ANaConDA: A Framework for Analysing Multi-threaded C/C++ Programs on the Binary Level*

Jan Fiedor and Tomáš Vojnar

IT4Innovations Centre of Excellence, FIT, Brno University of Technology, Czech Republic

Abstract. This paper presents the ANaConDA framework that allows one to easily create dynamic analysers for analysing multi-threaded C/C++ programs on the binary level. ANaConDA also supports noise injection techniques to increase chances to find concurrency-related errors in testing runs. ANaConDA is built on top of the Intel's framework PIN for instrumenting binary code. ANaConDA can be instantiated for dealing with programs using various thread models. Currently, it has been instantiated for programs using the pthread library as well as the Win32 API for dealing with threads.

1 Introduction

Due to the arrival of multi-core processors to common computers, multi-threaded programming has become a standard in all widely used programming languages. Such programming, however, is more demanding and brings much more space for errors. Hence, adequate tools for discovering concurrency-related errors are highly needed.

One way to find errors in multi-threaded programs is *dynamic analysis* that monitors the execution of a program and tries to extrapolate the witnessed behaviour and issue warnings about possible errors even when no error is really witnessed in the given execution. However, monitoring the execution of a program can be quite challenging and programmers might spend more time writing the monitoring code than by writing the analysis code itself. In this paper, we present the ANaConDA framework which is a framework for adaptable native-code concurrency-focused dynamic analysis built on top of PIN [7]. The goal of the framework is to simplify the creation of dynamic analysers for analysing multi-threaded C/C++ programs on the binary level. The framework provides a monitoring layer offering notification about important events, such as thread synchronisation or memory accesses, so that developers of dynamic analysers can focus solely on writing the analysis code. In addition, the framework also supports noise injection techniques to increase the number of interleavings witnessed in testing runs and hence to increase chances to find concurrency-related errors.

The general ideas behind the framework and preliminary experiments with it have been presented in [2]. In the present paper, apart from mentioning some recent additions to the framework, we focus more on how to write an analyser using the framework, how

* This work was supported by the Czech Science Foundation (project P103/10/0306), the Czech Ministry of Education (projects COST OC10009 and MSM 0021630528), the EU/Czech IT4Innovations Centre of Excellence project CZ.1.05/1.1.00/02.0070, and the internal Brno University of Technology projects FIT-S-11-1 and FIT-S-12-1.

to get some useful information which may help the user in locating an error, and how to use the tool. As the framework can be instantiated to support various multithreading libraries, we also describe some concrete instantiations, in particular, the instantiation for `pthreads`, already used for the experiments in [2], and a new instantiation for Win32 API. Finally, we discuss several real-life experiments done with the framework.

As for related tools, there exist many frameworks which may be used to simplify the creation of dynamic analysers for Java programs. The closest to ANaConDA is IBM ConTest [1] which inspired some parts of the design of ANaConDA. RoadRunner [3] is another framework very similar to ANaConDA. Both of these frameworks can monitor the execution of multi-threaded Java programs and provide notification about important events in their execution to dynamic analysers built on top of the frameworks. CalFuzzer [5] is an extensible framework for testing concurrent programs which can also be used to create new static and dynamic analysers and to combine them. Chord [8] is another extensible framework which might be used to productively design and implement a broad variety of static and dynamic analyses for Java programs. When dealing with C/C++ programs, the options are much poorer. One tool somewhat related to ANaConDA is Fjalar [4] which is a framework for creating dynamic analysers for C/C++ programs. However, Fjalar is primarily designed to simplify access to various compile-time and memory information. It does not provide any concurrency-related information. Moreover, it is build on top of Valgrind [9], which brings several disadvantages as discussed in Section 3.

2 Monitoring Multithreaded C/C++ Programs on the Binary Level

As was mentioned in the introduction, monitoring C/C++ programs can be quite difficult, especially when the monitoring is done on the binary level. One of the problems to be dealt with is monitoring of function execution. This is because the monitoring code has to cope with that the control can be passed among several functions by jumps. Hence, the control can return from a different function than the one that was called. Another problem is that the monitoring code must properly trigger notifications for various special types of instructions such as atomic instructions, which access several memory locations at once but in an atomic way, or conditional and repeatable instructions, which might be executed more than once or not at all. Further, some pieces of information about the execution of instructions or fuctions (such as the memory locations accessed by them), which are crucial for various analyses, may be lost once the instruction or function finishes its execution, and it is necessary to explicitly preserve this information for later use. Finally, in order to support various multithreading libraries, the analysers must be abstracted from the concrete library used. Possible solutions to the these problems were discussed in [2].

A problem that has not been considered in [2] is that the information needed for analysis is not the only information useful for the users. When the analyser detects an error, it should provide the users as much information as possible to help them localise the error. Retrieving information about the executed code, such as names of variables or locations in the source code, can give the users some information about the error. However, this information is often not sufficient since it may be difficult to know how the program got to the variable or location where the error was detected. A much better help to the user is a backtrace to the erroneous part of the program.

ANaConDA currently supports backtraces equivalent to the ones given by the Linux `backtrace()` function, which contain the return addresses of the currently active function calls. The return addresses are stored on the call stack in the corresponding stack frames. The top stack frame's address can be obtained from the base pointer register, and each stack frame also contains the previous value of the base pointer, referring to the previous stack frame. By following the chain of base pointers, we can extract the return addresses and create a backtrace although we have to be careful when processing the stack frames as sometimes (e.g., during the initialisation of the program) the base pointer register may be used for other purposes and might point somewhere else than to a stack frame. The advantage of this approach is that we do not need to monitor every function call in the program and update the backtrace constantly. We are constructing the backtrace on demand, i.e., only when the analyser explicitly requests it, and we only need to know the value of the base pointer register, which can be retrieved with a negligible overhead. The only drawback is that the program must properly create the stack frames, which may sometimes not be true if some optimisations are used.

3 Implementation, Current Instantiations, and Usage

The ANaConDA[1] framework is an open-source framework written in C++ on top of PIN [7]. There are several reasons motivating the use of PIN as a binary instrumentation backend. First, PIN performs dynamic instrumentation, i.e., it instruments a program in the memory before it is executed. This means that the binary files of the program are left untouched. This is especially important when dealing with libraries as it allows one to transparently use an instrumented version of a library and simultaneously use the library as usual in other programs. PIN can also be used on both Linux and Windows, compared to Valgrind which is Linux-only, which allows a much wider range of programs to be analysed. Of course, PIN is primarily developed for use with Intel binaries. However, if the binary code does not contain any special AMD-only instructions, PIN works fine even for AMD binaries. Another advantage of PIN is that it preserves the parallel execution of threads of the analysed multi-threaded program. Valgrind, on the contrary, serialises thread execution [9], which may unnecessarily slow down the program and also the analysis as the analysis code usually runs in these threads too.

Instantiation. The ANaConDA framework abstracts analysers built on top of it from the specific multithreading library used, but it of course cannot do that without any information about the library. As explained in more detail in [2], the user must specify: (1) the names of the functions performing various thread-related operations, (2) the indices of parameters holding the synchronisation primitives the functions operate with, and (3) the `Mapper` objects used to abstract the synchronisation primitives to numbers uniquely identifying them. Abstraction of synchronisation primitives is necessary because their representation varies across various libraries, but analysers need to work with them in a uniform way.

For example, if we use the `pthread` library and want to get notifications about lock acquisitions and releases, we have to specify that the `pthread_mutex_lock`

[1] `http://www.fit.vutbr.cz/research/groups/verifit/tools/anaconda`

```
pthread_mutex_lock 1 addr()
pthread_mutex_trylock 1 addr()
```

(a) Lock acquisitions (`lock` file)

```
__pthread_mutex_unlock_usercnt 1 addr()
```

(b) Lock releases (`unlock` file)

Fig. 1. An example of the configuration of monitoring lock operations in the `pthread` library

and `pthread_mutex_trylock` functions are performing the lock acquisitions and the `__pthread_mutex_unlock_usercnt` function the lock releases. This is done by adding the names of these functions to the `lock` and `unlock` configuration files, respectively. All of these functions are taking the lock as the first parameter, and because locks are objects of the `pthread_mutex_t` structure, we can use the ANaConDA framework's built-in mapper object `addr` to convert the addresses of these objects into numbers uniquely representing them. To give this information to ANaConDA, we have to specify the index and the name of the `Mapper` object right after the name of the corresponding monitored function as can be seen in Fig. 1. The instantiation for signaling conditions and waiting on them is similar, we just have to instruct the framework to monitor the `pthread_cond_signal`, `pthread_cond_broadcast`, and `pthread_cond_wait` functions by inserting the appropriate information to the `signal` and `wait` configuration files.

As for the Win32 API, there is no function that performs purely lock acquisitions. Instead, the `WaitForSingleObject` function is used taking a generic `HANDLE` as the first parameter and performing a lock acquisition only if the `HANDLE` represents a lock (it may also represent, e.g., a thread or an event). In this case, we have an alternate way to tell ANaConDA when a function performs a lock acquisition. We can specify that the `WaitForSingleObject` function is a generic wait function whose behaviour depends on the type of the synchronisation primitive passed to it and then name a function which creates or initialises new locks. The framework then remembers which synchronisation primitives are locks because they were created by the user-identified lock creation/initialisation function. Subsequently, when a generic wait function (like `WaitForSingleObject`) is called, it will first determine what kind of synchronisation primitive its parameter represents. If it is a lock, it will properly trigger the lock acquisition notifications. In particular, in Win32 API, locks are created by the `CreateMutex` function which returns a `HANDLE` representing the lock. Configuring lock releases is much simpler as they are performed by a dedicated `ReleaseMutex` function which takes the lock (`HANDLE`) as the first parameter. As the `HANDLE` is in fact a generic pointer, we can also use the `addr` mapper object here to transform it into a unique number.

The Win32 API has no functions for signaling conditions and waiting on them. If such operations are needed, the users usually implement the operations themselves or use some libraries like `pthread-win32` implementing them. However, as ensuring that the functions performing these operations will trigger the corresponding ANaConDA notifications is as easy as adding a few lines to the appropriate configuration files, the framework does not have any problems with the users using their own custom functions for these operations, which illustrates the generality of the framework.

Another problem with the Win32 API is that some of the functions that need to be monitored are jumping at the beginning of other monitored functions. In this case, PIN executes the monitoring code inserted before such functions, and if no special care was

taken, the analyser would get a notification about a single event multiple times. The solution could seem to be easy as one could, e.g., think of simply specifying that one of the functions should not be monitored. However, the functions often have exactly the same names, so one cannot so easily differentiate between them. The framework solves this problem by checking if the stack pointer changed when a monitored function is about to be executed, and it does not issue a notification if its value remained the same as that means that nobody called the function, and the control must have jumped to it.

Usage of ANaConDA. To analyse a multi-threaded C/C++ program using ANaConDA, one first has to write (or get) an analyser to be used. The analyser must have the form of a shared object (in Linux) or a dynamic library (in Windows) which contains a set of functions that ANaConDA should call when a specific event, such as a lock acquisition, occurs in the program being analysed. The analyser has to register the callback functions for the events it needs to be notified about. This is done by calling the appropriate registration functions (provided by ANaConDA) in the `init()` function of the analyser, which ANaConDA executes once the analyser is loaded. For example, to be notified about lock acquisitions and releases, the analyser has to register its callback functions using the `SYNC_AfterLockAcquire` and `SYNC_BeforeLockRelease` functions, respectively.

Performing the actual analysis is then quite simple. One just needs to execute the PIN framework with ANaConDA as the *pintool*[2] to be used and specify the analyser which should perform the desired analysis together with the program which should be analysed. Noise injection can be enabled and configured in the `noise` section of the `anaconda.conf` configuration file. Currently, only the *sleep* and *yield* noise is supported, but the user may use different noise injection settings for the read and write accesses and also for each of the monitored functions. The slowdown of the execution of the analysed program is similar to Fjalar, i.e., around 100 times. Note, however, that the slowdown is mainly due to PIN and depends on many factors such as the amount of instrumentation inserted, the amount of information requested by the analyser, the amount of noise injected into the program, etc.

4 Experiments

A set of preliminary experiments with the framework was done in [2] where we analysed more than 100 student projects implementing a simple ticket algorithm (100–500 lines of code) under the `pthread` library. The projects passed all the tests originally used to mark them, but we still found errors in around 20 % of them using a simple data race detector called AtomRace [6], which we use in the tests discussed below too.

To test whether ANaConDA can handle really large and complex programs, we have used it to analyse the `Firefox` browser (more than 3 million lines of code) which uses the `pthread` library. We did not find any severe or unknown errors. We did, however, find several data races which are left in the code since they are considered harmless. Considering the size of the program, the fact that it is thoroughly checked for data races regularly, and also that we used a very simple data race detector and performed only

[2] A pintool can be thought of as a PIN plugin that can modify the code generation process inside PIN, i.e., it determines which code should be executed and where in the monitored program.

a very limited set of tests since we did not have any automatic test suite to use, we consider these results to still be quite promising.

We further analysed the `unicap` libraries for video processing, which also use the `pthread` library and are considerably smaller (about 40k lines of code) which allowed us to perform a larger number of tests. We have found several (previously unknown) data races in the `libunicap` and `libunicapgtk` libraries. Two of the data races can be considered severe as they may cause a crash of the program which uses these libraries. In both cases, one thread may reset a pointer to a callback function (i.e., set it to `NULL`) in between of the times when another thread checks the validity of this pointer and calls the function referenced by it, which can cause an immediate segmentation fault. We are currently preparing to report these errors to the developers using the ANaConDA's recently added backtrace support that can provide a rather detailed information where and why the error occurred.

Finally, we also successfully tested the framework on several Windows toy programs (100–500 lines of code). An application to larger programs is planned for the near future.

5 Conclusion

We have presented ANaConDA—a framework simplifying the creation of dynamic analysers for analysing multi-threaded C/C++ programs on the binary level. We have shown how to instantiate it for several widely used multithreading libraries and demonstrated on several case studies that it can handle even large real-life programs. With the help of the framework, we were able to write a simple analyser in a day and successively find several errors with it, which shows the usefulness of the framework.

References

1. O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur. Framework for Testing Multi-threaded Java Programs. In *Concurrency and Computation: Practice and Experience*, 15(3-5):485–499, Wiley & Sons, 2003.
2. J. Fiedor and T. Vojnar. Noise-Based Testing and Analysis of Multi-threaded C/C++ Programs on the Binary Level. In *Proc. of PADTAD'12*, ACM Press, 2012.
3. C. Flanagan and S. N. Freund. The RoadRunner Dynamic Analysis Framework for Concurrent Programs. In *Proc. of PASTE'10*, ACM Press, 2010.
4. P. J. Guo. A Scalable Mixed-Level Approach to Dynamic Analysis of C and C++ Programs. Master's thesis, Department of EECS, Cambridge, MA, May 5, 2006.
5. P. Joshi, M. Naik, C.-S. Park, and K. Sen. CalFuzzer: An Extensible Active Testing Framework for Concurrent Programs. In *Proc. of CAV'09*, LNCS 5643, Springer, 2009.
6. Z. Letko, T. Vojnar, and B. Křena. AtomRace: Data Race and Atomicity Violation Detector and Healer. In *Proc. of PADTAD'08*, ACM Press, 2008.
7. C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proc. of PLDI'05*, ACM Press, 2005.
8. M. Naik. Chord: A Static and Dynamic Program Analysis Platform for Java Bytecode. URL: `http://code.google.com/p/jchord`.
9. N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proc. of PLDI'07*, ACM Press, 2007.