GOBLINT does, this approach is less suitable, as it requires knowledge of the call graph beforehand. The approach by Van Es et al. [6] focuses on parallelizing modular analyses. This is similar to the Saturn system, as the program is divided into modules independently of each other. These modules can be functions or threads. In contrast to the Saturn system, it uses a demand-driven top-down approach to decide which functions to analyze. Similar to our usage of the `create` edges, the algorithm by Van Es et al. generates a task to analyze a thread or a function. While the modules might only depend on each other through function calls, GOBLINT deals with side-effects across function and thread boundaries, which we had to take into account for our parallelization approaches.

6 Conclusions

In order to explore the potential of speeding up abstract interpretation by parallelizing the fix-point computation, we implemented three approaches in the GOBLINT analyzer and evaluated them with respect to analysis time. We introduced `create` edges to the side-effecting constraint systems used in GOBLINT to let `rhss` indicate where parallelized solving is reasonable. Furthermore, we adapted the framework of the analyzer, such that the parallelized solvers could run without concurrency issues, and implemented a lockable hash-table to allow for concurrent updates on that data structure. As our main contribution, we implemented three concepts of parallelized top-down solver in GOBLINT and evaluated them on different types of programs.

From the Evaluations in section 4 we conclude, that the three parallelized solvers can be considered as precise as the single-threaded solver in most cases. The speedup of parallelized solving depends a lot on the program to be analyzed and the selected solver. The stealing TD was the only solver that was able to achieve a noticeable speedup on two of the coreutil programs, that only use multithreading to a minimal extent. In contrast to that, the shared memory TD and the dis-

junct TD perform worse on these programs but provide noticeable speedup on the pthread group of programs. This is to be expected, however, since these two solvers depend on the `create` edges we placed at the creation of threads. As for a comparison between these two solvers, we saw the disjunct TD performing better than the shared memory TD in general. Lastly, we note, that we were not able to find a correlation between the size of the analyzed program and the speedup achieved for any of the solvers.

6.1 Future Work

We think that we have not yet exhausted the potential of parallelization. Besides thread creation, there are other locations in the constraint system, where parallelization can potentially be useful, e.g., when at a function call it is not certain which function is called and both have to be evaluated in parallel. Thus, we want to introduce a different kind of `create` edge at such positions. In contrast to the existing one, this `create` indicates that the value of the target variable is queried at a late point. Furthermore, we want to investigate the programs more closely, where certain solvers achieve a speedup. We hope to identify certain characteristics that we can use to automatically decide which solver is likely best suited for analyzing this program quickly. Additionally, we want to run benchmarks with more than two worker threads, to investigate for which type of programs a higher number of worker threads might be beneficial and for which type of programs this creates more overhead without providing a speedup.

Abbreviations

rhs right-hand side

CFG control-flow graph

TD top-down solver

prio priority

References

- [1] Aiken, A., Bugrara, S., Dillig, I., Dillig, T., Hackett, B., Hawkins, P.: An overview of the saturn project. In: Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering. pp. 43–48 (2007)
- [2] Blaß, T., Philippsen, M.: Gpu-accelerated fixpoint algorithms for faster compiler analyses. In: Proceedings of the 28th International Conference on Compiler Construction. pp. 122–134 (2019)
- [3] GNU: Coreutils - gnu core utilities. <https://www.gnu.org/software/coreutils/>, last Accessed: 2023-02-01
- [4] Goblint: Goblint benchmark suite. <https://github.com/goblint/bench>, last Accessed: 2023-02-01
- [5] Seidl, H., Kocal, A.R., Schwarz, M.: private communications, repeated communication since March 2024 until October 2024
- [6] Van Es, N., Stiévenart, Q., Van der Plas, J., De Roover, C.: A parallel worklist algorithm for modular analyses. In: 2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM). pp. 1–12. IEEE (2020)

Appendix

Program	baseline	stealing 1 thread	stealing 2 threads	shared memory 1 thread	shared memory 2 threads	disjunct 1 thread	disjunct 2 threads
coreutils							
cksum	1.18	1.07	1.09	1.72	1.51	1.25	1.04
cp	11.91	15.82	17.94	21.97	24.70	17.24	17.51
cut	9.13	11.32	10.74	22.35	17.64	12.73	10.05
dd	3.22	3.57	3.61	4.65	4.77	3.47	3.44
df	15.57	16.85	12.81	18.55	21.06	15.26	17.16
du	170.30	193.86	172.38	257.71	295.06	166.55	186.61
ls	166.09	181.77	127.30	201.16	236.66	164.94	191.92
mv	15.57	17.92	16.21	21.14	23.60	15.30	16.62
nohup	2.79	3.12	3.16	6.19	3.71	3.90	2.80
ptx	26.77	31.97	32.39	48.68	39.13	28.26	27.08
sort	10.75	11.81	11.18	14.69	16.15	12.10	12.38
tail	32.77	40.27	36.55	55.22	62.05	32.85	35.76
pthread							
aget	1.94	2.08	1.90	2.41	1.67	1.94	1.33
knot	3.49	3.96	900.00	5.49	3.39	3.50	2.20
pfscan	0.94	1.04	0.92	0.97	0.74	0.66	0.53
ctrace	3.24	3.09	3.20	4.03	2.31	3.09	1.70
smtprc	16.74	18.91	18.52	17.66	11.62	11.54	7.94
zebedee	54.28	63.78	60.48	79.51	52.94	54.80	39.27

Table 1 Timing values in seconds from the benchmarks. The stealing solver with two threads timeouted for the knot program at 900s.