Annales Univ. Sci. Budapest., Sect. Comp. 30 (2009) 141-155

Goblint: PATH-SENSITIVE DATA RACE ANALYSIS[1]

V. Vojdani and V. Vene

(Tartu, Estonia)

Abstract. We present Goblint, a static analyzer for detecting potential data races in the multithreaded C code. The implemented analysis is sound on a "safe" subset of C and sufficiently efficient to be used for race-detection of multithreaded programs up to about 25 thousand lines of code. It uses a global invariant approach to avoid the state space explosion problem and is both context- and path-sensitive.

1. Introduction

A *multiple access data race* is a problem in low level concurrent programming, where different threads simultaneously attempt to update shared memory without synchronization. Due to the non-deterministic nature of thread execution, this may lead to unexpected behaviours of the program. Errors of this kind are very difficult to find by testing, as usually it is not feasible to simulate all possible thread schedulings in a controlled environment. Hence, it is important to have tools that can detect races and, if it is the case that the code is race-free, prove the absence of races by a static program analysis.

In order to prove the absence of data races, the analysis must be sound. This means that the analysis result must always contain all real data races but may also include spurious false alarms. Of course, we want the analysis to be as precise as possible, so as to keep the number of false warnings minimal. However,

[1]Partially supported by the Estonian Science Foundation under grant No. 6713.

there is trade off between efficiency and precision, and finding the right balance is of utmost importance for any practically useful analyzer. In particular, one has to deal with the exponential state space explosion inherent to the analysis of multithreaded programs, due to all possible interactions between the different threads.

In this paper, we present Goblint, a static analyzer for detecting potential data races in the multithreaded C code.[2] The implemented analysis is sound (on a "safe" subset of C, e.g. no arbitrary pointer arithmetic, etc.) and sufficiently efficient to analyze software projects of about 25 thousand lines of code in a few minutes on standard PC configurations. The analyzer is based on the multithreaded inter-procedural framework by Seidl et al. [15], using the so called side-effecting constraint systems to compute a single safe approximation of the global state – a global invariant. The global invariant is then used for analyzing each thread separately, thus avoiding the state space explosion.

Related work. The analysis of multithreaded software is notoriously difficult [13]. Most race detection tools are either based on dynamic analysis, which can not prove the absence of bugs, or use type-based approaches, which rely on time-consuming programmer annotations [5, 12]. In the last two years, however, some impressive static data race analyzers have been presented [6, 7, 12]. Of these, the Locksmith analyzer is the most similar to Goblint: it is sound, open-source and analyzes C.

While soundness is recently gaining popularity [1], it is still not something everyone is aiming for. Kahlon et. al [6] do not mention whether their method is sound. Their basic approach is similar to the Chord analyzer [7], which was not sound. Naik and Aiken have since developed a novel technique, *conditional must-not aliasing*, to obtain a sound race detection algorithm for Java [8].

When it comes to analyzing Posix threaded C, the Trier data race analyzer [15] was able to prove the absence of races in safety-critical aviation software, and Goblint is based on these methods. Locksmith [12], which uses a CFL reachability based approach, is a more recent project. An interesting innovation in Locksmith is the use of existential types to correlate locks and data in dynamic date structures [11].

Goblint is unique among these race detection tools in relying on a sound thread-modular constant propagation and points-to analysis. This allows the analysis of conditional locking schemes and can take into account the possibility that locking operations may fail.

Outline. The rest of the paper is organized as follows. In the next section we give an overview of the methods we use for context-sensitive constant propagation

[2] Goblint is an open-source project that lives at `http://goblint.at.mt.ut.ee`

and points-to analysis of multithreaded C code. Then, in Section 3, we describe how this information can be used to perform path-sensitive data race analysis. We then briefly describe the Goblint static analyzer in Section 4. Finally, in Section 5, we report some preliminary benchmarking results and, in Section 6, we conclude the paper.

2. Analysing multithreaded C

Precise data race analysis of real world programs requires careful analysis of control flow, but this in turn requires precise knowledge of the values of pointers and conditional variables. The most straight-forward approach is to adopt normal data flow analysis techniques to the more complicated situation of multithreaded analysis.

At the core of this technique is a typical abstract interpretation based data flow analysis [2]. From the program code, one generates a control flow graph (CFG) containing only assignments, calls and branches. One has to define how each type of edge transforms the state from one node to the other. Based on this specification a constraint system is generated and solved. The difficulty, however, lies in dealing with multithreaded code. Consider the following example:

```
int global;
void race() { global++; }
void nice() { printf("mu"); }
void (*f)() = nice;
void *tfun(void *arg) {
  f(); return NULL;
}
```

```
int main() {
  pthread_create(tfun);
  f = race;
  global++;
  return 0;
}
```

The program starts by spawning a thread which executes the function `tfun`. This thread makes an indirect call through the function pointer `f`. This pointer is at the beginning of the program initialized to the harmless function `nice()`. However, by the time the pointer is dereferenced, the main thread might have already updated it, so that instead the dangerous function `race()` is called. A sound analyzer like Goblint must assume the worst and take such unlikely interleavings of thread execution into account.

The problem, as we mentioned in the introduction, is that considering all possible interleavings is computationally unfeasible. The method we use to address this issue has been treated formally in a paper on the Trier static analyzer [15]. The key idea is to analyze each thread in separation by identifying the effect it has on the rest of the program. This information can then be used to anal-

yse each thread in separation, that is, in a thread-modular fashion. To do this efficiently, one can compute the side-effects simultaneously with the sequential analysis using a demand-driven solving engine.

In the example, we would start analyzing the main function, but as a thread is spawned, the solver first looks into the execution of the thread code. This is initially analyzed such that only the call to `nice()` is considered, but the solver notices that the call depends on the value of the global function pointer. As the solver returns to the analysis of the main function, the pointer is updated. This triggers the re-evaluation of all nodes depending on the variable `f`. Since it may now also point to `race()`, this function is therefore analyzed as well and the result of its call is joined with the previously analyzed function.

In general, we compute this by solving a system of control flow equations using a regular constraint solver, but whenever the global state changes, we must recompute the analysis with respect to a new global invariant. However, when the global state is such that it maps each global variable to a certain abstract value, we can use a more efficient algorithm that tracks dependencies between globals and the nodes that use them. Thus, we re-evaluate as few nodes as possible, while still remaining sound. In the above example, we return to the call of `f()` and only analyze the dangerous function to notice the data race that may occur.

Analyzing functions precisely is in itself a difficult problem. Context sensitivity is the ability to distinguish different calling contexts. When checking for data races in C code, one immediately faces the problem that many functions indirectly access variables and the locks that protect them. Analyzing such cases requires careful treatment of function calls. The following is a typical example:

```
void safe_inc(int *v, pthread_mutex *m) {
  pthread_mutex_lock(m);
  v++;
  pthread_mutex_unlock(m);
}
```

A vital ingredient in our approach is the use of a general purpose constraint solver [4]. Rather than running a fixpoint computation immediately on the control flow graph, we generate a system of constraints and use a dedicated constraint solver to compute the fixpoint. This allows us to achieve effective cloning based inter-procedural analysis.

The benefit of using a generic solver is that we are free to redefine what constitutes a variable in the constraint system. For intra-procedural analysis it suffices to take constraint variables to be the nodes of the control flow graph. However, by attaching some context information to each node, one can easily choose between different approaches to inter-procedural analysis [14].

We currently use the so-called functional approach to inter-procedural analysis [16]. Let $\mathcal{F}$ be set of procedures, $\mathbb{D}$ our abstract domain and N the nodes in the CFG. Then the set of variables V in the constraint system are:

Calls: $\langle f, d\rangle$, where $f \in \mathcal{F}$ is a procedure and $d \in \mathbb{D}$ is the state in which the function was called. The result of function calls are stored in these variables.

Nodes: $\langle n, d\rangle$, where $n \in N$ is a node in the control flow graph. If this node was in the body of procedure f, then the second component d denotes the state in which the function f was called. This is how function calls are kept separate.

Thus the set of variables is the product $(\mathcal{F} \cup N) \times \mathbb{D}$ which is an infinite set. However, *demand-driven* solvers can deal with an infinite constraint system as long as only a finite number of variables need to be solved. Note that this implies a lot of cloning procedure bodies. This is quite expensive, but in return precise information about the values of program variables is not lost in function calls.

The domain we use for constant propagation and points-to analysis assigns abstract values to the program variables. The value domain is a recursive domain of either arrays, structures, unions, addresses or integers.

Arrays are represented as either a sequence of values of a finite size or a collapsed single value.

Structs are always fully expanded. We assign a value to each field of the structure.

Unions are analyzed as structures, but we keep a pair of the value and the type of the field that was last used to assign to it.

Addresses is a set of host and offset pairs, where the offset is a list of either indexes from our integral domain or field names.

Integers are represented using a special domain to represent either definite values or a finite set of excluded values.

The integer value domain we use is specifically designed to deal with heavily branching code. It is similar to the commonly used Killdal domain, but it is topped by finite exclusion sets rather than a single unknown. Thus, it can embed the boolean domain (it can express `true` as the exclusion set containing zero) and it can also express some other conditions that are useful when analysing `switch`-constructions.

More formally, values in this abstract domain $\mathbb{D} = \{\bot, \top\} \cup \mathbb{Z} \cup \mathcal{P}(\mathbb{Z})$ are ordered in the obvious way that respects the following concretization to the powerset domain $\mathcal{P}(\mathbb{Z})$:

$$\gamma(\top) = \mathbb{Z} \qquad \gamma(\bot) = \emptyset$$
$$\gamma(n) = \{n\} \qquad \gamma(X) = \mathbb{Z} \setminus X$$

and the least upper bound is defined as follows for the non-trivial cases:

$$n_1 \sqcup n_2 = \begin{cases} n_1 & \text{if } n_1 = n_2 \\ \top & \text{if } n_1 = 0 \text{ or } n_2 = 0 \\ \{0\} & \text{otherwise} \end{cases} \qquad \begin{aligned} X \sqcup n &= X \setminus \{n\} \\ X_1 \sqcup X_2 &= X_1 \cap X_2 \end{aligned}$$

The special treatment of zero is to support the embedding of the boolean domain. This domain is also useful for analyzing operations that do not tolerate certain values. Thus, it can be used for proving that divisions by zero do not occur.

3. Path-sensitive data race analysis

Path-sensitivity is the ability to distinguish between real executable paths in the control flow graph and imaginary paths that are unreachable due to logical constraints. The `gcc` manual has the following example, where "GCC is not smart enough" to see that the code is bug free:

```
int save_y;
if (change_y) save_y = y, y = new_y;
...
if (change_y) y = save_y;
```

Being "smart enough" would in this case require detecting the relationship between the conditional guards. Among the four possible paths in the CFG, one should only analyse the two logically possible paths. Engler and Ashcraft [3] propose what they call "unlockset analysis" to achieve path-sensitive analysis of locked mutexes. This was needed to deal with the complicated control flow in FreeBSD code and although practical, it is an ad-hoc solution that does not aim to be sound.

In order to treat this in a sound way, we could use a powerset domain $\mathcal{P}(\mathbb{D})$, but this is not feasible since the constant propagation domain is infinite. One could limit the number of paths that are distinguished with some arbitrary constant and just merge any further branching. Early experiments showed that this is not a feasible approach either, and one really must decide which paths to distinguish. Note that path-sensitivity depends on the values of program variables and this is one reason we have to track even variables that initially seem irrelevant to race detection.

We will return to path-sensitive race detection shortly, but let us first describe our approach to data race analysis. To detect races one needs to know the locks that are held when shared variables are accessed. In Posix threaded C, the shared data can be either global variables or dynamically allocated heap data. Goblint attempts to abstractly evaluate the program to see what shared memory locations are accessed. Whenever a global is accessed it registers the currently held locks and information to identify the threads that could have been involved. The most important information it has to compute, in addition to the constant propagation described above, is the set of locks that are held at each program point.

A simple lockset analysis for sound race detection keeps track of the set of mutexes that *must* be held when a global variables is accessed. The domain here is the powerset of the set of all possible mutexes and the join operation is intersection. The analysis is thus a gen/kill-analysis [10], where lock operations generate at most one mutex, and unlock operations kill a set of locks. Consider again the example:

```
void safe_inc(int *v, pthread_mutex *m) {
  pthread_mutex_lock(m);                    (1)
  v++;                                      (2)
  pthread_mutex_unlock(m);                  (3)
}
```

If the destination of the pointer variable `m` can be ascertained by the base analysis, that is, if we know the location it *must* point to, then we add it to the lockset on line 1. The next line then requires that we register a write operation with the current lockset for all globals that `v` *may* point to (in this particular call of the function). The unlock operation on line 3 should remove all possible locks that could be the destinations of the mutex pointer `m`.

The above works well under the assumption that locking always succeeds. For a sound lockset analysis, we cannot make such assumptions. The standard practice in POSIX threaded C is to always check the returning value of the locking function:

```
status = pthread_mutext_lock(m);
if (status != 0)
    err_abort(status, "Lock mutex");
```

Sound and precise analysis of this situation requires that we be path-sensitive, so that when the user checks the `status` variable, the bodies of the conditionals are analyzed with the correct lockset. We will now see how to be path-sensitive in an efficient way. Consider the following simplified example, which will raise a false alarm on line 4, unless the analysis is path-sensitive:

```
void foo (int do_work) {
    if (do_work)                                (1)
        pthread_mutext_lock(&mutex);            (2)
    ...
    if (do_work)                                (3)
        work++;                                 (4)
    ...
    if (do_work)                                (5)
        pthread_mutext_unlock(&mutex);          (6)
}
```

The problem, again, is that there are now eight potential paths, but only two logically possible paths, and we must eliminate the false paths. We want to do this inter-procedurally and without sacrificing the soundness of the analysis.

We denote with $\mathbb{D}_b$ the domain used by our base analysis and with $\mathbb{D}_l$ the lockset domain. If we combine them trivially using the product domain $\mathbb{D}_b \times \mathbb{D}_l$, then information is lost and the eight paths are mixed together. In order to discriminate different paths, we should track the state of the conditional variables on lines 1, 3 and 5 because their correlation logically prohibits all but the two real executions (assuming the omitted code does not modify the `do_work` variable).

The immediate idea would be to track the lockset for each state of the conditional variables $\mathbb{D}_b \to \mathbb{D}_l$, but since the base domain is essentially infinite, this is just as bad as the powerset domain. Instead, we try to be as path-sensitive as necessary for this particular analysis. We use the domain $\mathbb{D}_l \to \mathbb{D}_b$ to get just as much precision as needed to solve the problem at hand without creating a potentially infinite domain (assuming the set of locks used by the program are finitely represented).

The analyzer can use this information to distinguish the paths by performing a Conditional Constant Propagation [17]: when reaching conditionals with an unknown guard (line 1), the true-branch is analyzed assuming the conditional is true and yields the resulting state $[\{\&\texttt{mutex}\} \mapsto [\texttt{do_work} \mapsto \mathsf{true}]]$, and assume the opposite for the false-branch, which in the current example trivially results in $[\emptyset \mapsto [\texttt{do_work} \mapsto \mathsf{false}]]$. By merging in the domain $\mathbb{D}_l \to \mathbb{D}_b$ we keep the states separated when exiting a branch, if they contain different locksets, as is currently the case.

When the relevant states are kept separate, subsequent conditional guards can be evaluated as constants and the wrong paths will be considered dead code for the states with irrelevant locksets. Thus, when analyzing line 3, the true-branch is considered dead code in the state $[\emptyset \mapsto [\texttt{do_work} \mapsto \mathsf{false}]]$, and line 4 is only analyzed with the correct mutex set. The false alarm is therefore avoided.

During the mutex analysis, we generate information about the usage of global variables in the program. For each global, we have a set of accesses with informa-

tion about the held mutexes. We can use this information to warn the user about any potential race. Currently, we check for a very simple, but most commonly used, programming idiom to ensure race freedom. Namely, we require that all accesses to the same global variable must have a common mutex locked, i.e. we take the intersection of the locksets and give a warning if it is empty.

Of course, we could use the computed lockset information to check for more complex idioms, such as requiring that all accesses of the given global have pairwise overlapping locksets. For instance, if the global has three accesses in total and the corresponding locksets are $\{\texttt{mutex1}, \texttt{mutex2}\}$, $\{\texttt{mutex2}, \texttt{mutex3}\}$ and $\{\texttt{mutex1}, \texttt{mutex3}\}$, then that is sufficient to guarantee the absence of a data race. Our approach allows checking for such coding convention, and is also compatible with the use of statistical methods to infer programmer assumptions about which globals really require mutexes.

In a sound framework, statistical methods can only be used to rank the warnings, but we would like to avoid as many spurious warnings as possible. The problem is that globals can be protected from race conditions not just by mutexes, but through other synchronization methods and the use of specific sharing idioms. To identify when a global variable is only accessed by a single thread, one would need a fairly sophisticated thread identification analysis. Currently, we only use the most crude distinction, namely, we analyze the code before the very first thread is created in a single-threaded fashion, and then distinguish the main thread from other threads. This avoids the large number of spurious warnings when globals are initialized in the main thread before spawning any other threads.

4. The Goblint analyzer

The Goblint analyzer consists of three parts, a user interface component, an analysis module and a C frontend (Figure 4). The frontend uses CIL [9] to parse and simplify C into an intermediate form that can easily be turned into our representation of a control flow graph. Based on specifications of the analyzes, we generate a constraint system that we solve using a general purpose constraint solver [4]. The result is then mapped back to the original program and warnings about potential bugs are reported.

The data race analysis can be roughly divided into three separate components. As outlined in the previous section, we perform a base analysis to deal with the complications of C control flow. Simultaneously with the base analysis and drawing heavily on the information it provides, we perform the lockset analysis, where we register what mutexes are held when a global variable is accessed. When

this is done, we post-process the information about the globals and determine any potential races, and then report this information and display it on the Goblint User Interface.

We expect Goblint to be a useful tool for detecting real bugs, and when possible, proving the absence of data races in real code. We hope it will be found useful for C programmers. If we want to reach this goal, we must integrate into a real world development environment. However, the result of a static program analysis is often more difficult to interpret than compiler warnings. The few analysis tools that aim to be user-friendly use their own user interface, which show the result directly on the control flow graph. We have experimented with a different approach. We have developed a plug-in for the Eclipse integrated development environment's C development tools, and we try to display the result on the original program rather than the intermediate structures that are used for analysis.

The Goblint analyzer itself is a command line tool with many options for displaying the output. When Goblint presents the result of the analysis as an XML file, it can be parsed by the Eclipse plug-in. This leverages on all the benefits that a proper IDE has to offer, such as easy navigation to warnings and much more (see Figure 2). When the user views the program and clicks on a line, the state of the analysis at that program point is displayed. This process is not entirely trivial, as one has to map the result from the CFG back to lines in the program code while distinguishing different calling contexts. The current user interface is reasonably successful at these tasks.

5. Experimental results

We have experimented with Goblint on a set of open-source programs of up to 25 thousand lines of code. In order to compare more easily with the state of the art, we discuss here the same benchmarks as were used by the authors of Locksmith. We ran these experiments on an Intel Core 2 Duo @ 1.83GHz PC with 2GB of RAM. The latest benchmarking results on our complete set of test

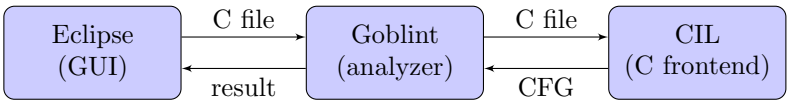

Figure 1. The components of Goblint

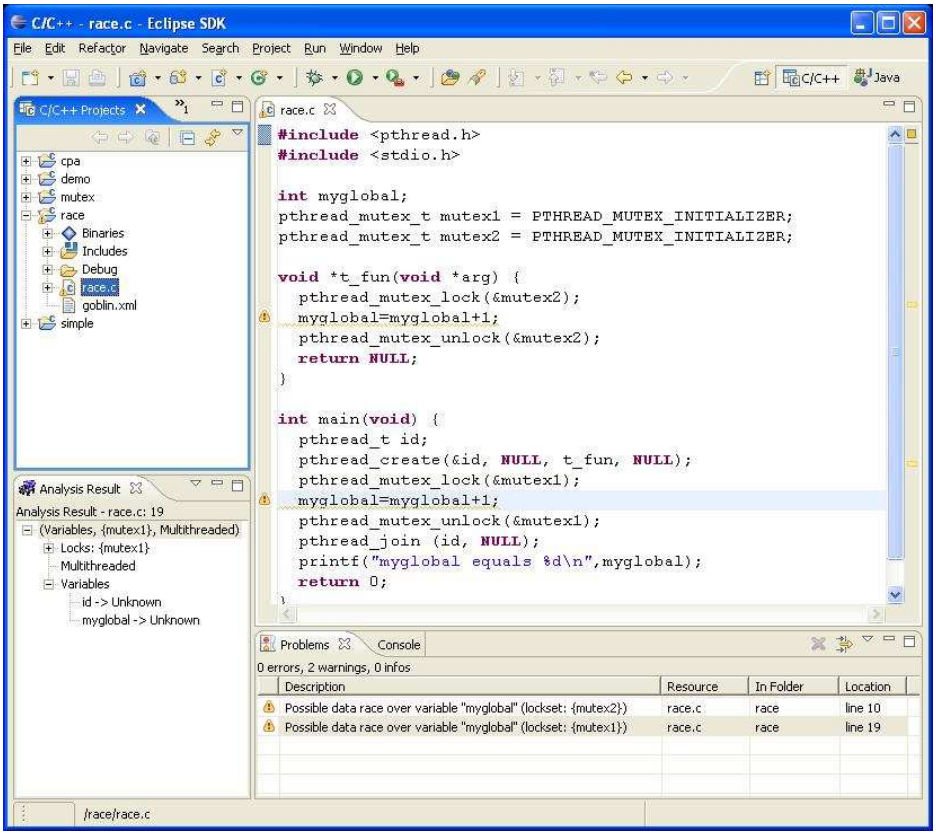

Figure 2. A screenshot of Goblint

programs are available on the Goblint homepage. The current evaluation was obtained using version 0.9.3 of Goblint and version 0.4 of Locksmith.

The following is a description of the test programs that we used: *aget* is a multithreaded download accelerator; *pfscan* is a multithreaded parallel file scanner; *knot* is a multithreaded web server distributed with the Capriccio threads package; *ctrace* is a fast, lightweight trace/debug C library. It contains a sample program with many data races; *smtprc* is a fully configurable, multithreaded open mail relay scanner.

We compared the number of warnings reported with the number of warnings reported by the Locksmith under two restrictions. We are not considering at this point any dynamically allocated memory and our analysis is field-insensitive. The latter means that we fail to distinguish locks that are correlated to particular fields, but rather we see the entire struct as a single memory location when it

Benchmark	Size (kloc)	Goblint		Locksmith		Races
		Time	Warn.	Time	Warn	
aget	1.2	0.3	5	1.0	4	4
knot	1.3	0.3	7	9.1	8	7
pfscan	1.3	0.1	2	0.6	2	0
ctrace	1.4	0.3	2	3.0	2	0
smtprc	5.7	12	2	8.2	0	0

Table 1. Summary of experimental results

comes to race detection. To be fair to Locksmith, we turned these features off, since they have a negative impact on its performance.[3]

The result is summarized in Table 1, where we indicate for each program the runtime in seconds for its analysis and the number of warnings raised by the different analyzers. We also provide what we believe is the right number of warnings. This number is based on our manual analysis of the programs and is therefore completely subjective.

The analysis of *aget*, *pfscan*, and *smtprc* are more accurately handled by Locksmith. We have a couple of false alarms due to our more naive treatment of dynamic data structures.

The test-program *knot* is more interesting from our perspective, because it has many global configuration options, and one of them is to turn caching on or off. Goblint can see this difference, and therefore it does not warn on two caching related data races. There is also a potential race over the setting of a *global* thread attribute, which we warn about but Locksmith does not.

Similarly, for *ctrace* the initialization function has a parameter to determine whether tracing is asynchronous or not. In the code we analyze, tracing is not performed by a separate server thread, so the two races reported by Locksmith can not occur in the code we are analyzing. If we *change* the code to make tracing asynchronous, then Goblint does raise the corresponding warnings for *such* a program. Unfortunately, we give two other warnings for this program. These are false alarms because the variables are protected by semaphores, which that we do not handle, yet.

[3] We ran Locksmith with the flags no-linearity, no-existentials, and field-insensitive.

6. Conclusions

We have presented a static analyzer for detecting potential data races in the multithreaded C code — Goblint. The implementation was done in O'Caml using CIL as our C frontend and the Eclipse IDE as the graphical user interface. The implemented analysis is sound on a reasonably large subset of C. It uses a global invariant approach to avoid the state space explosion problem and is both context- and path-sensitive.

Although we have not been able to demonstrate the efficacy of our sound path-sensitive flow computation using open-source programs, it was useful on the set of proprietary benchmark from aviation control code [15]. We are looking towards Linux device drivers and other safety critical code to demonstrate the full benefit of our approach on open-source software. Nevertheless, the preliminary benchmarking on these programs are encouraging in terms of precision and the system's performance. In particular, we have successfully used the analyzer on a set of open-source multithreaded programs resulting in very few false alarms that we do not know how to deal with.

An interesting (and quite surprising) discovery was the impact of constant-propagation to the result of the analysis. Changing arguments of initialization procedures and configuration constants can propagate to change the nature of race conditions elsewhere in the program. While not quite as tantalizing as quantum-entanglement, it does provide justifications for spending computational power on constant propagation in order to prove the absence of races.

Acknowledgement

We would like to thank Kalmer Apinis, Jaak Randmets, and Toomas Römer, for their help in the development of the analyzer. We are also grateful to the authors of Locksmith [12] for having made their benchmarking suite freely available. Finally, we would like to thank Helmut Seidl for his useful suggestions on how to improve Goblint.

References

[1] **Anderson Z., Brewer E., Condit J., Ennals R., Gay D., Harren M., Necula G. C. and Zhou F.,** Beyond bug-finding: Sound program analysis for linux, *HotOS*, 2007.

[2] **Cousot P. and Cousot R.,** Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, *POPL '77*, ACM Press, 1977, 238–252.

[3] **Engler D. and Ashcraft K.,** RacerX: effective, static detection of race conditions and deadlocks, *SOSP '03*, ACM Press, 2003, 237–252.

[4] **Fecht Ch. and Seidl H.,** A faster solver for general systems of equations, *Science of Computer Programming*, **35** (2) (1999), 137–161.

[5] **Flanagan C. and Freund S.N.,** Detecting race conditions in large programs, *PASTE '01*, ACM Press, 2001, 90–96.

[6] **Kahlon V., Yang Yu, Sankaranarayanan S. and Gupta A.,** Fast and accurate static data-race detection for concurrent programs, *CAV '07*, LNCS **4590,** Springer-Verlag, 2007.

[7] **Naik M., Aiken A. and Whaley J.,** Effective static race detection for Java, *PLDI '06*, ACM Press, 2006, 308–319.

[8] **Naik M. and Aiken A.,** Conditional must not aliasing for static race detection, *POPL '07*, ACM Press, 2007, 327–338.

[9] **Necula G. C., McPeak S., Rahul S. P. and Weimer W.,** Cil: An infrastructure for C program analysis and transformation, *CC '02*, LNCS **2304** , Springer Verlag, 2002, 213–228.

[10] **Nielson F., Nielson H.R., and Hankin C.L.,** *Principles of program analysis*, Springer Verlag, 1999.

[11] **Pratikakis P., Foster J. S. and Hicks M.,** Existential label flow inference via CFL reachability, *SAS '06*, LNCS **4134**, Springer Verlag, 2006, 88–106.

[12] **Pratikakis P., Foster J.S. and Hicks M.,** LOCKSMITH: Context-sensitive correlation analysis for detecting races, *PLDI '06*, ACM Press, 2006, 320–331.

[13] **Rinard M.,** Analysis of multithreaded programs, *SAS '01*, LNCS **2126**, Springer Verlag, 2001, 1–19.

[14] **Seidl H. and Fecht Ch.,** Interprocedural analyses: a comparison, *Journal of Logic Programming*, **43** (2) (2000), 123–156.

[15] **Seidl H., Vene V. and Müller-Olm M.,** Global invariants for analyzing multithreaded applications, *Proc. of the Estonian Academy of Sciences: Phys., Math.*, **52** (4) (2003), 413–436.

[16] **Sharir M. and Pnueli A.,** Two approaches to interprocedural data flow analysis, *Program flow analysis: Theory and applications*, 1981, 189–234.

[17] **Wegman M.N. and Zadeck F.K.,** Constant propagation with conditional branches, *ACM Trans. Program. Lang. Syst.*, **13** (2) (1991), 181–210.

V. Vojdani and V. Vene
Depatment of Computer Science
University of Tartu
J. Liivi 2
EE-50409 Tartu, Estonia
{vesal,varmo}@cs.ut.ee