Live-Variables within CPAchecker

Sebastian Ott
University of Passau
Email: ottseb@fim.uni-passau.de

Thomas Stieglmaier
University of Passau
Email: stieglma@fim.uni-passau.de

Thomas Ziegler
University of Passau
Email: zieglert@fim.uni-passau.de

***Abstract*—This work introduces a formalism for a configurable program analysis (CPA) that collects information about the liveness of variables in a given control-flow graph. Two configurations of this CPA and their usefulness are discussed. The evaluation is done on an implementation of the LIVE-VARIABLES CPA in CPACHECKER where other analyses can use the gathered liveness information in the abstract successor computation or during the precision adjustment phase. Our experiments show that the basic value analysis substantially profits from using liveness information.**

***Index Terms*—Live-Variables, CPA, Model-Checking**

I. INTRODUCTION

Model-checking is a technique to verify if a program meets a given specification. This can be done by analyzing the CFA (see Section II-A) of a program and checking if there is a path that leads to a state where the specification is violated.
The larger the CFA is, the more states are created during the analysis. And with the amount of variables in the CFA, the overall size of the created states increases.
In this case a live-variables analysis can help to save time and memory consumption by removing unneeded variables (due to reassignments, or no more reads) from the states.

II. BACKGROUND

A. Program Representation

A program is represented by a *control-flow automaton* (CFA) [BHT07]. This is a graph consisting of a set L of locations (nodes), that model the program counter pc, and a set $G \subseteq L \times Ops \times L$ that represents the control-flow and the initial location pc_0 (the program entry point). We allow only assumptions and assignments of arbitrary form as Ops.
A CFA may have several nodes with no outgoing control-flow edges, we call them **exit nodes**. Let X be the set of all variables in the CFA and C the set of all concrete states. The *concrete state* for a location assigns a value to each variable from the set $X \cup \{pc\}$. For every edge $g \in G$ the transition relation is defined by $\xrightarrow{g} \subseteq C \times \{g\} \times C$. By unifying all edges we create the complete transition relation $\rightarrow = \bigcup_{g \in G} \xrightarrow{g}$.
Reachability can now be defined as follows:
If a chain of concrete states $\langle c_0, c_1, ..., c_n \rangle$ exists, such that $\forall i : 1 \leq i \leq n \implies c_{i-1} \rightarrow c_i$ and there is a region r such that $c_0 \in r$, the state c_n is called reachable from the region r.

B. Live Variables

In this section we will first give a definition of liveness of variables. Afterwards we will explain liveness using an example in the C programming language.

```
1  int foo(int a, int b) {
3      int w;
4      int x = a;
5      int y = 5;
6      int z = x + 3;
8      if (b < 5) {
9          w = y + 2;
11     } else {
12         y = b - 1;
13     }
15     x = b;
16     z = x + y;
18     return z;
19 }
```

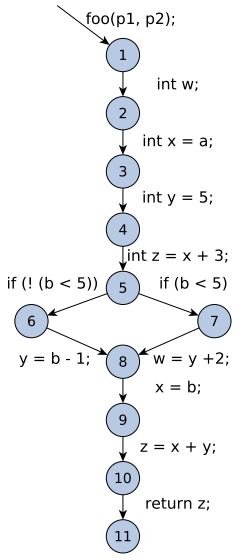

Fig. 1. Example program with corresponding CFA

Definition 1 (Liveness (adapted from [Ken81])) *A path in $CFA = (L, G, pc_0)$ is said to be x-clear if that path contains no assignment to the variable x. The variable x is **live** at point p in G if there exists an x-clear path from p to a use of x.*

The best way to implement an analysis following this definition is is a backwards data-flow analysis [Ken81]. If the CFA is traversed bottom-to-top, the first read of a variable indicates it becoming live, as it will be the last read of this variable in the normal program, while a write always indicates that a variable stops being live.
In this paragraph we will use the example from Listing 1 to explain liveness of variables. The variable `x` becomes live in line 4 as it is written there and will be read in line 6. After the read occurred, it stops being live as it is not read before it is written again. In line 15, `x` becomes live again because it is written and will be read in the next line. In contrast, the variable `z` does not become live in line 6 even if it is written there. But it will be written in line 16 and is not read in between line 6 and 16 so it becomes live in line 16 and stops being live in the next line where it is read. The variable `w` is written in line 9 but never read. So it is not live in the whole program. Function arguments have to be analyzed as well. Calling a function with an argument is a write on the local variable in which the values are stored. In this example those are the variables `a` and `b`. There is an implicit write on

those in line 1. While a is live up to the read in line 4, b does not stop being live on the first read in line 8, as it will be read in line 15 for sure and maybe in line 12 as well. The part where a variable is live ends at the *last* read of a variable before the next write. If no further write occurs it is live until the last read in the program.
As the liveness analysis is a data-flow analysis we merge the live-variables of all branches where the control-flow meets. This applies to if-statements for example. The liveness scope of the variable y is therefore from line 5 to line 16. This is because we do not know which of the branches of the if-statement is taken. As one can see by analyzing the liveness of variable x a variable can be live in multiple distinct parts of a program. Also an assignment of a value to a variable does not necessarily cause a variable to become live and reading a variable does not always stop a variable from being live.

C. CPAchecker

CPAchecker is an open-source software verification framework, based on the concepts of configurable program analysis and dynamic precision adjustment [BHT08]. There is already a set of CPAs like the Predicate Analysis CPA and Value Analysis CPA which can be used for analyzing programs written in C or Java. CPAcheckers interfaces allow the implementation of new CPAs as well as the composition of CPAs to create new analyses. More information can be found in related literature [BK11] and on http://cpachecker.sosy-lab.org.

III. Formalism

In this chapter we define a configurable program analysis (CPA)[BHT07] that collects the liveness information about each variable in a given control-flow automaton. It will be called Live-Variables CPA. It has to be used in combination with a location tracking CPA. Unlike other CPAs the Live-Variables CPA is a backwards analysis (see Chapter II). This has some implications on the initial states of the analysis:

- first, the initial state is not $\top$ but the empty set $\{\}$,
- second, the single initial state for the location pc_0 is replaced by a set of initial states:
  - one for each exit node of the CFA
  - and one for each endless loop (as in a CFA an endless loop has no outgoing edges, we would not be able to collect liveness information about variables inside such loops otherwise).

The next paragraph gives defintions for the parts of this CPA. The **abstract domain** $D = (C, \mathcal{E}, [\![\cdot]\!])$ consists of three parts, a set C of concrete states, a semi-lattice $\mathcal{E} = (E, \top, \sqsubseteq, \sqcup)$ and a concretization function $[\![\cdot]\!] : E \to 2^C$. The components of the semi-lattice are:

- a set $E = 2^X$ of sets of live variables, where X is the set of all variables in the CFA,
- a top element $\top = X$, representing a state where every variable is live,
- a preorder $\sqsubseteq \subseteq E \times E$ represented by the inverse subset relation $\subseteq$; for two abstract states e, e' we define $\sqsubseteq (e, e') = e' \subseteq e$.
- and a total function $\sqcup : E \times E \to E$, available through the union operator $\cup$ on sets; for two abstract states e, e' we define $\sqcup(e, e') = e \cup e'$.

The concretization function $[\![\cdot]\!] : \{c \in C\}$ assigns all concrete states to a given abstract state (for the Live-Variables CPA the abstract state has no influence on the possible amount of concrete states).
For the **transfer relation** $e \overset{g}{\rightsquigarrow} e'$ we define e' as $e' = (e \setminus \mathit{written}(g)) \cup \mathit{read}(g)$ where $\mathit{written} : G \to 2^X$ is a function that returns the set of variables that got assigned for a given edge, and $\mathit{read} : G \to 2^X$ is a function that returns the set of variables that were read for a given edge.
As merge **operator**, merge^{join} is used.
The **termination check** stop is based on the preorder $\sqsubseteq$ of the abstract domain. It uses the subset relation $\subseteq$ on sets for inclusion testing. For two abstract states e, e' the termination check is defined by: $\mathsf{stop}(e, e') = e \sqsubseteq e' = e' \subseteq e$.

IV. Implementation

In this chapter we describe the implementation of the live-variables analysis as a CPA in CPAchecker and the ways information collected by this CPA can be used by other analyses. Additionally we discuss different configurations of the Live-Variables CPA and some heuristics for enhancing other analyses with the information about the liveness of variables.

A. The Live-Variables CPA

Architecture. The Architecture of the Live-Variables CPA is the same as for every other CPA. However, due to the special requirements (backwards analysis, multiple initial states), it is hardly possible to use the Live-Variables CPA as a composite CPA with other analyses and is therefore integrated within the pre-processing step of CPAchecker (see Section IV-B). The analysis can be used for both, analyzing Java and C code.
Configuration Options. Using the configuration option *liveVar.evaluationStrategy* one can choose whether the analysis runs globally (*GLOBAL*) or function-wise (*FUNCTION_WISE*). When analyzing globally, the whole CFA of the program will be processed. This yields the advantage that the liveness of global variables, parameters and return values of functions can be analyzed. If the analysis is configured to run function-wise, the CFA of each function will be analyzed separately which is slightly faster than the global analysis[1].
Initial States. According to our formalism the set of initial states for the backwards analysis with the Live-Variables CPA consists of an empty state per endless loop (see chapter III) and additional states for all exit nodes. These states should be empty regarding the formalism, however, in real world programs function calls can occur additionally to assignments and assumptions. For being able to handle such programs we have to change this constraint slightly, in the way that for all

[1]This is the case because no cross-function references need to be analyzed, and therefore each function is only analyzed once and not as often as it is called by other functions.

exit nodes of the CFA we set the return variable[2] as being live. Furthermore depending on the kind of analysis additional nodes have to be added to the initial states:

- For a global analysis we need one initial state per exit node of the CFA and additionally one node inside each endless loop.
- For a function-wise analysis we need one initial state for the end of each function, and additionally one node inside each endless loop.

Special Cases for C Code. To support analyzing real-world C code with the LIVE-VARIABLES CPA we implemented handling of pointers partially[3], structs, arrays and function calls. In the following piece of code, `p` becomes live when initialized as it is read in the following line. This is the case because `p` holds the address where the value is stored. To assign a new value to the memory `p` points to, the address stored in `p` has to be read first.

```
1 int* p = malloc{sizeof(int)};
2 *p = 5;
```

The same applies to arrays. Assume we created an array like that: `int a[8];`. Every time we assign a value to a field of the array, the variable `a` is read. If we have a variable `i` and assign a value like `a[i] = 5;` the variable `i` is read as well. Structs are another one of those special cases.
Regarding the following piece of code one would say that `a.second` is live from line two to line three, however we see structs as a single variable, and partial assignments are ignored in liveness computation, thus `a` is live from line one up to line three, and therefore `second` is live, too.

```
1 a = {.first = 0, .second = 0};
2 a.second = 1;
3 if (a.second) { ... }
```

The last case to handle specially we want to discuss are function calls. Assume we have a function `int foo(int arg)`. Inside the function `arg` can be treated like any other variable. But as we cannot say if an argument will be read in the function[4] we have to treat a call like a read on all arguments. If the return value of the function is assigned to a variable like this `int a = foo(3);`, a read occurs on the return value of `foo` and there is a write on `a` but if the return value is not assigned to a variable neither a write nor a read occurs.

B. Integration of the LIVE-VARIABLES CPA into the Preprocessing of CPACHECKER

To use the information gathered by the LIVE-VARIABLES CPA during the analysis with an other CPA one cannot use both CPAs as Composite CPA, as the live-variables analysis runs, in contrast to most other analyses, backwards directed through the CFA. In order to be able to use the live-variables information regardless of the analysis direction, we integrated the live-variables analysis into the preprocessing step of CPACHECKER such that the generation can be toggled via the configuration option *cfa.findLiveVariables* set to *true* or *false*. Furthermore we introduced two options to set time limits for the live-variables analysis during preprocessing. With *liveVar.overallLivenessCheckTime* one can set the overall amount of time that should be maximally used by this preprocessing step, and with *liveVar.partwiseLivenessCheckTime* one can set the time that both, the global and function-wise approach may consume. If gathering the variables globally is not possible[5] we fall-back to the function-wise approach, therefore the partwise timelimit should approximately be the half of the overall timelimit, thus the function-wise approach can still be executed if the global analysis times out.
If the generation is switched on, the liveness information can be obtained via a `CFA` object. Orthogonally to the loop structure we introduced a method *getLiveVariables()* that returns an object containing the liveness information.

C. Usage of the Live-Variables Information in the Value Analysis CPA and the Policy Iteration CPA

With the extension of a CPA to a CPA+[BHT08] a precision determining the abstraction level of an analysis was introduced. During the precision adjustment step of the reachability algorithm for a CPA+ the state and precision used for the next abstract successor computation can be changed. The Value Analysis CPA [BL13] already defines some precision adjustment heuristics, therefore we decided to implement the abstraction of dead variables also in the precision adjustment of the Value Analysis CPA. When setting the option *cpa.value.blk.doLivenessAbstraction* to *true*, all variables that are not live at the current location are removed from the value analysis state. Furthermore we implemented the configuration option *cpa.value.blk.onlyAtNonLinearCFA*. This is a heuristic that reduces the liveness abstractions when toggled, such that the abstraction is only done when the current location node has more than one entering or leaving edge.
The Policy Iteration CPA is based on an abstract domain using template constraints [SSM05]. The generation of these templates depends on a given set of variables. For our case, the given variables are either all variables of the CFA, or all variables that are live for a given location. This can be configured via the option *cpa.stator.policy.varFiltering* which can be set to *ALL* or *ALL_LIVE*.

V. EVALUATION

In this chapter we will evaluate the influence of the liveness information on analyses with the Value Analysis CPA and the Policyiteration CPA. The performance will be measured with several configurations of these CPAs. We use the appropriate categories from the benchmark suite of the International

[2]These are a CPACHECKER specific variables, for every function with a return value other than void. Expressions in return statements are assigned to these variables, and the variables are then used in the call-site of the function.

[3]What currently cannot be handled is pointer aliasing. If the address of a variable is taken and written to another variable, our analysis is not able to detect and keep track of the liveness of aliased variables, whose values could be read and written without referring to the variable).

[4]Checking the liveness of parameters is currently not implemented.

[5]This could be due to timeouts, but also if we cannot surely determine all initial states e.g. when we have no information about the loops in the CFA.

Configuration	#correct		#false		Time (h)			Memory (TB)		
	all	equal only	alarms	proofs	correct	equal only	overall	correct	equal only	overall
basic	2326	2318	4	0	21.2	20.9	221	2.16	2.14	10.2
fun	2347		3		19.1	18.5	212	2.01	1.96	9.67
fun-heur	2344		3		19.5	19.0	213	2.04	1.99	9.77
glob	2350		3		19.0	18.6	207	2.02	1.99	9.74
glob-heur	2351		3		19.3	18.7	210	2.06	2.01	9.81
ref	2307	2282	3	0	18.6	16.2	185	1.64	1.52	9.36
ref-fun	2299		2		19.0	16.9	186	1.61	1.52	9.25
ref-fun-heur	2299		2		18.6	16.7	186	1.62	1.52	9.24
ref-glob	2296		2		19.0	17.3	186	1.61	1.54	9.28
ref-glob-heur	2303		2		19.8	17.4	186	1.66	1.54	9.27
policy-all	1468	1451	0	4	13.3	12.6	326	.657	.635	6.33
policy-onlyLive	1530				9.90	6.15	291	.499	.396	5.14

TABLE I
Overall Performance of the Analyses on the SV-COMP Benchmark Set

Competition on Software Verification (SV-COMP)[6] to evaluate the LIVE-VARIABLES CPA. These categories are: *BitVectors*, *ControlFlowInteger*, *DeviceDrivers64*, *ECA*, *HeapManipulation*, *Loops*, *ProductLines*, *Sequentialized* and *Simple*[7]. Overall our benchmark set consists of 4011 different files.

A. Configurations

For the evaluation we used several different configurations, the following paragraph lists all of them with a short explanation of the features. The two basic configurations without live-variables are called **basic** (the standard configuration of the Value Analysis CPA without CEGAR) and **ref** (the standard configuration of the Value Analysis CPA with CEGAR). They are extended by the following configurations: **fun**, **fun-heur**, **glob** and **glob-heur**. **fun** and **glob** stand for the live-variables collection strategy defined in Section IV-A. **heur** signalises that the heuristics (*cpa.value.vlk.onlyAtNonLinearCFA*) to do the liveness abstraction should be used only if a location has more than one entering or leaving edge. In the tables we will use the extended configuration names prefixed with **ref-** for analyses with CEGAR, and no prefix for others.
The policy iteration analyses only differ in the chosen sets of variables for the template generation. **policy-all** uses all variables of the CFA, **policy-oneLive** at least one live variable for a given location and **policy-onlyLive** only live variables.

B. Evaluation Environment

The evaluation was performed on machines with a 2.6 GHz Octa Core CPU (Intel Xeon E5-2650 v2) and 135 GB of RAM. The operating system is an Ubuntu 14.04 (64-bit) with a Linux 3.13.0-48 kernel. For the Java support OpenJDK 1.7 is used. The CPACHECKER revision for the evaluation is 16411[8] (*trunk*). Each single verification run was limited to 13 GB of RAM and to 500 seconds of run-time. The Java heap was set to 12.6 GB. For each verification run the overall amount of CPU time[9] and memory usage is measured. In all tables time and memory consumption will be given in hours and terabytes with three significant digits unless specified differently.

C. Evaluation Results for the Policy Iteration CPA

In contrast to the value analysis configurations, the Policy Iteration CPA uses the live variables not for decreasing state space, but for generating the templates that are used for computing the abstract successor. By changing the source for the generation from all variables in the CFA to only those variables that are live at the given location, the correctly analyzed programs increase by 62. Time and memory consumption decreases drastically, when regarding only the equally and correctly analyzed programs, over 51 % time and over 37 % memory is saved. When regarding all correctly analyzed programs per configuration, **policy-onlyLive** is about 25 % faster than **policy-all**, the memory consumption was decreased by 24 %.
Overall, by using only variables that are live for generating templates, the time consumption could be reduced by 35 h or 10 %, whereas the memory consumption could be decreased by 1.19 TB or 18 %.

D. Evaluation Results for the Value Analysis CPA

An overview over all results can be seen in Table I. The **basic** configuration is the slowest analysis and additionally it analyses only 2326 programs correctly, the lowest amount of all tested configurations. By using the live-variables information in combination with the **basic** configuration, the amount of correctly analyzed programs is increased, as well as the overall analysis time and memory consumption reduced. Only regarding correctly analyzed programs, using any of the configurations with live-variables abstraction results in about 10 % less time

[6]A more detailed description of the certain categories can be found here: http://sv-comp.sosy-lab.org/2014/benchmarks.php

[7]The other categories, for example *Arrays*, *Concurrency*, *Recursive* and *MemorySafety* use features like recursion or concurrency and can currently not be handled properly by the Value Analysis CPA.

[8]The policy iteration benchmarks were made on a later revision: 16870

[9]This time measure refers to the CPU time of the whole verification run.

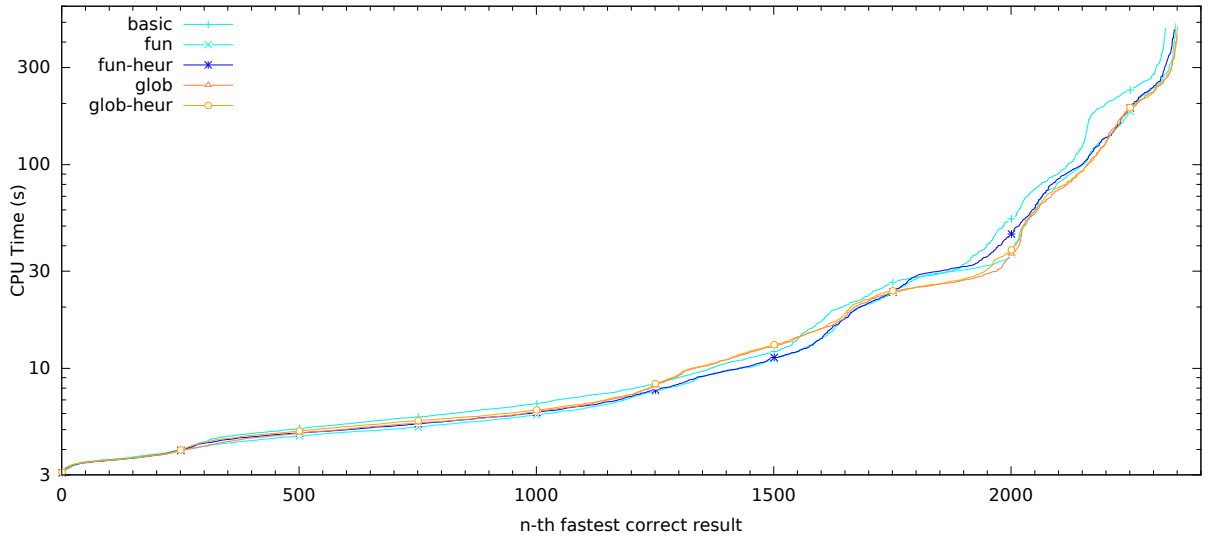

Fig. 2. A plot of quantile functions of the configurations **basic**, **fun**, **fun-heur**,**glob** and **glob-heur**

and approximately 5 % less memory consumption. None of the configurations produces false proofs and the live-variables abstraction reduces the false alarms by one compared to the **basic** respectively **ref** configuration.

If comparing all configurations that use refinement, one can see that the time and memory consumption (overall and correct results only) are almost equal, as well as the number of correctly analyzed programs. That the results with live-variables abstraction do not improve the analysis results further can be explained by the usage of CEGAR. As already only variables that are needed for refuting certain counterexamples are tracked, the states are more abstract from the beginning. For that reason removing dead variables from states has no noticeable impact on the analysis. The small reduction of correctly proved results while using the liveness information will be discussed in Section V-E. A more detailed comparison of the tested configurations is given in the following paragraphs.

Basic Configurations. The heuristic has no benefit for the function-wise configuration in combination with the basic configuration. Compared to **fun**, it produces less correct results and requires more time and memory for both, correct and all results. **glob-heur** can solve one program more than the **glob** configuration without the heuristic, but consumes about 1.5 % more time and 2 % more memory for the correct results. Considering only tasks with correct results for all basic configurations the liveness abstraction reduces CPU and memory usage. The **fun** configuration consumes 2.4 h or 11 % less CPU time and 0.18 TB or 8.4 % less memory compared to **basic**.

The **glob** configuration analyses three more programs correctly than the function-wise configuration without the heuristic. The overall time is 5 h or 2.4 % reduced by the global live-variables abstraction, however the memory consumption is increased by 70 GB. Regarding only the correct results both, time and memory values, are less than 1 % higher when **glob** configuration is used.

Configuration	∅ #reached states	∅ #reached locations
basic	280000	2250
fun	232000	2230
fun-heur	234000	2230
glob	229000	2230
glob-heur	233000	2230
ref	73800	1600
ref-fun	68600	1590
ref-fun-heur	70500	1590
ref-glob	66800	1570
ref-glob-heur	70000	1600

TABLE II
Average size of the reached set, and number of reached locations, only successfully-analyzed verification tasks are considered

The quantile plot in Figure 2 shows that the live-variables abstraction decreases the required CPU time for successfully analyzed verification tasks. The function-wise approach is faster for tasks requiring only about 10 s of CPU time and the global configurations are faster for tasks running longer than 20 s. The more costly preprocessing has only a positive impact on the CPU time if the task is complex enough. Around 150 s of CPU time the live-variables abstraction has the greatest benefit compared to the **basic** configuration.

Table II shows the average number of reached states and reached locations of correct results. The liveness abstraction reduces the average number of reached locations by 20 compared to the **basic** configuration. Liveness abstraction in combination with the **basic** configuration reduces the number of reached states by about 50,000 or 18 %. The **glob** configuration generates the least number of states. The heuristic leads to more reached states, because it performs the abstraction not for every abstract successor. The difference is 2000 between **fun** and **fun-heur** and 4000 between **global** and **global-heur**. The impact of the live-variables abstraction on the **ref** configuration is not so significant. The number of states is only

```
1  int main() {
2    int i, j = 10, n = nondet(), sn = 0;
3    for(i = 1; i <= n; i++) {
4      if (i < j) sn = sn + 2;
5      j--;
6    }
7    assert(sn == n * 2 || sn == 0);
8  }
```

Fig. 3. Program that cannot be analyzed correctly with configurations using live-variables abstraction

reduced about 7 % without and 5 % with the heuristic.

Ref Configurations. As mentioned in the overview of the evaluation (see Section V-D) removing variables that are not live from abstract states does not further improve the performance of the value analysis with refinement. That all **ref**-based configurations are mostly equal can also be seen in Table I. The time consumption for these analyses and their number of correctly analyzed programs are almost equal.

However, due to additional computations during precision adjustment, the overall time (correct results only) increases for almost all configurations by approximately 0.5 %. The only exception is **ref-fun-heur** which takes the same amount of time but produces 8 correct results less than **ref**. The memory consumption overall and per correct results only is for all configurations that use live-variables lower than for **ref**, except for **ref-glob-heur**. This can be explained with the smaller amount of reached states, which can be seen in Table II. Thus one can say that states created with a configuration based on **ref** using liveness abstraction are still more abstract than without liveness abstraction.

E. Restrictions and Challenges

Regarding the number of correctly analyzed tasks from Table I one can see, that for all configurations with CEGAR (**ref** prefix) using the liveness information during the precision adjustment is worse than not using it. On the surface this should not be the case, as deleting variables which values are not used should not impact the analysis. However some correct results of the analysis can only be found by chance.

In the example in Listing 3, a loop is traversed a nondeterministic number of times. Each traversal increments i and decrements j. During the first five iterations ($n \leq 5$) everything is fine, and the assert after the loop will not be triggered. If $n > 5$ the assert statement does not hold, as $sn = 10$ and $n * 2 \geq 12$. The Value Analysis CPA is not able to infer any information about n, however without the live-variables analysis, the abstract states are always different when reaching the end of the loop, and therefore there is no coverage. Thus the counterexample check finds that the assert statement does not hold and reports an error. In contrast to that, when using the liveness information in the precision adjustment step of the value analysis, all information about dead variables is erased from the states. This means that there are exactly six possible states after exiting the loop, where sn will either be zero, two, four, six, eight or ten. When reaching the state where $sn = 10$ the first time, the counterexample check cannot find an error, as everything is valid. But later on, all states will also only know that $sn = 10$ and therefore coverage is assumed, as the same state was found before. This leads, due to the fact that the value analysis cannot store relations between variables, which would be necessary for proving this program, to the result *"unknown"*.

VI. Conclusion

A variable is live in the part of a program where it is accessed. Information about the liveness of variables can be used in model checking to reduce the amount of states in a CFA and by this save time and memory during the analysis of programs.

The Live-Variables CPA is an implementation of a live-variables analysis in CPAchecker to provide information about the liveness of variables for other CPAs. It can improve time and memory consumption of other CPAs by removing unnecessarily tracked variables from their abstract states.

As shown in Section V-D the impact depends on the analysis that is using the information provided by the live-variables analysis.

We expext that for CPAs that need more memory per variable and state than the Value Analysis CPA (e.g. octagon or polyhedric domains) the impact will be higher, as also more memory can be saved by abstracting these states from non-live variables.

Besides using this information to have smaller states it is used in the policy iteration configuration of CPAchecker for template generation.

Future Work. The Live-Variables CPA can still be refined. For example tracking liveness of struct fields or arrays could be implemented more fine grained. Such that the liveness of single array cells, or struct fields is tracked. Another possible improvement is, to take the liveness of function parameters into consideration when analysing globally.

Moving the implementation of the live-variables abstraction in the Value Analysis CPA from the precision adjustment to the transfer relation could yield performance improvements due to the additionally available information.

References

[BHT07] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. *Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis.* Proc. CAV, LNCS 4590, pages 504–518. Springer, 2007.

[BHT08] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. *Program Analysis with Dynamic Precision Adjustment*. Proc. ASE, pages 29–38. IEEE, 2008.

[BK11] Dirk Beyer and M. Erkan Keremoglu. *CPAchecker: A Tool for Configurable Software Verification.* Proc. CAV, LNCS 6806, pages 184–190. Springer, 2011.

[BL13] Dirk Beyer and Stefan Löwe. *Explicit-State Software Model Checking Based on CEGAR and Interpolation*. Proc. FASE, LNCS 7793, pages 146–162. Springer, 2013.

[Ken81] Ken W. Kennedy. *A survey of data flow analysis techniques*. Program Flow Analysis: Theory and Applications, chapter 1, pages 5–54. Prentice-Hall, 1981.

[SSM05] Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. *Scalable Analysis of Linear Systems using Mathematical Programming*. Proc. VMCAI, LNCS 3385, pages 21–47. Springer, 2005.