Fault Detection in Multi-Threaded C++ Server Applications

Arndt Mühlenfeld
Institute for Software Technology
Graz University of Technology
Inffeldgasse 16b/2
A-8010 Graz
Austria
amuehlen@ist.tugraz.at

Franz Wotawa
Institute for Software Technology
Graz University of Technology
Inffeldgasse 16b/2
A-8010 Graz
Austria
wotawa@ist.tugraz.at

ABSTRACT

Due to increasing demands in processing power on the one hand, but the physical limit on CPU clock speed on the other hand, multi-threaded programming is becoming more important in current applications. Unfortunately, multi-threaded programs are prone to programming mistakes that result in hard to find defects, mainly race-conditions and deadlocks. The need for tools that help finding these faults is immanent, but currently available tools are either difficult to use because of the need for annotations, unable to cope with more than a few 10 kLOC, or issue too many false warnings. This paper describes experiments with the freely available tool Helgrind, results obtained by using it for debugging a server application comprising 500 kLOC. We present improvements to the runtime analysis of C++ programs that result in a dramatic reduction of false warnings.

Categories and Subject Descriptors

D.1.3 [**ProgrammingTechniques**]: Concurrent Programming; D.1.5 [**ProgrammingTechniques**]: Object-oriented Programming; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Experimentation, Languages, Measurement, Performance

Keywords

data races, race conditions, debugging, parallel programs, synchronization, multi-threaded programming, object-oriented programming, static-dynamic co-analysis

1. INTRODUCTION

Hardware is becoming more powerful every year. Recent developments have pushed CPU speeds on the edge of manageable frequencies, further improvements are only possible by multiplying the number of CPUs (or CPU cores on one chip). Today's software does not benefit from multi-processor machines unless it utilizes multiple threads. Thus, the need for "multi-threading" applications is increasing.

In addition, multi-threading is a powerful paradigm that may help partitioning programs into logical threads of execution, thereby making interleaved operations easier to model. But parallel execution introduces possible faults that are difficult to detect and localize by conventional debugging techniques, due to the unpredictable and therefore nondeterministic execution order of concurrent programs.

Without proper synchronization two major classes of faults unique to concurrent execution can be identified. *Data races*, where unsynchronized access to shared memory locations results in inconsistent data, and *dead locks*, where two or more threads block each other, mostly because of cyclic dependencies.

Hence, developers need tools that help in finding these kind of faults. Proposed solutions for multi-threading fault detection include model-checking, static analysis and runtime analysis.

Model-checking suffers from the problem of state space explosion. In spite of all efforts (e.g. counter example guided abstraction refinement [3]) it is still not applicable to large programs. Static analysis techniques try to locate possible faults by applying heuristics to the source code. Detecting all feasible data races by static analysis is well-known to be an NP-hard problem [11]. Another problem in the C++ domain is the lack of freely available correct and reliable parsers that generate suitable abstract representations.

Runtime-methods scale well, but only faults on the path of execution are taken into account, therefore the detection of all data races is impossible, too. But it is possible to detect and report all apparent data races on the execution path. An efficient lock-set based runtime-algorithm called Eraser [14] was implemented in the open-source tool Valgrind and is thereby available for all Linux-x86 based environments. Unfortunately, at least for C++ applications, the number of falsely reported possible data races is too large, making the tool difficult to use since every reported location has to be checked by hand.

In general, the algorithm is easy to use, because it does not require special tuning or annotations by the programmer making it a perfect tool for every days use. Though, the average programmer will not use a tool that generates hundreds of spurious warnings that he has to analyze by hand, which is a time consuming and error-prone task. Hence, it is necessary to reduce the number of false positives while keeping the original unsupervised reliable behavior. In addition, programmer written annotations should not be needed.

The solution is to combine both, static and runtime analysis, by annotating the program automatically and transparently to the programmer in order to provide the runtime method with additional knowledge gathered from the structure of the source code. While these hints reduce reporting of false positives, they are not necessary. Therefore, it is still possible to analyze programs, where only parts of the source code are available.

This work presents results from experiments where the Eraser implementation in the tool Helgrind was applied to an existing network server application. Two improvements were made: One to better simulate actual hardware behavior (i.e. bus locking). Another to cope with effects introduced by C++ specific implementation issues. This drastically reduces the amount of false positives reported, making the tool usable for the debugging of large C++ applications. In particular, the amount of false positives removed by our improvements during our experiments was in the range of 65% to 81% of the total number of warnings.

This paper is organized as follows: Section 2 contains an overview of runtime methods for fault detection in multithreaded programs and the runtime detection implemented in Helgrind is presented in greater detail. In Section 3, we present our method of source-code annotation in order to make runtime analysis more accurate and describe the general environment for the experiments. Results from our experiments are presented in Section 4. Finally, Section 5 concludes the paper with a general discussion of our results.

2. FAULT DETECTION

In this section, we first present some definitions of faults unique to concurrent programs. Then an overview of runtime methods for detecting these faults is given, followed by a more detailed description of the algorithms implemented in the freely available tool Helgrind, which was used as a basis for our experiments.

2.1 Definitions

Faults that are unique to concurrent programs are *data races* and *deadlocks*.

A *deadlock* is defined as a state, where each thread in a collection of two or more threads tries to acquire a lock already held by one of the other threads in the collection. Hence, the threads are blocked on each other in a cyclic manner.

A *data race* occurs, when two or more concurrent threads access a shared location which is not protected by a proper synchronization construct (e.g., a monitor) and at least one of them modifies the contents of the accessed component.

This definition is a bit restrictive, in fact, it describes the locking policy enforced by the Eraser algorithm. The weakness of the definition is that the program can reach an inconsistent state, even if every single access to a shared location is protected by proper synchronization.

This will become clear in the following example:

Suppose, we have a data structure containing two elements, let us say the date-of-birth and age of an arbitrary person. Because the current age of the person could be calculated by counting the time elapsed from date-of-birth until now the variables depend on each other. In addition, there is a synchronization object protecting access to the data. Two setter-methods exist, one to set the date-of-birth and the other to set the age. Now, when updating the structure we first write the new date of birth followed by a call to set the new value for age. Both methods use synchronization to protect their field accesses. Therefore the rule, that every single access to the shared location is protected by synchronization, is satisfied. Nevertheless, it is possible to reach an inconsistent state between two write accesses that depend on each other, because the lock is released in-between.

Even when every single access to a data structure is protected by a lock, it might be possible to reach an inconsistent state for the data. In [1] this is called a *high-level data race*, because the notion of a data race does not seem to be powerful enough, In other works [4, 15], the problem is tackled by the definition of *atomicity* and atomicity violations.

While usually not resulting in actual faults, the locking strategy itself has an impact on the performance of the application. At worst, all data are protected by a single (global) lock, resulting in unnecessary blocking of independent threads. Or, more general, heavy usage of a global resource by all threads degrades performance and drastically reduces the speed-up in multi-processor systems.

2.2 Dynamic Methods

Most dynamic methods are based on the lock-set algorithm Eraser or on Lamport's happens-before relation [7].

The algorithm implemented in *Eraser* [14] tries to identify the locks that guard a shared location by maintaining a lock-set containing all locks that are active at each access. Therefore, it is able to detect violations of the locking discipline, to require that each shared location be protected by the same lock (or set of locks) on each access to it.

A method, that was developed to detect data races in the DSM System Millipage is the algorithm DJIT [6]. It utilizes vector time frames and access logging to check the happens-before relation between concurrent accesses to a shared location. It relies on the assumption of an underlying coherent system and detects only the first apparent data race.

The main advantage of the lock-set algorithm is the ability to detect all possible data races that exist on the execution path. On the other hand it sometimes gives too many false detections. DJIT tries to locate only apparent data races, hence detects data races on a subset of shared locations that are reported by the lock-set approach and misses some real

data races. Therefore, Multi-Race [13] tries to improve the data race detection capabilities by combining enhanced versions of Lock-set and DJIT into a common framework.

In [12] the authors combine a lock-set based data race detector with a vector clock based happens-before relation check on Java synchronization primitives. Actions on these primitives are viewed as events that impose an order on memory accesses between them. Unfortunately, neither their assumption that unsynchronized memory writes become visible in causal order is true on all SMP systems, nor is the relation between signal and wait operations on conditions strong enough to impose the assumed order.

A major disadvantage of online techniques is that they slow down the execution of the application under observation significantly. Consequently, their use requires adaption of the environment to support slower reactions. Principally, on-the-fly checkers can work post mortem, too, reducing the performance impact due to the online calculations. But they still need logging of the execution trace. Hence, offline techniques suffer from their need of large amounts of data.

Nevertheless, on-the-fly checkers scale well with program size. They are not complete as only faults on the execution path are found, but already in use in industrial software development. One implementation is the freely available tool Helgrind. We used it as a base for our experiments and the underlying algorithm is described in the following section.

2.3 Runtime Analysis with Helgrind

2.3.1 The Tool

Helgrind is a Valgrind tool [10, 9] for detecting data races in C and C++ programs. It uses the Eraser algorithm [14] and improvements from Visual Threads [5] in order to reduce reporting of false positives.

Valgrind is a binary instrumentation framework for Linux ELF Binaries and was at first used as a memory checker. Starting with version 2 the application was divided into a core that generates intermediate code from an executable binary and interprets the code using just-in-time compilation for speed improvements, and a skin or tool that instruments the intermediate code before it is executed and interprets the results.

This makes Valgrind a powerful and flexible tool for all kinds of runtime checking.

In order to suppress false reportings in subsequent runs of the checker, it is possible to write a so-called suppression-file that contains report-type and call-stack-patterns of locations that are false positives or part of code that is not modifiable (e.g., third-party libraries).

2.3.2 Basic Algorithm (Eraser)

Eraser [14] is an algorithm that checks a given program whether each access to a shared memory location is protected by proper synchronization. In this implementation it works only for programs that use the POSIX-Threads library, because calls to that library are intercepted in order to track the status of the memory and the thread system.

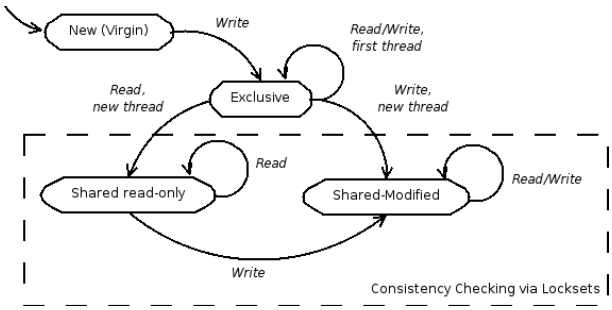

Figure 1: **States for a memory location. After being allocated, it starts in state NEW. During initialization it is owned EXCLUSIVEly by the allocating thread until another thread reads from or writes to the location. Then, one of the SHARED states is reached and the lock-set is initialized for consistency checking. Nevertheless, race conditions are only reported in the SHARED-MODIFIED state.**

The basic synchronization object in POSIX-Threads is a mutex (**mut**ual **ex**clusion), with methods to acquire (*lock*) and release (*unlock*) it. Only one thread can hold a lock at any given time, all other threads, that try to lock it, are blocked, until the mutex is released again.

In order to avoid the need for annotations, the Eraser algorithm tries to infer the mutex that protects a shared memory location, and if the location is unprotected issues a warning. Therefore a set is maintained for every shared memory location, that contains the intersection of the sets of locks that were held during all accesses to it.

The basic algorithm in pseudo-code:

Let $locks\_held(t)$ be the set of locks held by thread t.
For each v, initialize $C(v)$ to the set of all locks.
On each access to v by thread t,
  set $C(v) := C(v) \cap locks\_held(t)$;
  if $C(v) = \{\}$, then issue warning.

This should find all possible data-races, but results in too many false positives. One major drawback is, that initialization and read-shared data is not handled properly.

Some shared variables are initialized once by one thread and subsequent accesses by other threads are only reads, hence a lock is not needed. Therefore, states were introduced in the Eraser algorithm, that enable it to deal with these situations (cf. Figure 1).

The lock-set is not initialized as long as only one thread uses the memory location. When another thread accesses the memory location, the lock-set is initialized with all active locks and the algorithm reports the next write access that results in an empty lock-set.

Now, a thread that allocates a memory location, owns it until another thread accesses the same memory location. Hence, the allocating thread may initialism the shared variable and then share it with other threads for reading only without resulting in a warning by the race-detector.

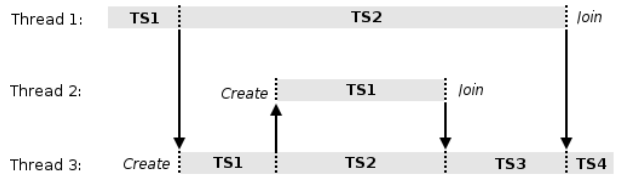

Figure 2: **A thread consists of thread segments, that are separated by thread-create and -join operations. Memory accesses that are limited to non-overlapping thread segments are still exclusive even if not done by a single thread.**

There are cases, where the algorithm is now incomplete, because of its dependence on the actual interleaving. It is possible, that a possible data race exists, where the first read access by another thread can occur before the initialization is finished. It is not detected by the algorithm, because in the observed interleaving, all writes took place before the first (shared) read access.

According to the authors this drawback is out-weighted by the reduction in the amount of false positives reported by the modified algorithm. Repeated tests with different test data (resulting in different interleavings) could help find such data-races, if they exist.

An extension for read-write locks that is presented in the original Eraser algorithm is **not implemented** in Helgrind. Albeit it would be useful in certain circumstances.

When a variable enters the Shared-Modified state, checking is as follows:

Let $locks\_held(t)$ be the set of locks held in any mode by thread t.
Let $write\_locks\_held(t)$ be the set of locks held in **write** mode by thread t.
For each v, initialize $C(v)$ to the set of all locks.
On each **read** of v by thread t,
  set $C(v) := C(v) \cap locks\_held(t)$;
  if $C(v) = \{\}$, then issue warning.
On each **write** of v by thread t,
  set $C(v) := C(v) \cap write\_locks\_held(t)$;
  if $C(v) = \{\}$, then issue warning.

Thread Segments

Another typical scenario that results in a warning is as follows:
A thread allocates memory, initializes it by setting it to something useful and fires up a second thread, that should work on the data. After a while the first thread waits for the second thread to finish, before it uses the memory again.

Thus, the memory is shared between threads, but at any time only one thread accesses it. The ownership is passed onto the second thread until it terminates.

This observation is used by VisualThreads ([5]) to further reduce the number of false positives by introducing thread segments (cf. Figure 2).

Instead of a thread being owner of a shared variable that is in EXCLUSIVE state, it is now a thread segment that owns it. Then, whenever another thread accesses the memory, it is checked, whether the thread-segments overlap. If not, the new thread-segment becomes the new owner instead of the variable switching into SHARED state.

The modification to the Eraser algorithm:

1. When data d is marked as EXCLUSIVE, associate it with the thread segment id of the current thread instead of the thread id.

2. If data d is marked as EXCLUSIVE to thread segment TS_i, and is being touched by TS_j, and TS_i *happens before* TS_j in the graph, then instead of moving the data to one of the shared states, associate d with TS_j. The state remains EXCLUSIVE.

Deadlock Detection

VisualThreads (and Helgrind) contains detection for explicit and potential deadlock.

Explicit Deadlock

```
CheckDeadlock(object, mark)
{
    // if we find current mark, then cycle detected
    if object.mark == mark then report deadlock;

    object.mark = mark;

    for each o on which object depends
        CheckDeadlock(o, mark);
}
```

3. IMPROVEMENTS AND EXPERIMENTS

After early experiments with Helgrind we found two improvements that help by reducing the number of false positives.

3.1 Improvements

First, we corrected the implementation of the hardware bus lock in Helgrind. It was implemented by using a special mutex, that is locked on every explicit invocation of the LOCK prefix. According to Intel's i386 specification read operations do not require to use the LOCK prefix. it is only needed for writes. A correct implementation is more like a read-write lock. This required the implementation of read-write locks in Helgrind. Now, the modified version of the tool internally supports rw-locks. As a benefit, support for the corresponding POSIX API could be added easily.

Helgrind is able to work without the need of source code. Hence, the detection process is independent of the programming language. Unfortunately, many of the analyzed warnings turned out to be caused by C++ specific code.

When the destructor of an object is called every destructor of its parents classes is called prior to actually releasing the memory associated with the object. The destructor of the super-class should only see the properties of its class and therefore the environment has to be changed in order to reflect this change in properties and virtual method pointers.

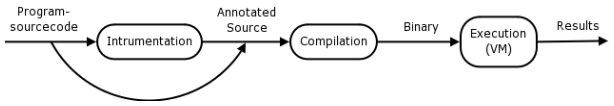

Figure 3: **Data flow of the debugging process. All or part of the source code is analyzed and instrumented. The resulting binary is executed on the VM Valgrind with data race detection.**

This change is done by writing to a location in the object's memory, hence resulting in a warning, because Helgrind does not know anything about objects and destructors and that accesses to an object's memory in its destructor can not result in a data race on itself.

Since the number of false positives due to polymorphic object destruction code is rather large and identifying them by hand is too much work, it is necessary to suppress them automatically. It is done by annotating every delete operation in the source code of the program in order to mark deleted memory for the race detection as exclusive. Parts of the program, where the source code is not available will not benefit from this annotation therefore still resulting in false positives, but the overall number of false reportings is reduced.

The method does not need whole program analysis and is easily integrated into the build process. The annotation could be inserted into production code, because the user-space call to Helgrind is a no-op under normal program execution with negligible execution time.

Annotation is done on-the-fly and it is easily removed from the build process, since the source code is not modified, neither by the annotation tool nor by the programmer. The whole process is described in greater detail in the following section.

3.2 Debugging Process

When checking a program using the original Helgrind algorithm, it is not necessary to compile the source code in a special way. Symbol information is needed for convenience. Without the debug symbols, Helgrind is not able to print source line information or the function names on the call stack for locations, where a fault is suspected. To check a program for errors, it can be run unmodified with Helgrind. A second step to interpret the results is necessary.

The instrumentation necessary to improve Helgrind requires an additional first step. As shown later, the instrumentation can be done during the build process without visible modifications to the source code and, more important, without user interactions, thereby retaining the ease of use of the debugging technique. After adding automatic source code annotation to the process, the modified debugging process consists of three parts (cf. Figure 3).

Instrumentation

All available source code could be instrumented to help reduce false reportings of the runtime analysis. For now, only delete operators are annotated to mark the memory of the destructed object as *exclusive, destroyed*.

For an example of instrumented code, see Figure 4, but note, that it is presented in a state that does not exist in the real process, because the preprocessing was omitted for clarity.

The runtime analysis works without source code instrumentation, but the results are better with instrumentation.

Execution

The program is executed on the virtual machine with test data from an automated test suite. The runtime analysis is based on the tool Helgrind. Results are written to a log file.

Analysis

The log file is analyzed by the user in order to verify if the reported *possible* data race are in fact data races, and if they are, corrections to the program are made.

```
/* Original source code */
void g(char * p)
{
    delete p;
}
```

```
/* Annotated source code */
#include <valgrind/helgrind.h>
namespace {
    template <class Type>
    inline Type * ca_deletor_single(Type * object)
    {
        VALGRIND_HG_DESTRUCT(object, sizeof(Type));
        return object;
    }
}
void g(char * p)
{
    delete ca_deletor_single ( p);
}
```

Figure 4: **An example for the annotation. The argument for operator `delete` is passed through a function which announces the memory to be destroyed to the race detector. The macro VALGRIND_HG_DESTRUCT expands to a sequence of mnemonics that do nothing under normal execution, but are recognized by the interpreter of Helgrind as a special function call.**

3.3 Setup and Environment

SIP Proxy Server

All experiments are carried out on a Linux x86 system. The application under test is a signaling server application for the Session Initiation Protocol (SIP) that is used for Voice-over-IP (VoIP) phone networks and utilizes POSIX-Threads for multi-threading and synchronization primitives.

The concurrent pattern in use is "thread-per-request", i.e., for each request a new thread is created. This fits well into the thread-segment improvement from VisualThreads, because the ownership is passed to the worker thread by thread creation. Although, synchronization is already done by locks, it is necessary to check the application for data races and dead locks, as it has shown non-deterministic failures when run with multiple threads.

The application is built from several hundred kLOC of C++ code, hence experimental tools, only written as proof-of-concept, are not applicable. Furthermore, there are no restrictions on the usage of C++ language constructs ruling out many of these experimental tools that rely on the usage of only a subset of the C++ language (e.g., to keep the parser simple).

Instrumentation

For instrumentation, the C++-parser ELSA is used. ELSA is based on Elkhound, a GLR-Parser generator[8]. ELSA builds an abstract syntax tree that is used for source code analysis and annotation.

The input for the parser must be preprocessed, because external files are not read by the parser, all needed information must be included in the source file. Hence, the instrumentation and compilation process has three stages.

First, the GNU compiler is used to preprocess the source file. Then the parser reads the preprocessed source file and generates the annotated source file.

In the third and last step, the compiler generates object code from the annotated source file.

This can be done in a shell script that replaces the compiler call during the build process, making the instrumentation transparent to the build tools and the programmer.

Since instrumentation adds only user-space calls to the VM that are, besides a small delay, without effect under normal execution, it could be done in every build process. A drawback would be that build times are increased, because of the additional second stage (instrumentation).

Test Bed

Debugging with data race detection requires a testing environment, preferably an automated test suite to guarantee reliable repeatability of the test runs. A difficulty lies in the different timing behavior of the program under test since execution on the virtual machine slows it down by a factor of 20-30. Therefore, in some cases timeouts have to be adapted to the changed response times. Furthermore, the virtual machine in itself is single-threaded, thus adding more processors does not help either.

In our environment, eight of eleven test cases on the SIP proxy server ran without changes, they were used for the experiments. The basic request patterns are delivered to the application by an automated test suite. The main utility of this test suite is SIPp, a tool for SIP load testing.

For data races an on-the-fly checker (Helgrind) is used, deadlocks on Mutex locks are detected by the application using a timeout while trying to acquire a lock inside the lock-function, but the race-checker does dead-lock detection, too, therefore the application level detection is not needed.

4. RESULTS

The test-suite with eight test cases was run with three different configurations. After running it with the original

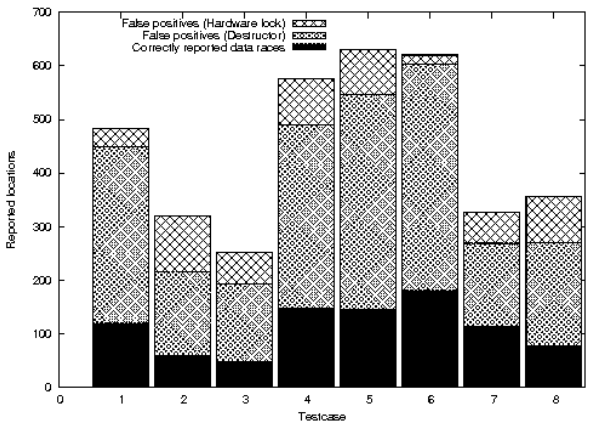

Figure 5: **Results of the debugging process. Eight simple test cases where run with different configurations of the race checker. The two upper parts of a bar denote false positives, the smaller (top) part counts warnings due to misinterpretation of the hardware bus lock, the bigger part due to accesses in the destructor of an object.**

Test case	Original	HWLC	HWLC+DR
T1	483	448	120
T2	319	215	60
T3	252	194	49
T4	576	490	149
T5	631	547	146
T6	620	604	181
T7	327	269	115
T8	357	270	78

Figure 6: **Results of the debugging process. The numbers are number of reported "possible data race locations under different configurations. First, the originally implemented algorithm in Helgrind was used. Second, corrections were made to the emulation of the hardware bus lock in Helgrind. The last column contains the results after additionally applying source code annotations to delete operations.**

Helgrind the number of reported possible data races was recorded and after inspecting individual warnings, it was clear that most of the warnings are false positives resulting from a wrong implementation of the hardware bus lock semantics and automatic modifications of objects on destruction. Both cases are explained below.

A second run was done with a corrected implementation of the hardware bus lock (HWLC), where many warnings disappeared. This run did not need source code instrumentation.

In the third run (HWLCD+DR), the source code was annotated to mark memory that is up to be destroyed just before the destructor is called. This further reduces the amount of reported possible data races by more than a half in all cases (Figure 6).

Still, the number of reported data races is significant and most of them are real synchronization failures, but some faults form groups that stem from the same origin. That

```
map<string,DomainData*> & ServerModulesManagerImpl::getDomainData()
{
        MutexPtr mut(m_pMutex); // Guard
        return m_DomainData;
}
```

Figure 7: **Unprotected attribute due to return of reference.**

means, it is generally a good idea to rerun the test suite after fixing a problem. Then, all warnings related to the corrected defect will disappear and do not have to be considered again. Additionally, faults possibly introduced by the correction generate new warnings.

An issue arising when using Helgrind with the GNU C++ Standard Library, is false reporting due to the memory allocation strategy in the standard container objects. Memory is reused internally and accesses to the reused memory regions are reported as data races, even though the accesses are separated by freeing and allocating, as Helgrind does not know anything about them. Fortunately the allocation strategy of the GNU Standard C++ Library is configurable with environment variables, this must be done prior to calling Helgrind.

4.1 True Positives

During our experiments, we found a number of real bugs in the analyzed program. Since the application has about 500 kLOC, it is not always easy to decide, whether a reported warning is a true defect, a false warning or just a benign race. Nevertheless, we found a lot of real defects in the program, a selection of bugs that seem to be common is presented here.

One of the first reported data races was in the application's deadlock detection code. Unfortunately this code was not easy to change in order to remove the race condition, therefore, it was disabled for further experiments.

4.1.1 Initialization and Termination Order Problems

Another found error was a problem in the order of initialization, i.e., a thread is started before parts of the data structures it uses are initialized. This error was not directly found by the tool, but occurred due to the different schedule when running the program with instrumentation. In the "usual" environment, the fault would not occur often enough to attract attention.

On program shutdown, another data-race occurred, because a data structure was destroyed, before a thread using it terminated.

4.1.2 Synchronization Problems

Simple locking sometimes results in unnoticed bugs (Figure 7). A quick glance at the source-code reassures the programmer, that everything is fine, but even though a lock is held in all methods of a class, there is a data access without proper synchronization, for example a method returning a reference to an attribute instead of its contents.

In the case found in the program under test, the attribute is a map, therefore its use should be protected by the lock. This bug requires to rewrite the function and all functions that use it, because returning a reference to the internal data structure prevents proper protection. To change that, the signature of the function changes and all calls to the function change, too.

4.1.3 Improper Use of System Functions

In a multi-threading environment, the use of some of the system functions is not safe. Especially all functions that use static data or, even worse, return a pointer to static data are not thread-safe. The usage of some of these functions in the application resulted in possible data races reported by the tool.

A remark on the glib-c manual page acknowledges this: "The four functions `asctime()`, `ctime()`, `gmtime()` and `localtime()` return a pointer to static data and hence are NOT thread-safe"

4.2 False Positives

Three kinds of false positives predominate the results. The first two kinds described here were already addressed by our improvements, while others still remain.

4.2.1 Destructor of Derived Classes

Helgrind reports a locking policy violation in the destructor of a few classes. These destructors are mainly default-destructors, generated by the compiler, but the important common property is that they all belong to derived classes.

When the destructor of an object is called every destructor of its parents classes is called prior to actually releasing the memory associated with the object. The destructor of the super-class should only see the properties of its class and therefore the environment has to be changed in order to reflect this change in properties and virtual method pointers.

This change is done by writing to a location in the object's memory, hence resulting in a warning, because Helgrind does not know anything about objects and destructors and that accesses to an object's memory in its destructor can not result in a data race on itself.

This holds under the assumption that data is not accessed after calling delete and thereby invoking its destructor. Violations of this assumption are detected by ordinary memory checking tools, therefore it is not a special case for multi-threaded programs and could be neglected during data race detection.

4.2.2 Hardware Bus Lock

In this simple example (Figure 8) Helgrind reports a possible data race at the assignment in Line 22.

The report is caused by the shared access to the reference counter. The operation is protected by a hardware bus-lock, but the read accesses preceding this write are not using the lock, therefore the lock-set is empty.

The technique not to initialize the lock-set until the first occurrence of a write does not help here, because there already was a write to this memory location in the other thread (`workerThread`).

```
1  /*! \ file    stringtest . cpp
2   *  \ brief   Test  shared  read-access  of  std :: string - objects .
3   */
4
5  #include <string>
6  #include <pthread.h>
7
8  void * workerThread(void * arguments)
9  {
10         std::string text = *(std::string *)arguments;
11         return 0;
12 }
13
14 int main()
15 {
16         std::string text("contents");
17
18         pthread_t thread_id;
19         pthread_create(&thread_id, 0, workerThread, &text);
20
21         sleep(1);
22         std::string text_copy = text;  // <- reported  conflict
23
24         void * result = 0;
25         pthread_join(thread_id, &result);
26
27         return 0;
28 }
```

Figure 8: **Example for a shared object of type std::string. Strings are often implemented with reference counting. Thus, when a string object is copied, it is sometimes necessary to modify the source object by adding the new reference.**

```
==19670== Possible data race writing variable at 0x1D6F9168
==19670== at 0x1D548451: std::string::_Rep::_M_grab(std::allocator<char>
      const&, std::allocator<char> const&) (in /usr/lib/gcc-lib/i686-pc-
      linux-gnu/3.3.2/libstdc++.so.5.0.5)
==19670== by 0x1D548517: std::string::string(std::string const&) (in /usr/
      lib/gcc-lib/i686-pc-linux-gnu/3.3.2/libstdc++.so.5.0.5)
==19670== by 0x804879F: main (stringtest.cpp:22)
==19670== Address 0x1D6F9168 is 8 bytes inside a block of size 21 alloc'd by
       thread 1
==19670== at 0x1D4A8433: operator new(unsigned) (vg_replace_malloc.c:133)
==19670== by 0x1D545A98: std::__default_alloc_template<true, 0>::allocate(
      unsigned) (in /usr/lib/gcc-lib/i686-pc-linux-gnu/3.3.2/libstdc++.so
      .5.0.5)
==19670== by 0x1D54B3F7: std::string::_Rep::_S_create(unsigned, std::
      allocator<char> const&) (in /usr/lib/gcc-lib/i686-pc-linux-gnu
      /3.3.2/libstdc++.so.5.0.5)
==19670== by 0x1D54C13E: (within /usr/lib/gcc-lib/i686-pc-linux-gnu
      /3.3.2/libstdc++.so.5.0.5)
==19670== Previous state: shared RO, no locks
```

Figure 9: **Example for a warning issued by Helgrind. The source of the warning is in the GNU Standard C++ library, the method `m_grab()` adds a reference to a string object.**

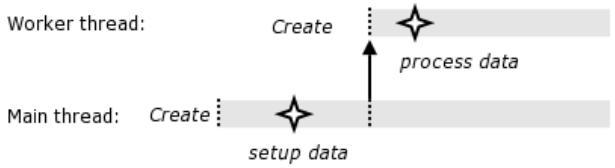

Figure 10: **In the thread-per-request pattern, accesses to the message data are separated by thread-create and -join operations. The algorithm infers by comparing the thread segments that accesses are still exclusive (per thread segment).**

The detection algorithm does not take into account that the operations are atomic. The read and write operations of the reference counter are atomic, because it is an integer value and all writes are protected by a bus locking prefix. It is impossible to derive that from simple observations, as the reference counter is part of the structure that contains the data.

If Helgrind supported read-write locks, the hardware bus-lock could be emulated as a read-write lock, being held for reading in every read access and locked for writing, when the lock prefix is used. This would emulate the behavior of bus-locks more accurately and remove the spurious warning in the string class.

As already described, we implemented this correction successfully.

4.2.3 *Transition of Ownership*

The method used in the application is to spawn a new thread for every request. When the amount of outstanding requests at any time becomes more than the maximum allowed number of threads that run in parallel, the application will fail.

High performance server applications have to deal with many parallel request and usually work by putting all incoming request into a queue, while having a fixed number of threads (i.e. a thread pool) fetching data from the queue for processing. That also avoids the overhead of creating and destroying a thread for each request.

Thus, for the application under test it is planned to utilize patterns that use thread pools in one way or the other.

This leads to the problem that the race detection algorithm will report more false positives.

In the case of the thread-per-request pattern, accesses to the data that are passed to the worker thread are clearly separated (cf. Figure 10).

When using thread pools the situation changes. Thread creation is done before the data is initialized and passed to the worker thread, hence the data race detection algorithm reports a warning on the first write to this data. The accesses are clearly separated by the put and get operations on the message queue, but the algorithm does not detect that (cf. Figure 11).

4.3 False Negatives

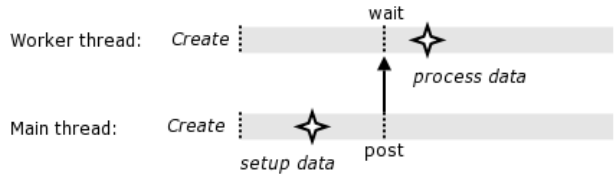

Figure 11: **In patterns using thread pools, accesses to the message data are separated by message put and get operations. (post-wait) The algorithm used by Helgrind does not take into account that accesses are still exclusive.**

The introduction of states in the Eraser algorithm in combination with delayed lock-set recording reduces the detection capabilities of the algorithm. One of its greatest strength is the ability to report data races independent of execution order.

Suppose, one thread writes a shared location without acquiring a lock, whereas another thread does the same, but coincidentally holds a lock during that access. If the first access takes place before the second one, no warning is reported, because lock-set initialization is delayed until the access of the second thread, and on that access a lock is held.

If a different schedule leads to another execution order, the (possible) data race is found and reported. But this is not guaranteed to happen in the development environment, and may cause failures after delivering the software to the customer.

In fact, during the experiments such cases were found in the source code and they have not been reported by the testing process. This indicates, that it should be addressed in future enhancements of the algorithm.

4.4 Summary

The problems of the algorithm we encountered (false positives and false negatives) fall into three categories:

- C++ implementation specific issues (Destructor). This was fixed by source code analysis.
- Hardware related interpretation (CPU lock prefix). After correcting the implementation of the hardware bus lock, this was fixed too.
- The execution order imposed by higher level synchronization primitives was not taken into account (false positive) or it was falsely assumed to be guaranteed (false negative).

Higher level synchronization based on the low level constructs is a field for further improvements

4.5 Performance

Unlike static methods, dynamic analysis scales well for large programs. Program length is not very important, because the main parameter for space and time complexity is the length of the execution trace.

Generally, most runtime techniques can execute on-the-fly or offline. Both have their advantages. On-the-fly analysis usually has a significant negative impact on the execution speed of the analyzed program, offline analysis needs information logging which may result in heavy memory usage. On the one hand, on-the-fly techniques are preferred, when the amount of information that otherwise had to be logged is large, on the other hand, logging and offline analysis is necessary, when runtime analysis would slow down the program to uselessness.

In our case, where each access to a memory location had to be logged, offline analysis would be almost impossible for long execution traces. Thus, the time consumed by analysis directly reduces the execution speed of the observed program. Furthermore, since Valgrind executes binaries on a virtual machine, even without instrumentation program execution is slow.

Execution of the program with analysis using the presented algorithm is 20-30 times slower than when run without Helgrind. When comparing this number to other works, where the reported slowdown by Eraser-like algorithms is around 2-3 (or even less), one has to take into account, that these results are obtained in environments, where the program is always executed on a virtual machine like the Java VM (as in [2, 12]) or Microsoft's Common Language Runtime (as in [16]). If run on Valgrind, the program is slowed down by a factor of 8-10 without instrumentation. Thus, our results are comparable to previous works.

5. CONCLUSION

Nowadays, many implementations of on-the-fly race detection algorithms exist. Unfortunately, most academic proof-of-concept implementations are not applicable to real-world applications. At least, the need to cope with more than a subset of C++ is a knockout criterion, because to our knowledge no parser is freely available that is able to generate an abstract syntax tree for the full ISO C++ language.

Furthermore, for a concrete implementation in a tool additional criteria decide whether it is useful or not. Usually, programmers are not willing to spend much time on parameter tweaking or analyzing tool output. To be most useful, a tool should not require the user to be an academic. Hence, the analyzing process must be easy to setup and the results should contain very few if not zero false warnings. A good example is the open source memory checker Valgrind, that is widely accepted by programmers in different environments because of its ease of use and the usefulness of its output.

Our experiments with the freely available tool Helgrind (which is part of Valgrind) showed, that it is a good base for research, but generates too many false warnings to be usable in production environments. We analyzed hundreds of warnings generated by the tool for a large commercial server application. The results contained many false positives, but we found real bugs, too. Based on our results we made improvements to the algorithm and analyzed the consequences. That is, we added knowledge about language specific properties to the runtime analysis by unsupervised annotation of object delete operations in the source code.

We have shown, that our improvements have a significant effect on the amount of false positives reported by the lock-set algorithm implemented in Helgrind. That is, the correct interpretation of the hardware bus lock on the x86-architecture and special marking of objects that are about to be destroyed. Furthermore, our improvements do not complicate the debugging process much, in most cases only a configuration switch for the build process has to be set. And no manual source code annotations are necessary.

Further improvements could have a similar impact on the detection abilities of the data race checker, but require more effort. The weaknesses with regard to common multi-threading patterns should be addressed. Common concurrent patterns often rely on higher level constructs for synchronization that the lock-set algorithm is unaware of.

6. REFERENCES

[1] C. Artho, K. Havelund, and A. Biere. High-level data races. *Softw. Test., Verif. Reliab.*, 13(4):207–227, 2003.

[2] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *PLDI*, pages 258–269, 2002.

[3] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In E. A. Emerson and A. P. Sistla, editors, *CAV*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.

[4] C. Flanagan and S. N. Freund. Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs (summary). In *IPDPS*. IEEE Computer Society, 2004.

[5] J. J. Harrow. Runtime checking of multithreaded applications with visual threads. In K. Havelund, J. Penix, and W. Visser, editors, *SPIN*, volume 1885 of *Lecture Notes in Computer Science*, pages 331–342. Springer, 2000.

[6] A. Itzkovitz, A. Schuster, and O. Zeev-Ben-Mordehai. Toward integration of data race detection in DSM systems. *J. Parallel Distrib. Comput.*, 59(2):180–203, 1999.

[7] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[8] S. McPeak and G. C. Necula. Elkhound: A fast, practical glr parser generator. In E. Duesterwald, editor, *CC*, volume 2985 of *Lecture Notes in Computer Science*, pages 73–88. Springer, 2004.

[9] N. Nethercote. *Dynamic Binary Analysis and Instrumentation*. PhD thesis, University of Cambridge, Trinity College, 2004.

[10] N. Nethercote and J. Seward. Valgrind: A Program Supervision Framework. *Electr. Notes Theor. Comput. Sci.*, 89(2), 2003.

[11] R. H. B. Netzer and B. P. Miller. What are race conditions? some issues and formalizations. *LOPLAS*, 1(1):74–88, 1992.

[12] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *PPOPP*, pages 167–178. ACM, 2003.

[13] E. Poznianski and A. Schuster. Efficient On-the-Fly Data Race Detection in Multithreaded C++ Programs. In *IPDPS*, page 287. IEEE Computer Society, 2003.

[14] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, 1997.

[15] C. von Praun and T. R. Gross. Static detection of atomicity violations in object-oriented programs. *Journal of Object Technology*, 3(6):103–122, 2004.

[16] Y. Yu, T. Rodeheffer, and W. Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. In A. Herbert and K. P. Birman, editors, *SOSP*, pages 221–234. ACM, 2005.