RCanalyser: A Flexible Framework for the Detection of Data Races in Parallel Programs

Aoun Raza and Gunther Vogel

University of Stuttgart
Institute of Software Technology, Universitaetsstrasse 38
70569 Stuttgart, Germany
{raza,vogel}@informatik.uni-stuttgart.de

Abstract. Creating multiple threads for performance gain is not only common for complex computations on supercomputers but also for ordinary application programs. Multi-threaded/parallel programs have many advantages but also introduce new types of errors that do not occur in purely sequential programs. Race conditions are one important class of these special problems because the effects of race conditions occur nondeterministically and range from incorrect results to unexpected program behaviour. This paper presents RCanalyser, a tool for the detection of race conditions, which is based on a $Must\_Locks$ analysis using a flexible interface for the integration of different points-to analyses. As the problem of detecting race conditions is NP-hard in the general case, the tool is restricted to the detection of so-called data races [1]. The tool is able to analyse C/C++programs that use thread APIs for the implementation and synchronization of concurrent units. We applied the tool to a set of real programs, which use the POSIX thread API, and present results and statistics.

1 Introduction

In parallel programs, different threads are created and often communicate with each other. Different communication methods are available for these interactions, e.g., message passing or shared memory. Furthermore, threads may need to claim other system resources that are shared among them. As multiple threads try to access shared resources, their actions must be protected through some synchronisation mechanism to avoid interleaving. Absence of such a mechanism during these accesses can lead to inconsistent states of shared resources, which can result in abnormal or unpredictable program behavior. An important class of inter-process or inter-thread anomalies is race conditions. A race condition occurs when different threads simultaneously perform read and write access on shared data without prior synchronization. Such erroneous situations tend to be very difficult to detect or to recreate by test runs; they arise in real-life execution as an inexplicable, sudden, and not re-creatable, sometimes disastrous malfunction of the system. Debugging of parallel programs requires tools and mechanisms that can discover such situations and assist programmers in locating the culprit source code. Moreover, for parallel programs using shared resources, mechanisms are required to determine when a race condition can manifest and, to provide further assistance in locating it. Due to their critical effects on the deterministic behaviour of software, guidelines on thread programming have been devised to avoid data races [2]. However, this

F. Kordon and T. Vardanega (Eds.): Ada-Europe 2008, LNCS 5026, pp. 226–239, 2008.

can restrict a programmer's benefits achievable through full use of the concurrency features of a programming language.

Many research communities have investigated this issue and have proposed different dynamic and static techniques to detect race conditions [3,4,5,6,7,8,9,10]. The available race detection techniques exhibit different limitations depending on the nature of analyses underneath, i.e., dynamic or static analyses. Some of them raise the degree of false positives while others impose overhead in terms of time and space complexity [11].

This paper presents the tool RCanalyser, which performs flow- and context-sensitive analyses of the program to find a $Probable\_Lock$ for each shared variable. Further, it provides the flexibility to combine different points-to analysis mechanisms while present tools do not incorporate points-to analysis or cover only simple aspects. Our mechanism performs detection of data races based on locks and shared variable analysis. Shared variable discovery has a major effect on the accuracy of data race detection. Therefore, unlike others [4,12], in our approach we first discover shared variables and then investigate if they are consistently protected. For the safe detection of accesses to shared variables and the execution of synchronization operations, knowledge about the potential targets of pointers is essential. RCanalyser is a part of the Bauhaus [13] tool suite and uses its interface for different points-to analysis techniques implemented in the infrastructure which helps us to increase the precision for those programs which heavily use pointers. This paper is structured as follows; section 2 discusses the related work in this area. In section 3 we explain our definitions and terminology for the scope of this paper. The design and implementation of RCanalyser is described in section 4. Section 5 presents the evaluation of RCanalyser. Finally, section 6 concludes the paper and discusses future trends.

2 Related Work

Since the detection of race conditions in parallel programs is notoriously difficult, a large community has focused on this issue. In fact, it is quite difficult to detect such problems by manually testing the programs. Additionally, most of the existing concurrent software systems are written in C. Therefore, the need of an efficient mechanism for detecting parallel program anomalies is always present. As a consequence of difficulties involved in the race detection process, tools and mechanisms which provide automatic detection are extremely valuable. Hence, there has been a substantial amount of past work in building tools for analysis and detection of data races [3,4,5,6,7,8,9,10]. These tools are either based on the verification of access event ordering or they verify a locking discipline for mutual exclusion [7]. This means, if there is no unordered access to a shared variable such that at least one access is a write, the program is free from race conflicts. Similarly, if the accesses to shared variables in a program obey a locking discipline then the program is race free. In the traditional manner, the research can be categorised as on-the-fly, ahead-of-time, and post-mortem techniques. These techniques exhibit different strengths for race detection in programs. The ahead-of-time approaches encompass those detection techniques that apply static analysis and compile-time heuristics while on-the-fly approaches are dynamic in nature.

In well-known techniques and tools *RacerX, Locksmith* and *Chord* are based on static analysis, whereas *Eraser* uses dynamic analysis. RacerX [4] performs a flow-sensitive inter-procedural analysis to extract lock-set information and uses it for race and deadlock detection. It detects multi-threaded parts of the program and shared accesses that can be dangerous. However, RacerX assumes that code segments which are protected through locks perform parallel accesses to shared variables which may not be true if they are contained in threads which do not run in parallel. Eraser [5] employs a binary code instrumentation approach for runtime race detection, which is further extended for Java by [7] [6]. Chord [14] detects race conditions in Java programs by employing a combination of static analyses (reachability, aliasing, escape and lockset analyses) for successive reduction of memory access pairs. Locksmith [12] assumes the common approach that shared memory locations are consistently protected by a lock (*consistent correlation*). It uses a constraint-based analysis that context-sensitively infers the consistent correlation, and uses its outcome to check the proper guarding of locations by locks. However, these tools have restrictions in terms of time and space complexity. Some of them use very naive points-to information. Therefore, the need for a scalable solution is always present considering the pervasive presence of complex multi-threaded applications.

3 Terminology

Here we present the terminologies and definitions, which are used throughout this paper.

3.1 Threads

In sequential programs there is only one thread, which controls the execution of the program in a defined order. However, a sequential execution on a multi-processor system is unable to utilize the multiple processors in an efficient manner. Further, sequential programs cannot support the response time characteristics required in complex systems. These problems are alleviated by parallel programs by creating multiple threads of control. Unlike processes, threads share the same memory area and resources. Communication among threads is achieved through shared program memory. In this paper the term thread is used to refer to a POSIX thread, which is defined by the tuple

$$t = (id, attributes, start\_routine, data)$$

Each thread has a unique id, attributes (e.g., scope, state, stacksize, etc. or can be NULL if default attributes are meant to be used), a procedure to start with and some data that can be either a shared resource or specific to it.

Threads Execute in Parallel: Threads which are active at the same time may run in parallel. On a single processor system threads execute concurrently (logical parallelism), whereas on a multi-processor system they can truly run in parallel. Unless the execution of threads depends on each other and is synchronised through some mechanism, this paper considers them to be running in parallel. Further, the execution of a thread is parallel to other threads if its execution is not deterministic with respect to

those threads. If T represents the set of threads of a program and $SynchOrder$ is a relation of synchronization between threads t_i and t_j then the parallel relation is defined as follows:

$$t_i \| t_j \Leftrightarrow \neg SynchOrder(t_i, t_j)$$

The parallel relation $\|$ is symmetric, therefore if $t_i \| t_j$ then also $t_j \| t_i$. As threads are created dynamically we need a static representation for threads such that each static thread corresponds to a set of dynamic threads. In RCanalyser we chose invocation sites of thread-create functions for the representation of threads, as a call to a thread-create API might be performed in a loop with the same start routine and create multiple threads at runtime. There is no further distinction between the created threads and we consider the static thread having multiple instances. If the invocation is guaranteed to be performed only once in each execution of the program, the static thread is considered to be a single thread.

The parallel relation is reflexive *iff* multiple instances of a (static) thread are present in the program and might run in parallel.

3.2 Race Condition (RC)

A race condition can be formalised through different definitions, however for the scope of this paper the following equation defines a race for a memory location $m \in M$ as a symmetric binary relation $RC(m) \subseteq S(m) \times S(m)$. Where S is the set of all statements in a program source and $S(m)$ contains all those statements s which access a shared memory location m such that $S(m) \subseteq S$.

$$S(m) = \{s \mid m \in DEF(s) \cup USE(s)\}$$

As in standard data-flow analysis, the sets $DEF(s)$ and $USE(s)$ contain memory elements which are modified or read by the statement $s \in S(m)$. The set of shared memory locations M is defined in section 3.3. Race conditions relating to $m \in M$ are defined as

$$RC(m) = \{(s_i, s_j) \mid s_i \parallel s_j \wedge m \in DEF(s_i) \cap (DEF(s_j) \cup USE(s_j))\}$$

$s_i \parallel s_j$ represents the parallel relation between ststaments s_i, s_j and is defined in section 3.6.

3.3 Shared Accesses

Shared variables or memory locations (used interchangeably in this paper) are generally variables in parallel programs, which are accessed from more than one thread. Therefore shared memory locations are potential targets of data races. A shared memory location can include stack, global and heap variables. We do not consider volatile and atomic variables. Depending upon the requirements of the analysis compound objects and elements/components of these objects are distinguished. Additionally, a local variable whose reference is passed to other threads through a pointer also belongs to the category of shared memory locations. However, reference variables require special

consideration because their accessibility to more than one thread does not necessarily result in the same memory location during dereference. RCanalyser can perform two different analyses for shared memory location for compound types and their elements: it can either consider an access to a compound object's element as an access to the whole compound object or consider all elements as individual variables in a program. Differentiation between elements of the compound object will decrease the number of false positives. Furthermore, local static variables are also treated as global variables because they remain preserved even after a call to the enclosing procedure has finished. With this, the set of shared memory locations M is defined as:

$$\begin{aligned} M = \{m \mid \exists s_i, s_j \in S : \exists t_i, t_j \in T : \\ (s_i \in statements(t_i) \wedge s_j \in statements(t_j) \wedge \\ (t_i \neq t_j \vee (t_i = t_j \wedge t_i \in mult\_inst))) \wedge m \in Nonlocals \wedge \\ m \in (DEF(s_i) \cup USE(s_i)) \wedge m \in (DEF(s_j) \cup USE(s_j))\} \end{aligned}$$

In the above definition, T represents all threads of the program, $statements(t)$ contains all statements reachable by a thread t and $multi\_inst$ indicates if multiple active instances of a thread might exist at runtime. $Nonlocals$ are global variables accessible in all functions and procedures of a program but are not local to them. Further, they also include those references which escape their definition scope.

```
int *arrptr;

int *copy (int *p, int size)
      {
        int *tmp;
        tmp = malloc(size * sizeof(int));
        for(int i=0; i<size; i++) tmp[i] = p[i];
        return tmp;
      }

int main() {
   int a[5];
   ...
   arrptr = copy(&a, 5);
   ...
}
```

For example in the above code snippet `arrptr`, the allocated heap object and the array `a` are nonlocal objects.

Existing techniques [4,5] perform shared variable detection based on the underlying assumption that accesses to shared variables almost always follow a lock acquisition. By focusing on the lock variable relation, however, consideration of only such variables as shared can lead to false negatives of data races, because shared variables may be accessed without lock acquisition if a programmer assumes its access is safe without acquiring a lock e.g., in interactive user input. Therefore, we discover shared variables

according to the above definitions and then detect if they need to be protected or not. For details see section 4.3

3.4 Critical Section (CS)

A critical section is a part of the program that accesses shared resources and needs to be executed atomically by threads, i.e., no other thread may enter a critical section if a thread is currently executing it. Critical sections are necessary to avoid inconsistent states of shared variables during successive operations. Critical sections implement mutual exclusion mechanisms, which prevent other threads to access the shared data simultaneously. Critical sections can be implemented using synchronization instructions such as semaphores, locks or synchronized objects. During execution a thread acquires a lock and enters the critical section. Meanwhile, other threads who want to acquire the lock before entering the critical region have to wait until the lock is released. On release waiting threads attempt to obtain the lock and execute their critical code. RCanalyser considers a section of the program protected by a lock as a critical section to be extended to the point where it finds a release statement for the lock. A critical section of a thread t can be defined as a single-entry single-exit sequence of statements between lock acquire and release statements in a thread t:

$$\begin{aligned} CS(l,t) = \{ s_n \mid & \exists \pi = (s_1, \ldots, s_n) : \\ & \forall s_i \in \pi : s_i \in statements(t) \wedge \\ & s_1 \text{ locks } l \wedge \nexists s_x \in \pi : \text{ unlocks } l \} \end{aligned}$$

This definition of a critical section does not differentiate between global locks and locks which are kept as fields of dynamic data-structure. If RCanalyser does not find any shared variables we argue that it is unnecessary to implement critical sections for mutual exclusion because threads do not contain accesses to shared variables and resources.

3.5 Locks and Thread Synchronisation

Critical sections are protected by locks or other synchronisation mechanisms. However, in the scope of this paper we consider locks as a mutual exclusion mechanism used for the protection of the critical sections. The term lock is synonymously used for mutex. Before entering into a critical section a thread must obtain the associated lock and on exit it must release the lock to allow other threads to execute their critical section. If the critical section contains more than one shared variable then all these variables are protected using the lock associated with this critical section. This condition must hold for all accesses to shared variables in critical sections in other threads, otherwise inconsistent lock protection to shared variables could lead to race conditions during accesses among different threads. A thread can contain nested critical sections accessing variables shared between different threads and protected through multiple locks. The only constraint is that threads must hold a common lock before performing accesses to shared variables, locks protecting critical sections may hold locks for the contained

shared variables. All locks definitely held at a statement s in a thread t without considering path conditions can be defined as the set of $Must\_Locks$

$$Must\_Locks(s,t) = \{l \mid l \in Locks \wedge \\ \forall \pi = (t.entry, \ldots, s) : l \text{ is held at } s\}$$

In the above definition $Must\_Locks$ contains all locks held by a thread before executing a statement s. We define a function $LockCount(m,l)$ to compute the number of statements which hold the lock l and access the shared variable m.

$$Lock\_Count(m,l) = |\{s \in S(m) \mid l \in Must\_Locks(s,t)\}|$$

The $Lock\_Count(m,l)$ serves two purposes, first we use it to compute a single lock i.e., $Probable\_Lock(m)$ which must be obtained before an access to the shared variable m in a safe program. The computed lock has the highest acquisition number for shared variable m. Second, if two locks have the same acquisition number for a shared variable then both locks become plausible and are considered in $Probable\_Lock(m)$. However, during race detection accesses to the shared variable are considered unprotected due to inconsistent locking and both locks are reported to the user to decide the appropriate lock.

$$Probable\_Lock(m) = \{l \in Locks \mid \\ \forall l' \in Locks : l \neq l' \wedge Lock\_Count(m,l) \geq Lock\_Count(m,l')\}$$

3.6 Statements Exceuting in Parallel

The statements of two threads which run in parallel potentially participate in a race condition if they are not synchronised. However, if two threads run in parallel not all of their statements necessarily run in parallel. Statements accessing a shared resource can only happen in parallel if they are not synchronised through common locks, however, this does not represent their execution order i.e., a statement will happen before the other.

$$s_i \| s_j \Leftrightarrow \quad \exists t_i, t_j \in T : s_i \in statements(t_i) \wedge s_j \in statements(t_j) \wedge \\ t_i \| t_j \wedge \nexists l \in Locks : s_i \in CS(l,t_i) \wedge s_j \in CS(l,t_j)$$

Simultaneously reachable statements participate in a data race, therefore, statements in critical sections of two different threads with the same lock cannot execute concurrently. Additionally, statements cannot execute in parallel or perform parallel accesses, if there is a prior access to the must lock associated with shared variables.

4 Design and Implementation

The static recognition of race conditions in parallel programs is not a simple problem. Therefore many tools and mechanisms analyse the synchronisation structure of input

programs and perform unsafe approximations to detect the absence of necessary synchronisation [4]. RCanalyser assumes that a shared variable is consistently protected by a single lock. Hence, if different locks protect a shared variable then the lock with a higher acquisition count will be considered. RCanalyser has been designed and implemented in six different stages as illustrated in figure 1. In the following sections we discuss them in detail.

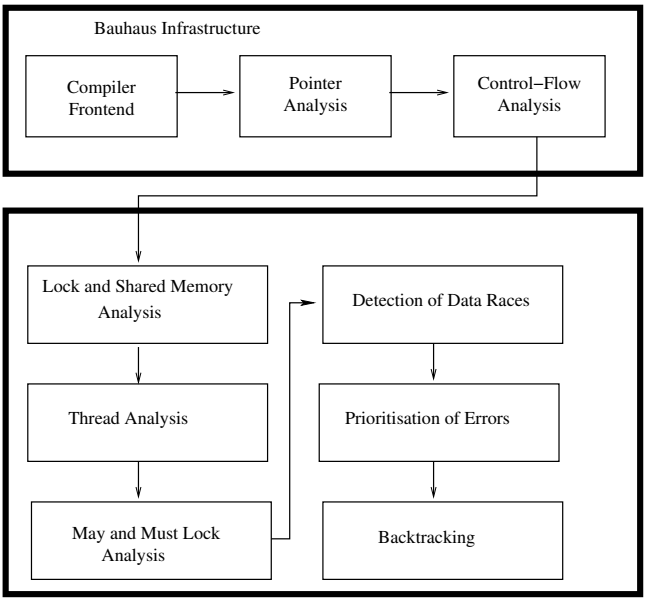

Fig. 1. The components of RCanalyser

4.1 Bauhaus Infrastructure

RCanalyser has been implemented on top of the Bauhaus infrastructure [13]. Bauhaus provides a base for implementing different high and low-level static program analyses. For our implementation we have used different Bauhaus features, e.g., an annotated abstract syntax graph (IML) for the full source program generated through language frontend and a local control-flow analysis to obtain intra-procedural control-flow graphs for all subprograms. The generic pointer analysis interface of Bauhaus provides us with different classical points-to analyses, e.g, Steensgaard [15], Das [16], and Andersen [17], which approximate the effects of pointers and determine the targets of indirect or dispatching calls.

4.2 Escape Analysis

If a majority of locks and shared variables are accessed via pointer dereferences in an analysed program, then the precision of the analysis depends directly on the quality of

the pointer analysis. By inaccuracies of the points-to analysis, the degree of false positives will be high, since apparent accesses to shared variables will be noted through the analysis, which will not occur during execution of the program. Therefore, to mitigate the imprecise effect of points-to analyses, RCanalyser provides the flexibility to use different points-to analyses to achieve different levels of precision. Further, RCanalyser performs a thread specific escape analysis to improve the quality of error reports. RCanalyser checks if a local variable's reference is ever assigned to a pointer during execution and marks such locals as escaped references. If a reference never escapes its scope a race condition on such variable is considered spurious and ignored, because a flow- and context-insensitive points-to analysis result for a pointer can nevertheless contain this variable's reference.

4.3 Lock and Shared Variable Analyses

Shared variable analysis detects all variables which are accessed (read, write) in at least two threads. Therefore, all accesses to global and reference variables are computed on a per-thread basis. If a global variable is read or written in a statement by a thread it is immediately marked as a shared access. An access to a reference variable is registered as an access to all global or local variables to which it possibly refers. However, if a local variable's address is never assigned to a reference variable it cannot contribute in a data race and accesses to it are simply ignored. Shared variable analysis determines read and write accesses on a variable and stores this information for each thread in the program. Later this information is used to determine conflicts between threads.

The lock analysis is one of the most important parts of RCanalyser. RCanalyser performs a flow- and context-sensitive lock analysis as defined in [4]. The task of this analysis is to compute $Must\_Locks$ for each statement. It is important to note that it is not determined which lock in the program is always set for a statement, but all the acquired locks are computed. This benefits the analysis by providing the information if a lock is obtained on all paths in the program. However, due to flow- and context-sensitivity its runtime complexity can increase exponentially. The analysis is performed in a depth first search order. To determine all possible locks RCanalyser computes which locks are acquired before a procedure call and visits the control-flow-graph of the procedure to determine which locks are acquired and released during the procedure call. This result is propagated back to the call site and $Must\_Locks$ information is updated along the analysed path. If a procedure call contains more than one possibility to exit, then information propagated to the call site can contain different locks or locksets of $Must\_Locks$. Further analysis then has to compute $Must\_Locks$ within the context of each lock or lockset. The resulting $Must\_Locks$ of this procedure is saved in a cache to avoid a re-computation, in case the procedure is called again with the same locks. In the same manner $Must\_Locks$ information for statements is computed and saved in the cache.

4.4 Thread Analysis

Thread analysis computes which threads in the program can run in parallel. RCanalyser considers threads in parallel with other threads if their start and end is in the range

of the other thread's execution span. To determine the execution span of threads, a global logical clock is defined in such a way that it is incremented each time a thread starts or gets joined by other threads independent from its runtime execution behavior. Execution span is calculated flow sensitively in topological ordering from the value of the global clock at a thread's creation call to its corresponding join call. Therefore, the termination of a thread in relation to other threads is defined by the clock count value when its join call is found. After execution spans are computed for all threads, a run in parallel relation between threads is computed for each thread. All threads which are alive during the execution span of a specific thread are considered to potentially run in parallel. Due to the symmetric nature of the run in parallel relation, the computation complexity is reducible.

4.5 Variable Lock Relation

The next step in RCanalyser is to determine $Must\_Locks$ for each shared variable as defined in section 3.5. For each shared variable we count the number of statements which access the shared variable and hold a specific lock l. The lock with the highest number of statements is considered as must lock for the shared variable. If the number of statements and lock acquisition count is equal then shared variables are consistently protected. If a statement appears in two critical sections protected through different locks which include accesses to a shared variable such that one critical section is nested in a thread and the other is not then there will be no data race on this variable if a common lock is held by both threads. This common lock will be considered as a must lock for this variable.

4.6 Detection of Data Races

Having computed the information about threads, shared variables and must locks, a potential data race can be determined by using the equation defined in section 3.2. If RCanalyser does not find a required lock for a shared variable v which is held at all parallel statements accessing v, we consider accesses to this variable as data races. However, read accesses are not considered as a data race, at least one single write access on a shared variable is necessary for a data race. It is possible that a thread makes a procedure call after obtaining a lock and another thread without acquiring a lock calls it and itself holds the lock for a shared variable, such a case will not fall in the category of race condition. Because a common lock is always held before the access is performed. Furthermore, accesses on a shared variable with different locks held in different threads will also indicate a race condition.

4.7 Prioritization and Backtraces

RCanalyser computes a prioritisation between detected data races depending upon the type of the shared variables and the threads' run in parallel information. The criterion followed for prioritisation based on severity and probability are

Prioritization: Severe Errors

- If a global variable is involved in a data race it has a high priority.
- The priority of a data race in a thread where must lock for a variable is not acquired is high as compared to the thread which acquires a lock for a shared variable but run in parallel with this thread.

Prioritization: Probable Errors

- If the involved variable is accessed in a loop the priority is high.
- For reference variables with many possible destinations the priority is low.
- If the accessed object is of a compound type, i.e., a structure or an array, the priority is low because the exact location index of the accessed element may be imprecisely calculated.
- When two threads who cannot possibly run in parallel access a shared variable without acquiring a common lock the priority of data race is also low.

The error reports can be viewed using RCanalyser interactive shell in the order of their priority. Backtraces report the path along which a data race can occur. If the path contains conditional statements the trace report represents which branch is considered.

5 Experimental Results and Evaluation

RCanalyser delivers results to our expectations, i.e., all locks are computed, and shared variables are recognised. It computes the parallel threads and must locks for each variable and partially excludes the variable initializations from data races, and successfully prioritizes the errors, depending upon their nature. Nevertheless, due to the conservative nature of analyses implemented by RCanalyser it can also report false positives. It can report a data race on a compound object accessed through a pointer even if different elements are involved during accesses. However, this can be mitigated by enabling the field sensitive points-to analysis for program variables. Similarly, it is undecideable to distinguish between different elements of an array object. An access to an array element is considered as an access to the complete array. RCanalyser performs a context- and flow-sensitive lock analysis. Therefore, the tool handles function calls precisely and it does not consider infeasible paths due to invocations of functions. Nevertheless, infeasible paths might be considered in local contexts because the tool does not evaluate conditions and always considers all paths after a branch.

Currently RCanalyser can be configured to detect races in POSIX/Apache Runtime Environment based multi-threaded C programs. The experimental results of our test suite downloaded from *sourceforge.net* are listed below. We have used *Das* analysis to compute points-to information for these programs. The result clearly shows the effective discovery of shared variables and number of threads in each program (columns *Sh Vars* and *Threads*). The results also illustrate that RCanalyser is scalable and can be applied to benchmarks with up to 6.1 kloc (same as Locksmith [12] tool.)

The column *Warnings* shows the number of locations where a race condition might manifest. The reported numbers are higher as compared to others [12], because, after locating the first unprotected access to a shared variable we record all following locations

Benchmark	KLOC	Threads	Sh Vars	Warnings	Unguarded	Real Races
aget-0.4	1.6	4	13	49	47	11
smtprc-2.0.3	6.1	3	10	200	180	7
ctrace-1.2	1.8	3	14	53	50	6
tplay-0.6.1	3.9	3	5	47	46	2

Fig. 2. Test Results

as well. The figures in column *Unguarded* describe program statements performing access to a shared variable without lock acquisition. These figures do not contain the locations where a lock has been held before access to a shared variable. It might be possible that these warnings are only about some shared variables which are targets of a data race. *Real races* presents the number of data races found after a careful inspection of unguarded program locations.

5.1 Discussion of Results

Due to the conservative nature of the static analyses in RCanalyser, it will safely find all potential race conditions in a program. But it may also report false positives which come from over-approximations done in the base analyses and RCanalyser itself.

In Figure 2 we can see that the differences of the number of *Warnings* and the number of *Real races* are still high. A detailled inspection lets us conclude that many of the reported false positives are manifestations of the features not yet present in our implementation.

A great deal of inaccuracy has its cause in an inadequate handling of (conceptual) reference parameters in our base analyses. The context- and flow-insensitive pointer analyses which are currently used in Bauhaus merge the targets of a reference parameter for multiple invocations of a subprogram. Therefore RCanalyser currently does not distinguish between different invocation contexts. The usage of a context-sensitive pointer analysis would bring a great benefit and is planned as a future work.

Another reason for false positives was the lack of path conditions in RCanalyser. The analysis currently considers all branches in the control flow of a program as equally feasible. A first step towards an improvement is the integration of a copy propagation analysis which lets us detect if a condition is always true or false. This helps us to remove dead code which can not contribute to a race condition. We expect another improvement from the implementation of a same-value analysis which determines if the values of different conditions are definitely the same. With that we are able to exclude infeasible paths as a reason for race conditions. A same-value analysis for thread variables will also mitigate the decision process of which thread gets joined or canceled at a given point, because thread identifier are integers and may get another value during program execution.

6 Conclusion and Future Work

Obtaining information about which threads run in parallel has significant applications in the detection of anomalies such as race conditions and deadlocks. Further, C-programs

make intensive use of pointers, which make it difficult to find data races in parallel threads. Algorithms which can compute this information effectively and precisely are of great value. Previous approaches analyse programs without significant consideration of points-to information. We developed RCanalyser, a tool for the conservative detection of data races in multi-thread/parallel programs using $Must\_Locks$ analysis with flexibility to incorporate different points-to analysis mechanisms. The framework can handle multi-threaded programs of practical nature. The results have shown that due to the conservativeness of our technique and unavailability of a flow- and context-sensitive points-to analysis, it can produce false positives. However, RCanalyser can handle all types of pointer used in C programs. In the future, we plan to improve our mechanisms to correctly identify the accessed components of compound data types and thread escape analysis to optimize the precision and reduce the number of false positives. Additionally, we would like to implement a data-flow analysis for parallel programs in our framework to detect updates which may change thread identifiers. A further goal is to make RCanalyser more scalable to handle larger program code up 50-100K and incorporate the most used synchronization techniques e.g., condition variables, signal wait etc.

Acknowledgement

We would like to thank our colleagues at ISTE/PS department at university of Stuttgart and reviewers for their insightful comments on this paper.

References

1. Netzer, R.H.B., Miller, B.P.: What are race conditions?: Some issues and formalizations. ACM Letters on Programming Languages and Systems 1, 74–88 (1992)
2. Sun Microsystems, Inc.: Multithreaded Programming Guide (2002), `http://docs.sun.com/app/docs/doc/806-6867/`
3. Sterling, N.: WARLOCK - A Static Data Race Analysis Tool. In: USENIX Winter Technical Conference, pp. 97–106 (1993)
4. Engler, D., Ashcraft, K.: RacerX: Effective, Static Detection of Race Conditions and Deadlocks. In: Proceedings of the 19th ACM Symposium on Operating Systems Principles, pp. 237–252. ACM Press, New York (2003)
5. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: A Dynamic Data Race Detector for Multi-Threaded Programs. In: Proceedings of the 16th ACM Symposium on Operating Systems Principles, pp. 27–37. ACM Press, New York (1997)
6. Choi, J.D., Lee, K., Loginov, A., O'Callahan, R., Sarkar, V., Sridharan, M.: Efficient and Precise Datarace Detection for Multithreaded Object-Oriented Programs. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 258–269. ACM Press, New York (2002)
7. von Praun, C., Gross, T.R.: Object Race Detection. In: Proceedings of the 16th ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications, pp. 70–82. ACM Press, New York (2001)
8. Naumovich, G., Avrunin, G.S.: A Conservative Data Flow Algorithm for Detecting All Pairs of Statements that May Happen in Parallel. In: Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 24–34 (1998)

9. Masticola, S.P., Ryder, B.G.: Non-concurrency Analysis. In: Proceedings of the fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 129–138 (1993)
10. Burgstaller, B., Blieberger, J., Mittermayr, R.: Static Detection of Access Anomalies in Ada95. In: Pinho, L.M., González Harbour, M. (eds.) Ada-Europe 2006. LNCS, vol. 4006, pp. 40–55. Springer, Heidelberg (2006)
11. Raza, A.: A Review of Race Detection Mechanisms. In: Grigoriev, D., Harrison, J., Hirsch, E.A. (eds.) CSR 2006. LNCS, vol. 3967, pp. 534–543. Springer, Heidelberg (2006)
12. Pratikakis, P., Foster, J.S., Hicks, M.: LOCKSMITH: Context-Sensitive Correlation Analysis for Race Detection. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 320–331. ACM Press, New York (2006)
13. Raza, A., Vogel, G., Ploedereder, E.: Bauhaus – A Tool Suite for Program Analysis and Reverse Engineering. In: Pinho, L.M., González Harbour, M. (eds.) Ada-Europe 2006. LNCS, vol. 4006, pp. 71–82. Springer, Heidelberg (2006)
14. Naik, M., Aiken, A., Whaley, J.: Effective static race detection for java. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 308–319. ACM Press, New York (2006)
15. Steensgaard, B.: Points-to Analysis in Almost Linear Time. In: Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 32–41. ACM Press, New York (1996)
16. Das, M.: Unification-based Pointer Analysis with Directional Assignments. In: Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 35–46 (2000)
17. Andersen, L.O.: Program Analysis and Specialization for the C Programming Language. PhD thesis, DIKU, University of Copenhagen (1994)