Noise-based Testing and Analysis of Multi-threaded C/C++ Programs on the Binary Level

Jan Fiedor, Tomáš Vojnar
FIT, Brno University of Technology, IT4Innovations Centre of Excellence
Božetěchova 2, Brno
CZ 612 66, Czech Republic
{ifiedor, vojnar}@fit.vutbr.cz

ABSTRACT

This paper aims at allowing noise-based testing and dynamic analysis of multi-threaded C/C++ programs on the binary level. First, several problems of monitoring multi-threaded C/C++ programs on the binary level are discussed together with their possible solutions. Next, a brief overview of noise injection techniques is provided along with a proposal of improving them using a fine-grained combination of several noise injection techniques within a single program. The proposed ideas have been implemented in a prototype way using the PIN framework for Intel binaries and tested on a set of multi-threaded C/C++ programs. The obtained experimental evidence justifying the proposed solutions and illustrating the effect of various noise settings in the context of multi-threaded C/C++ programs is discussed.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming; D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Concurrency, Dynamic Analysis, Noise Injection, Testing

Keywords

Concurrency, Dynamic Analysis, Noise Injection, Testing

1. INTRODUCTION

A massive expansion of multi-core processors in the past decade accelerated development of software products that use a multi-threaded design to utilise the available hardware resources. As most of today's programming languages allow programmers to create multi-threaded programs, they are becoming more and more common. Nevertheless, writing correct multi-threaded programs is significantly harder than writing single-threaded programs because errors in concurrency are not only easy to create but also very difficult to discover and localise due to the non-deterministic nature of multi-threaded computation.

A desire to achieve a high performance is often one of the main motivations for using C/C++. It is thus natural to utilise the currently commonly available multi-processor systems in C/C++ programs to boost their performance as much as possible. However, much like programming in C/C++ is often considered more difficult than programming in some other higher programming languages, writing multi-threaded C/C++ programs can be more difficult and hence more error prone too. Moreover, even the diagnostics of errors that happen in running programs tends to be more difficult for C/C++ programs than, e.g., for some interpreted languages such as Java or C#. This is simply because the interpreter usually has more information to provide to the user than what one can obtain from a crashed C/C++ program.

Despite the very active research in the area of formal analysis and verification, software testing still belongs among the most common ways of discovering errors in programs. Testing multi-threaded programs is, however, much more difficult than testing sequential programs as errors in concurrency can manifest under very rare scheduling circumstances only. That is why techniques like *noise injection* and *dynamic analysis* were developed. Noise injection attempts to increase the chances to see the rare executions leading to an error by disturbing the scheduling of threads of a program in order to force it to execute differently than usual. Dynamic analysis takes a different approach of monitoring the execution of a program and trying to extrapolate the witnessed behaviour and issue warnings about possible errors even when no such error is really witnessed in the given execution. Moreover, these two approaches can of course be combined too.

In order to be able to monitor the execution of a program and perform some dynamic analysis as well as to insert some noise into the execution of the program, a need to execute some additional code in some places of the execution of the original program arises. There are several levels at which one can insert such additional code to the program—namely, at the source code level, at the level of the intermediate code, or at the binary level.

Inserting the code at the binary level has one big advantage over the other approaches in that it does not need to have the source files of the program being analysed, which is particularly important when dealing with libraries whose

PADTAD '12 Minneapolis, MN USA

source files might not be available even for the developers of the program under test. Another advantage might be that this kind of instrumentation is more precise in that we can insert the code exactly where we want it to be executed, and the placement is not affected by any optimisations possibly made by the compiler. These advantages of course come at the cost of that we may possibly lose access to various pieces of high-level information about the program (names of variables, etc.). However, even such information can be available if we have the debugging information present in the program, and moreover, we can also get access to some low-level information, like register allocations, which might be important for some analyses.

In this paper, with the aim of allowing the above advantages to be exploited by developers of multi-threaded C/C++ programs, we concentrate on noise-based testing and dynamic analysis of multi-threaded C/C++ programs on the binary level. We identify several typical problems that arise when trying to monitor the execution of multi-threaded C/C++ programs at the binary level (such as monitoring function execution, retrieving information about the executed code, dealing with atomic, conditional, or repeated instructions, as well as supporting different C/C++ multithreading libraries), and we discuss their possible solutions. The problems are first discussed on a general level and then a more detailed discussion based on our prototype implementation of the proposed ideas using the PIN [10] framework for Intel binaries follows. Since we concentrate on noise-based testing and dynamic analysis, we also provide a brief overview of existing noise injection techniques, and moreover, based on our experience, we propose their improvement using a fine-grained careful combination of different noise techniques within a single program. Last but not least, we present results of multiple experiments with a prototype implementation of our proposals on a set of C/C++ student projects. We use these results to justify the proposed solutions and also as a basis for a discussion of the possible influence of various noise injection settings on dynamic analysis of multi-threaded C/C++ programs.

Related Work. As many concurrency errors appear only under very specific scheduling circumstances, various ways to deal with this problem in testing were developed. One approach is to influence the scheduling of threads of a program to actually see the interesting scheduling leading to an error by inserting a random noise into its execution as it is done in the IBM Concurrency Testing (ConTest) tool [3], or by systematically forcing some conditions on the program interleaving [14]. Another possibility is to systematically explore all schedules up to some number of context switches as it is done, e.g., in the Microsoft CHESS tool [11].

Another often used approach is to use dynamic analysis that tries to extrapolate the witnessed behaviour of a program and warn about errors whose occurrence under different scheduling circumstances seems possible from the observed behaviour of the program. There exist dynamic analysers for various concurrency errors like data races [13, 7], atomicity violations [9], or deadlocks [1]. A problem is that the extrapolation may introduce false alarms.

As for binary instrumentation frameworks, many were developed over the years. Most of them, such as PIN [10] or Valgrind [12], control the whole execution of a program, instrumenting its code just before it is executed (using just-in-time compilation) or when it is loaded into the memory. Other frameworks, e.g., PEBIL [6], instrument the program's binary file in advance and do not participate in its execution in any way. There are also frameworks which combine these two approaches—e.g., VMAD [4] inserts several versions of the instrumented code into the program's binary file and then chooses at run-time which version will be executed.

Plan of the paper. The rest of the paper is organised as follows. In the next section, various typical problems of monitoring the execution of C/C++ programs at the binary level are discussed. Section 3 discusses the possibilities of utilising noise injection techniques to help various dynamic analyses with the detection of errors in multi-threaded C/C++ programs, including several proposals of improving the use of noise injection. Section 4 describes a prototype implementation of the proposed ideas. In Section 5, an experimental evaluation of the proposed solutions is provided. Section 6 then concludes the paper and discusses some of the interesting directions for future work.

2. MONITORING AT THE BINARY LEVEL

In this section, we discuss several typical problems that arise when trying to monitor and analyse the execution of a multi-threaded C/C++ program compiled to a binary form. In particular, after a brief introduction of the types of binary instrumentation frameworks that one can use to insert execution-monitoring code, we discuss the problems of monitoring function execution, retrieving information about executed instructions, handling atomic, conditional, and repeatable instructions, and abstracting concrete synchronisation primitives for the analysers to be used. For these problems, we analyse possible solutions, trying to stay on a rather general level. In Section 4, we will then present some further implementation details concerning the use of the proposed solutions in our prototype tool.

2.1 Instrumentation Frameworks

There exist several frameworks for binary instrumentation which can be used to insert execution-monitoring code to a program. They might be divided into two groups—static binary instrumentation and dynamic binary instrumentation frameworks.

Static binary instrumentation frameworks, like, e.g., PEBIL [6], insert execution-monitoring code to a program by rewriting the object or executable code of the program before the program is executed, thus modifying the content of the program's binary file. *Dynamic instrumentation frameworks*, like, e.g., PIN [10] or Valgrind [12], insert execution-monitoring code to a program at run-time, leaving the program's binary file untouched.

An advantage of static binary instrumentation is that it does not suffer from the overhead of instrumenting the code of a program every time it is executed. On the other hand, it cannot handle constructions like self-modifying or self-generating code, which is not known before the program actually executes. On the contrary, dynamic binary instrumentation is slower, but it can cover all the code that is executed by a program. Furthermore, since the binary file of the program is not modified in any way, the instrumentation is more transparent to the user who can run some (possibly lengthy) analysis on the program and, at the same

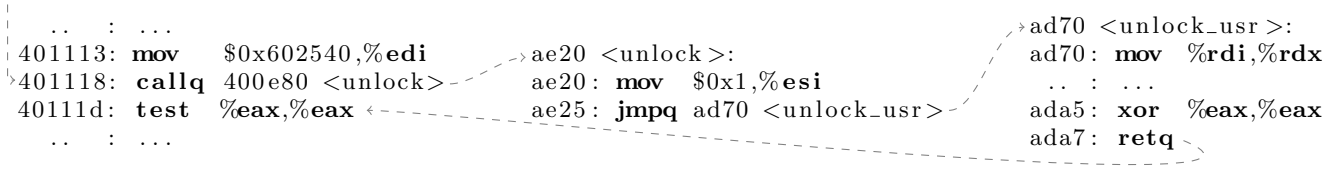


Figure 1: An example of an execution not triggering an after-function notification

time, use the program as usual. This possibility is also very important when analysing libraries as it allows the user to analyse a library when used by a program being analysed and simultaneously allow other programs to use the same library as usual. This is not possible without maintaining two separate versions of the library and coping with problems with paths to these versions when the code of the library is rewritten before its execution (usage) in case of static binary instrumentation.

However, regardless of which of the two types of the binary instrumentation approaches is used, there are some issues that need to be dealt with when analysing multi-threaded programs at the binary level. These issues are discussed below.

2.2 Monitoring Execution of Functions

The first problem to deal with is how to properly monitor the invocation and termination of functions. This is very important when analysing multi-threaded C/C++ programs as thread management, synchronisation of threads, and other thread-related actions are typically implemented by calling specific library functions. For instance, if an analyser needs to know that the monitored program acquired a lock, the best time to issue a notification about this event is after the function performing the lock acquisition is finished. However, since `call` instructions are not *fall-through instructions* (i.e., there is no guarantee that the instruction by which the program will continue after the invoked function finishes will be the instruction right after the given `call` instruction), one cannot place the code notifying an analyser about the event after the `call` instruction itself.

One way to solve the above problem could be to wrap the function to be monitored in another function and call everywhere in the program the wrapper function instead of the original one. The wrapper function could then internally call the original function, but also execute some code before and after it is called. This solution, however, suffers from two problems. First, the framework used for the binary instrumentation would have to support function wrapping (replacement), and second, calling the original function from the wrapper function might be quite time consuming and may lead to a significant slowdown of the whole analysis. Moreover, to wrap a function, we need to have a wrapper function with the same signature as the original function, containing the required monitoring code, so it might also be problematic to use this approach when we do not know in advance which functions we will be wrapping.

Another way to solve the problem could be to insert the monitoring code before and after the code of the monitored function itself, e.g., by inserting the code before the first instruction of the function and before each return instruction in the code of the function. So, instead of issuing the notification after the function returns, the analyser would be notified right before the function returns, which would be practically the same from the point of view of the analyser. An additional advantage of this approach would be that it would also decrease the instrumentation overhead as instead of analysing all `call` instructions (to see whether they could invoke the function to be monitored) and instrumenting many of them, the code of the functions to be monitored would only be instrumented. Nevertheless, this approach has one critical pitfall—namely, at the binary level, it is possible to return from a function even from code not belonging to that function!

A concrete example where instrumenting all return instructions of a function will not trigger the code that should be executed after the function returns can be seen in Figure 1. The figure shows three pieces of code. On the left there is a part of the binary code generated from a simple C++ program which uses the pthread library to guard a critical section through the `pthread_mutex_lock` and `pthread_mutex_unlock` functions. The other two code snippets are parts of the code of the pthread's `__pthread_mutex_unlock` and `__pthread_mutex_unlock_usercnt` functions. Once the execution of the program reaches the `call` instruction at address `401118`, the program calls `__pthread_mutex_unlock`[1] (the function `pthread_mutex_unlock` is in fact just an alias of `__pthread_mutex_unlock`). The execution then continues in the `__pthread_mutex_unlock` function which adds the second argument of the `__pthread_mutex_unlock_usercnt` function and then jumps to it. The program starts executing the `__pthread_mutex_unlock_usercnt` function, and after a while, it returns, but not to the `__pthread_mutex_unlock` function (because this function did not call it, it jumped to it), but to the function that called `__pthread_mutex_unlock`. Now, it should be clear where the problem is: If we instrument the `__pthread_mutex_unlock` function, there will never be a notification that a thread released a lock because there will be no code inserted before the return instruction which is executed to return from the call to the `__pthread_mutex_unlock` function. Moreover, when we try to insert the monitoring code before the return instruction that is really executed in the given example, we have to be still careful. The reason is that we do not know whether some optimisation did not make more functions jump to the given code. If so, it could, for instance, happen that the `pthread_mutex_lock` function jumps to this part of code too, which would lead to issuing a notification that a thread released a lock while it instead acquired it.

Nevertheless, there is a solution to the above described problem. Namely, we can use the fact that no library function can jump outside of the code of the library itself. This is because when the library is compiled, the compiler can insert jumps just to the parts of the code it knows, and it knows only the code of the library itself. So, if we insert some monitoring code before every return instruction in the library, we must be able to detect that the program is returning from a call to some of the library's functions. The only issue that is then left is to find out from which function

[1]For the sake of simplicity, we ignore here the fact that the function is not called directly, but through the jump table stored in the `.plt` section of the program's binary.

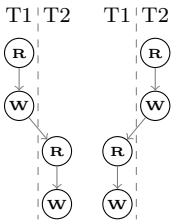
Thread 1

```
5f0da0 <AtomicIncrement>:
5f0da0: mov    $0x1,%eax
5f0da5: xadd   %eax,(%rdi)
5f0da9: add    $0x1,%eax
5f0dac: retq
```

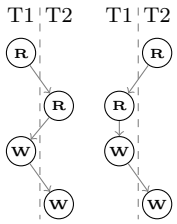
Thread 2

```
5f0da0 <AtomicIncrement>:
5f0da0: mov    $0x1,%eax
5f0da5: xadd   %eax,(%rdi)
5f0da9: add    $0x1,%eax
5f0dac: retq
```

(a) Parts of assembly code executed by the first and second threads, respectivelly. The code uses the exchange and add (`xadd`) instruction to increment a value at a specific memory address atomically.



(b) The only possible interleavings



(c) Some impossible interleavings

Figure 2: An example illustrating problems concerning atomic instructions

the program is actually returning. Probably the most efficient way in doing so is to partially monitor the call stack, i.e., when a function whose execution should be monitored is called, some monitoring code inserted before the first instruction of the function can be triggered, and in this code, we can save the current state of the thread's call stack (the value of the stack pointer is quite sufficient). Then, when a return instruction is executed in the library, we can check if the current stack pointer matches the previous stored one, and if yes, issue a notification that a certain library function has been executed.

2.3 Retrieving Required Information

A further problem to be solved is how to provide analysers with sufficient information about an instruction or function whose execution has just finished. This is because a lot of information is lost when an instruction or function is finished. For example, we often do not know which memory an instruction has accessed after the instruction is executed because the memory address might have been computed from the values of some registers, and those values might have been changed when the instruction was executed. Similarly, when a function is executing, we can access its parameters easily, but when its execution finishes, the arguments might not be easily obtainable anymore.

To provide an analyser all the information it needs, it is thus often necessary to preserve some of the information obtained before executing some instructions and to reuse this information in the monitoring code executed later. In case of multi-threaded programs, such information must be tracked for each thread separately, and since it might be accessed quite frequently, the efficiency of storing it is of a high importance. As the best way to deal with this problem, we see a utilisation of some kind of thread local storage, which is lock-free, i.e., it does not require any synchronisation between the threads. Fortunately, some binary instrumentation frameworks, such as PIN [10], provide a support of thread local storage.

In fact, according to our experience, it is often useful to store also information which can be computed even after executing some instructions. This is, in particular, the case of source-code information about the code being executed such as the names of variables accessed. Such information may be available, e.g., through the debugging information present in the program's binary, but accessing this information is often slow. If we get this information in some monitoring code and know that some other monitoring code executed after a while will need this information too, it is better to store the extracted information and reuse it later than to extract it again.

2.4 Atomic Instructions

Another problem which may arise when analysing programs at the binary level is the need to properly handle atomic instructions that access the memory more than once. Indeed, when a single instruction accesses the memory multiple times, the monitoring code should notify the analyser about that, which is typically done by generating the appropriate number of memory access events. If the instruction is not atomic, this is perfectly fine, but when the instruction is atomic, some analysers need to be informed about the atomicity of the appropriate sequence of memory accesses, or else they might produce false alarms.

In particular, some of the detectors which may have troubles with atomic instructions are data race detectors, such as Eraser [13] or AtomRace [7], and atomicity violation detectors, such as AVIO [9]. Both these kinds of detectors analyse possible interleavings of accesses to particular memory addresses and report an error if there are two unsynchronised memory accesses to the same memory address and at least one of the accesses is a write access, or there is an interleaving of the memory accesses which is unserialisable. Clearly, if such detectors are not informed that some sequence of memory accesses is guaranteed to execute atomically, they can produce false alarms. A concrete illustration of such a scenario can be seen in Figure 2 which shows a situation when two threads are executing the code of the `AtomicIncrement` function. This function uses the exchange and add (`xadd`) instruction to atomically increment a value at a given memory address. The `xadd` instruction first reads a value at a given memory address, then adds some value to it, and then stores the modified value back at the same memory address. If the monitoring code notifies the analysers that the program read a value from some memory address and then wrote to the same memory address without the information that these two accesses happened atomically, the analyser will assume that all the interleavings shown in Figures 2b and 2c are possible in the program because the threads are not synchronised in any way. However, in fact, the interleavings in Figure 2b are the only ones that can happen in the program.

One possible solution to this problem is to extend the access notifications with additional information saying that an access that has just happened occurred atomically wrt. some previous access. Pairing such accesses in the analyser may, however, be problematic (e.g., leading to backtracking in the analysis). In our opinion, a better way is to introduce a new type of notification which tells the analyser that one instruction performed multiple accesses at once, and let it decide if it wants to react to this situation and how (the analysers must of course be ready to receive such notifications).

Note that in some higher programming languages like Java, there is no need to solve this kind of problems when performing analysis at the binary, or more precisely byte-code, level as there are no byte-code instructions which access the memory more than once [8]. Atomic updates are implemented here as a block of byte-code instructions placed between the `monitorenter` and `monitorexit` instructions which lock the memory address securing that no other thread will access it until the update is completed. If all accesses to the memory address are guarded by the same lock, the analysers will see that there are no possible interleavings leading to an error on this memory address and will not produce false alarms. The same solution can be used in C/C++ programs, but using atomic instructions can be much more efficient, and hence it is often used.

2.5 Conditional Instructions and Loops

Beside the atomic instructions there are a few other kinds of instructions which must be carefully handled in order not to confuse various existing analysers. This is, in particular, the case of the conditional instructions and the repeat instructions. While the conditional instructions might not be executed at all even when the control reaches them, the repeat instructions, on the contrary, may be executed more than once as though they were placed in a loop. For instance, the `rep`-prefixed instructions, designed for manipulating continuous sequences of memory locations (e.g., within string operations), are both conditional and repeat instructions since they may be executed a fixed number of times, until some condition is met, or sometimes not executed at all (if the loop they involve should be executed zero times).

Since many of these kinds of instructions access memory, we need to be sure that the access notifications are sent correctly, i.e., that the analyser is notified only when the instruction was really executed or every time the instruction was executed in a loop. When using dynamic binary instrumentation frameworks, where the instructions are executed by some kind of a virtual machine, the virtual machine can usually handle these things for us. On the other hand, in case of the static binary instrumentation, where we might not know if the instruction will be executed or how many times it will be executed, the situation can become unsolvable without some approximation.

2.6 Abstraction of Synchronisation Primitives

Since thread management and synchronisation in C/C++ programs is usually done by calling suitable library functions as we have already mentioned above, and since there exist many different libraries which can be used for this purpose, a further question is how to support analysis of programs using any of these libraries with a minimum additional effort (at least when no highly non-standard synchronisation means are used).

In order to allow for an easy support of multiple libraries, the dynamic analysers themselves should clearly be separated from low-level details of using the libraries. For example, an analyser should not know that a lock is represented by a `pthread_mutex_t` structure or a Windows `HANDLE`, it should just be able to say whether two locks are the same or not. More generally, according to our experience, in order to satisfy the needs of common analysers, one needs to allow them (1) to suitably identify program threads in order to be able to distinguish which thread behaves in which way, (2) to recognise which functions are used for various standard synchronisation operations, and (3) to suitably identify the synchronisation resources used (such as locks or conditions) in order to be able to say which of them are used when.

Since one can hardly find a fully automatic solution of the above needs, we find as appropriate to provide users with a support allowing them to solve these issues in an easy way manually. In particular, for a given library, the users should be able to easily specify:

- Which functions in the library are performing common thread-related actions such as acquiring a lock, waiting on a condition, etc.
- Which arguments of these functions represent the synchronisation resources they work with.
- How to transform the concrete representations of synchronisation resources and threads to their abstract identifications.

In Section 4, we will describe in more detail the concrete implementation of this approach as used in our prototype tool.

3. NOISE INJECTION

We now proceed to noise injection as a means of increasing chances to spot errors when testing or dynamically analysing multi-threaded C/C++ programs at the binary level. We start by a brief summary of some of the ideas behind the ConTest tool [3] for noise-based testing and dynamic analysis of Java programs that can be applied in our setting too. Moreover, we also argue that the approach of [3] can be improved by a careful fine-grained combination of different noise injection techniques, which we then experimentally validate in Section 5.

Note that apart from increasing chances to see an error or to produce a warning from a dynamic analyser, noise injection can also be used in conjunction with the mechanism of C/C++ assertions. The assertions explicitly guard the execution of a program against situations which, from the perspective of the programmer, should never happen, but if they happen, the programmer should be notified about them. When dealing with the naturally non-deterministic execution of multi-threaded programs, it is hard to say, even for a skilled programmer, whether some situation is really excluded from happening, and so the number of assertion checks tends to be higher in multi-threaded C/C++ programs. The noise injection can then be conveniently used to increase the chances to see the executions which violate the assertions present in the code alerting the programmer about situations he/she did not expect to occur.

3.1 Noise Injection Basics

Noise injection techniques [3] aim at increasing the number of different interleavings witnessed in testing runs by disturbing the scheduling of threads of a program. This is achieved by inserting certain noise generating code at some locations of the program whose goal is to force the program to switch threads at times when it would normally seldom do it. This way, rare executions may be forced to appear, possibly leading to an occurrence of an error (or to a behaviour that can be claimed suspicious by a dynamic analyser).

When using noise injection, the user typically has to make several important decisions which include what *type of noise* should be used, with what *frequency*, and with what *strength*.

As for the types of noise, there have appeared many of them, but we focus here on two generic ones, namely, *yield* and *sleep*, which were found to often work well in the case of Java programs. The yield noise forces a thread to give up the CPU, which effectively forces the program to switch threads and to continue the execution elsewhere. The sleep noise puts a thread to sleep for some time which also forces the program to switch threads, and moreover, it prevents the program from switching back to the sleeping thread for a while. The noise frequency tells the noise injection code how often the noise should occur, while the strength specifies how strong the noise should be. The concrete meaning of the strength depends on the concrete type of noise used: for the yield noise, it says how many times the yield should be called; for the sleep noise, how long a thread should sleep. The specification of the strength can be interpreted either as a constant strength to be used, or as the maximum of randomly generated strength values.

Another important decision is where to put the noise injection code. Of course, one can insert noise generation before any instruction of the program under test, but placing the noise generation at too many places may significantly slow down the execution of the program, especially when the sleep noise is used. In such a case, the overall benefit of using noise could be negligible since significantly less testing runs could be performed. Moreover, as we discuss below, sometimes, the effects of noise used in unsuitable combinations at unsuitable places may cancel out. Since concurrency errors are mostly caused by missing or wrong synchronisation among threads, leading to inconsistent memory configurations or deadlocks, the most meaningful places where to put the noise injection code are typically various thread management and synchronisation functions and memory accesses (as witnessed by various experiments performed with the ConTest tool [3] on Java programs).

3.2 Fine-Grained Combinations of Noise

In our opinion, the ConTest tool has one drawback—namely, it allows the user to specify the noise injection settings at a global level only, meaning that the same type of noise with the same parameters will be used everywhere in the program (an exception is the random setting of ConTest when a random type of noise with random parameters is used every time some noise is to be generated). However, in our opinion, supported by the later presented experimental data, it is sometimes beneficial to use different noise injection settings for different locations in the program (e.g., for the read accesses, write accesses, each of the thread synchronisation functions, etc.) and to do it in a systematic rather than random way.

To illustrate our idea, take, e.g., the case of data races. Considering that a data race arises when there are two unsynchronised accesses to the same memory address and at least one of the accesses is a write access, it might appear to be better to use the sleep noise than the yield noise. This is because when we encounter some access to the memory address of interest, the best we can do is to search the other threads for the second (conflicting) access. This means that as many of the remaining threads should go through as many memory accesses as possible. Forcing the program to switch threads several times using the yield noise of a commonly used strength will help us to search only a small part of the executions of the other threads. The sleep noise will block the execution of the thread performing the first access giving us considerably more time to detect the second (conflicting) access in one of the remaining threads.[2] However, a problem is that if we use the same sleep noise for all accesses, we will block not only the thread performing the first access, but also many of the remaining threads, which will unnecessarily lower the number of memory accesses they will perform. Hence, what we want to achieve is to lower the amount of noise injected into the remaining threads which we search for the second conflicting access.

The above situation is where the more fine-grained noise injection configuration can help. In particular, we can use different noise settings for different, possibly conflicting types of accesses to hold the threads performing one of the types of accesses more than the threads performing the other type of accesses. There are two ways to do that. One possibility is to use the sleep noise only, but with a bigger strength for one of the access types and considerably lower for the other. However, an even better way is to force the threads performing the second type of accesses to give up the CPU (using the yield noise) as this will help more threads to perform more memory accesses. As we will show in the experiments section, this approach really gives us better results than using a single global configuration, and, in addition, it often slows the execution of the program much less.

A question left open in the previous paragraph is which accesses should be hold more and which less. In our opinion, this mainly depends on which of the conflicting accesses happens more rarely. Clearly, if one of the conflicting accesses happens more rarely (e.g., a write access if there are few possibilities to write and many possibilities to read), it is better to hold the thread performing the rare access using a sleep noise and search for the more common accesses than doing it conversely.

4. PROTOTYPE IMPLEMENTATION

To validate the solutions proposed in the previous sections, we have implemented a prototype tool which can monitor the execution of a multi-threaded C/C++ program, insert noise into it, and provide analysers that can be written on top of it with various important pieces of information that are typically needed when detecting errors in concurrency. In this section, we will discuss how the above discussed general ideas have been concretised in the implementation.

We have based our implementation on top of the PIN binary instrumentation framework [10]. This framework is primarily developed for use with the Intel binaries. However, if the binary code does not contain any special AMD-only instructions, PIN works fine even for AMD binaries. Also, as the PIN framework supports both Linux and Windows binaries, our framework can be used to analyse programs developed for any of these two operating systems.

To allow the tool to be easily used for developing dynamic analysers capable of running over various libraries for thread management and synchronisation, we followed the propositions made in Section 2.6 and implemented a system which allows the user to easily abstract the information needed by typical analysers from the concrete form used in

[2] A similar effect could be reached by using a much stronger yield noise, which would, however, involve actively waiting threads that would in turn unnecessarily slow down the entire system.

a given library into a common format available for the analysers. In particular, for each important type of synchronisation functions, e.g., functions for acquiring and releasing locks or for signaling conditions and waiting on them, the user may specify the names of the functions implementing these operations. For an abstract identification of the synchronisation resources used (e.g., locks or conditions), we introduced special `Mapper` objects. When the user defines the names of synchronisation functions, he also specifies the indices of their arguments holding the synchronisations resources used by these functions as well as the the `Mapper` object which should be used to translate these resources to their abstract names. The translation is then done through the `map()` method which takes a pointer to an address where the appropriate argument of the encountered synchronisation function is stored and returns a number abstractly identifying the appropriate synchronisation resource. For example, for the case of locks and conditions from the pthread library used to synchronise threads, the mapper objects can use the fact that the locks and conditions are objects of the `pthread_mutex_t` and `pthread_cond_t` structures existing in the same logical memory space shared by the threads. Since objects existing in the same memory space are uniquely identified by their addresses, one can devise a `Mapper` object which simply uses the addresses of the objects as their identifiers in this case (hashed to 32 bits as is usual also for various other identifiers in PIN). Finally, as for an abstract identification of threads, we use the fact that the PIN framework already provides some thread abstraction, and so we reuse it for a unique thread identification.

As the synchronisation functions to be monitored are assumed to be specified by the users, and hence we do not know them and cannot prepare wrappers in advance, we cannot use the function wrapping approach for monitoring function executions (not to mention that calling the original function from a wrapper function is often really slow). Therefore, we use the approach for monitoring function executions proposed at the end of Section 2.2. Namely, we insert a monitoring code before the first instruction of every synchronisation function specified by the user and also before all the return instructions in the libraries containing at least one of these functions. When some synchronisation function is about to be executed, the monitoring code stores the current value of the stack pointer together with a pointer to the notification function which should be called after the monitored function is executed to a separate shadow stack. Once a return instruction is to be executed, the monitoring code compares the current value of the stack pointer with the one stored at the top of the shadow stack, and if there is a match, it will notify the analyser that a synchronisation function was executed by calling the notification function stored at the top of the shadow stack.

We have implemented the support for notification about several atomic accesses to memory from an atomic instruction as discussed in Section 2.4 (so far for accesses to the same memory address only, which is sufficient for detecting data races on which we currently concentrated). To handle conditional instructions discussed in Section 2.5, we used the PIN's `INS_InsertPredicatedCall()` function to put the monitoring code around these instructions. Inserting the code through this function ensures us that PIN will check if the instruction will really be executed and invoke the monitoring code in this case only. However, unfortunately, using this function to insert the monitoring code around the `rep`-prefixed instructions led to a very strange behaviour in which the monitoring code was sometimes not invoked even when the instruction was executed. To fix this problem, we had to use the `INS_InsertCall()` function as code inserted using this function is always called, and we then check ourselves if the instruction will or will not be executed. This implementation is a little less efficient as letting the PIN do the checks is quicker, but it behaves correctly, and considering that the amount of `rep`-prefixed instructions is not so large, it is a quite negligible slowdown.

Although the framework we have developed is to a large degree generic as should be clear from the above, currently, we have instantiated the generic parts of the framework for use with the pthreads library only. A support for other multi-threading environments is a part of our future work.

5. EXPERIMENTS

In this section, we present results of our experiments with a prototype implementation of the above discussed ideas. We use these results to justify the proposed solutions and also as a basis for a discussion of the possible influence of various noise injection settings on dynamic analysis of multi-threaded C/C++ programs.

5.1 Experimental Setup

For our experiments, we used 116 multi-threaded programs implementing a simple ticket algorithm on top of the pthread library. These programs were created by students of an advanced operating systems course. Note that most of them were rated full points as the test script and a brief code review did not find any errors. We were, however, able to find various errors in many of these programs using dynamic analysis in conjunction with noise injection or even just the noise injection alone.

Algorithm 1: Ticket algorithm

```
1 foreach thread do
2     while (ticket = getticket()) < M do
3         sleep(random);
4         await(ticket);
5         doWork();
6         advance();
7         sleep(random);
```

Algorithm 1 describes the general idea behind the ticket algorithm that each of the programs implements. The goal is to synchronise all threads of a program in doing some mutually exclusive work (modelled by calling `doWork()`). When a thread wants to do the work, it is assigned a ticket number and waits for its turn. The `getticket()` function assigns a thread the first free ticket (i.e., a ticket with the lowest ticket number not assigned to any other thread yet). This is done using a shared variable `next_ticket` holding the number of the next free ticket. All accesses to this variable are done in a critical section guarded by the `tmutex` lock. The accesses to the part where the work is done are then guarded by a monitor entered by calling the `await()` function and left by calling the `advance()` function. These two functions work with a shared variable `curr_ticket` which determines the ticket number needed to enter the monitor. The `await()` function reads the `curr_ticket` variable and forces the thread to wait if it is not its turn (i.e., if it does not have a ticket with the `curr_ticket` number). The `ad-`

vance() function then increments the curr_ticket variable allowing another thread to enter the part guarded by the monitor. Accesses to curr_ticket in each of the functions are done in critical sections guarded by the mmutex lock. If the work is done M times, the threads finish their execution.

5.2 Detecting Data Races

To look for data races in the considered programs, we used the simple detector called AtomRace [7]. AtomRace tracks which memory addresses are being accessed by the particular threads by monitoring the before and after memory access notifications. If it discovers that two threads can concurrently access the same memory address (at least one of them for writing), which is detected by the appropriate pairs of before and after memory accesses being overlapped, it informs about a data race. As it is normally not very probable to see two concurrent memory accesses, AtomRace uses noise injection to disturb the usual scheduling of threads in order to witness executions in which such memory accesses happen (if that is allowed by the program under test). Clearly, when AtomRace announces a data race, it is a real data race, not a false alarm (of course, provided that it takes into account the possible appearance of atomic instructions).

Using AtomRace implemented on top of our infrastructure, we were able to find data races in 23 of the 116 considered programs, all of them having various kinds of negative impacts on the expected behaviour of the programs. We find this quite satisfactory taking into account that three quarters of these programs were rated full points because they passed all the standard tests.

To further analyse how the noise injection settings influence the overall success of the detector to find data races, we selected 13 of the 23 programs in which we were able to find bugs and performed a large number of tests on them. The remaining programs were not included in the tests as they contained deadlocks in addition to data races, which made them difficult to compare to the rest of the programs. The results are shown in Table 1, each column representing one of the tested programs and each row one of the configurations of the noise injection in the following format: First, a base configuration of noise generation used at memory accesses and synchronisation functions is given, consisting of the type of noise, its frequency, and its strength. The rs prefix put before the type of noise indicates that the strength is not implemented as constant, but as random with the given value being the maximum possible strength. The values of the frequency say how probable it is that some noise will be generated every time the given location is reached on the scale from 0 to 1000, i.e., 500 means 50 %, 100 means 10 % etc. The values of the strength say how many times a yield should be called at the given location of the given thread or how many milliseconds the thread should wait in case of the sleep noise. Then, if applicable, differences from the base configuration in the strength and possibly also type of noise are given behind a slash separately for the read and write accesses. The values in the body of the table then express the percentage of runs (out of 500) in which the data race detector found a data race, i.e., the percentage of executions which actually led to an error. The first noise configuration in the table corresponds to runs where no sleep nor yield noise is generated, but the noise generation code is inserted, together with the code notifying the AtomRace detector about the execution. However, even such instrumentation is already generating some small noise which can help manifestation of errors as we will see in Section 5.3.

The selected programs contain various kinds of errors that all lead to data races in the end. In two of the programs (t01 and t02), the data race is on the next_ticket variable. In the first program (t01), the variable is updated in a critical section, but then read outside of it. Since the getticket() function performing these accesses is frequently called from many of the threads, the data race manifests quite often[3]. In the second program (t02), the accesses to the variable are not guarded at all, so the data race manifests even more often. The next program (t03) contains a data race on a shared variable used to assign IDs to each of the threads. This variable is updated and read without any synchronisation, however, all of these accesses happen when the threads are started one immediately after another, so the data race may only occur during this short time. Program t04 uses a shared structure to store the thread IDs and their current tickets. Accesses to all the members of this structure are not synchronised, leading to data races on each of them, multiplying the probability that a data race appears. Program t05 has a rarely occurring data race on individual items of a shared array where each item may be accessed by the main thread, and one of the other threads simultaneously just before the main thread starts to wait for the second thread to end (join). Programs t06, t07, and t08 contain data races on a timespec structure, shared among all threads, used to randomly generate the number of milliseconds a thread should sleep before and after entering the monitor. Some of these programs access the structure more often than the others, so the frequency of encountering a data race vary between them. The next two programs (t09 and t10) read the curr_ticket variable outside a critical section at one place. All other accesses are, however, performed in the critical section, so it is not very likely that a data race would occur. Program t11 uses the same lock for guarding the critical section in the getticket() function as well as the critical sections in the monitor functions await() and advance() leading to an extremely rare situation where two threads enter the critical section in the getticket() function and access the next_ticket variable (because the code does not check if the tmutex lock was acquired successfully and just continues). A similar situation happens in program t12, which initialises the tmutex and mmutex locks in each of the created threads, which resets the locks' ownership information, status, and other fields. Changing the ownership information often leads to assertion errors as we will see in Section 5.3. On the other hand, resetting the lock status leads to data races since it allows more threads to enter the critical sections guarded by these locks. However, these data races can manifest only if the noise simultaneously prevents all assertion errors which may otherwise show up before the data races. The last program (t13) contains a data race on a shared variable used to store the return codes of pthread library's functions. Since this variable is accessed at so many places in the program, the data race occurs very often here.

Evaluation of the Results. As can be seen from Table 1, the sleep noise is clearly superior in helping AtomRace in finding data races when the same strength is used. However,

[3]This is the case when the program is run with the instrumentation needed for AtomRace and with the instrumentation for noise generation which is just not generating any sleep nor yield noise. The same holds for the discussion of the other case studies too.

Table 1: Success ratio of the AtomRace detector for various configurations of the noise injection (the values represent the percentage of runs, out of 500, in which a data race was found)

Noise configuration \ Program	t01	t02	t03	t04	t05	t06	t07	t08	t09	t10	t11	t12	t13
instrumented, no sleep or yield noise	2.4	11.8	0.2	1.2	0.0	1.0	1.6	2.2	0.4	0.0	0.0	0.0	32.2
sleep 500 10	69.2	46.6	100.0	100.0	1.2	53.6	69.4	98.6	0.6	0.2	0.8	0.2	100.0
yield 500 10	3.8	35.4	1.4	10.8	0.0	1.4	5.4	11.6	0.6	1.8	0.0	0.2	84.4
rs-sleep 500 10	**96.4**	**87.8**	97.0	86.2	0.6	31.0	79.0	**99.2**	9.2	10.0	**2.4**	0.2	100.0
rs-yield 500 10	6.0	17.0	0.2	0.6	0.0	0.6	6.2	9.0	0.4	1.6	0.0	0.4	71.2
sleep 100 10	64.0	69.2	80.2	56.0	5.4	40.2	70.4	81.0	31.2	31.4	0.6	**42.4**	98.4
yield 100 10	0.8	17.0	0.4	3.0	0.0	0.0	4.2	6.0	0.6	0.8	0.0	0.0	42.8
rs-sleep 100 10	21.4	55.8	23.0	11.8	0.0	18.2	60.4	71.8	**34.4**	**37.0**	0.8	27.6	92.8
rs-yield 100 10	1.8	9.0	0.6	1.0	0.0	0.4	3.2	3.4	0.6	2.0	0.0	0.0	34.8
sleep 500 20	34.6	48.2	100.0	100.0	0.6	31.8	79.0	97.8	0.0	0.4	0.2	0.6	100.0
yield 500 20	14.2	56.4	4.4	9.4	0.0	2.2	8.6	17.0	1.0	0.8	0.2	0.0	94.0
sleep 500 5	24.2	68.2	100.0	69.2	6.4	26.8	79.8	90.2	1.8	2.2	1.4	0.0	100.0
yield 500 5	2.8	22.2	3.0	13.4	0.0	3.6	5.8	5.2	0.6	1.2	0.0	0.0	63.0
sleep 100 20	59.2	30.4	78.6	6.0	5.6	61.0	67.2	86.4	32.2	33.6	0.8	**42.6**	98.8
yield 100 20	1.2	19.0	0.6	3.0	0.0	1.8	6.4	6.8	1.4	0.8	0.0	0.2	54.2
sleep 100 5	52.4	73.6	78.4	74.4	14.2	18.4	61.0	74.0	29.8	30.6	0.2	38.0	98.2
yield 100 5	1.4	13.0	0.2	1.4	0.0	0.6	3.4	4.2	1.8	1.6	0.0	0.2	38.8
sleep 500 10 / read 20 / write 5	**64.8**	**89.4**	99.0	80.8	0.0	17.0	28.6	91.2	0.4	2.2	0.4	0.0	100.0
sleep 500 10 / read 5 / write 20	33.4	57.2	100.0	91.6	43.0	**92.6**	**96.2**	**99.8**	1.2	0.6	1.4	0.0	100.0
yield 500 10 / read 20 / write 5	3.8	31.4	3.8	9.0	0.0	1.4	3.6	9.2	1.8	1.8	0.4	0.0	86.0
yield 500 10 / read 5 / write 20	5.0	59.6	2.0	8.2	0.0	1.6	9.2	19.8	1.8	0.2	0.0	0.0	88.0
sleep 100 10 / read 20 / write 5	**73.4**	44.4	67.4	2.2	0.0	79.0	51.2	73.0	**34.4**	**37.0**	0.8	0.0	99.2
sleep 100 10 / read 5 / write 20	31.4	27.6	70.8	69.2	4.8	**89.2**	79.6	68.8	30.0	29.6	1.2	0.0	99.2
yield 100 10 / read 20 / write 5	0.8	13.2	0.2	2.2	0.0	0.4	4.6	4.2	1.0	0.8	0.0	0.0	49.0
yield 100 10 / read 5 / write 20	1.2	23.0	0.4	2.6	0.0	1.4	6.6	5.8	0.6	0.8	0.0	0.0	48.8
sleep 500 10 / read sleep / write yield	51.2	61.2	59.2	100.0	0.0	2.4	0.8	83.8	0.2	2.6	0.2	0.0	100.0
sleep 500 10 / read yield / write sleep	18.6	38.4	99.4	100.0	**50.6**	80.8	**95.6**	**97.0**	7.4	4.6	1.2	0.0	100.0
yield 500 10 / read sleep / write yield	32.6	64.6	63.8	100.0	0.0	4.6	0.0	68.6	0.2	3.8	0.2	0.0	100.0
yield 500 10 / read yield / write sleep	10.0	52.2	98.0	100.0	**51.0**	**95.0**	**99.6**	**98.8**	6.0	4.2	**2.0**	0.0	100.0
sleep 100 10 / read sleep / write yield	34.2	**81.0**	44.0	7.4	**62.4**	0.0	2.2	55.0	28.8	**37.6**	0.8	0.0	87.4
sleep 100 10 / read yield / write sleep	9.4	35.6	52.4	96.8	9.6	37.6	78.6	62.0	1.0	2.4	0.0	0.0	88.2
yield 100 10 / read sleep / write yield	25.8	46.4	51.2	6.2	**64.4**	0.2	4.4	69.6	**43.0**	**43.4**	0.2	0.0	85.6
yield 100 10 / read yield / write sleep	16.6	33.6	43.6	94.4	7.2	35.6	80.2	61.2	1.4	1.6	0.2	0.0	90.6

using the sleep noise too much can sometimes have quite the contrary effect—hiding the data races instead of helping to find them. Take, for example, programs `t01` and `t02`. They contain a similar error of not guarding the accesses to the `next_ticket` variable, but while `t01` is not guarding only some of the read accesses, `t02` is not guarding any of the accesses, so the possibility of encountering a data race in an execution should be higher here than in case of `t01`. However, when using the sleep noise with frequency 500 (50 %) and strength 10, data races are detected in only 47 % of `t02` runs, while in case of `t01` it was nearly 70 %. After decreasing the frequency to 100 (10 %) or strength to 5, the success ratio of data race detection increases to nearly 70 %. The problem here is that if we put all the threads to sleep for about the same time (which is the more probable the higher the frequency is), the scheduling of threads will remain the same as without the noise, not introducing the uncommon executions we wanted to witness. In many cases, lowering the frequency helps, likewise using a random strength instead of the fixed one. Sometimes even using a smaller fixed strength might help as we do not block the threads for too much time which increases the chances that there will be some threads which we might search when we start sleeping (there are often none if we use a too strong strength—e.g., when using strength 20 in `t02`, we detect data races in only 30 % runs even if we use frequency 100). Similar problems happen when using noise injection in Java programs [2].

The results also support our opinion from Section 3.2 that using different noise injection settings for different locations can give better results. This approach can, e.g., help in solving the problem discussed in the previous paragraph. Indeed, in case of `t02`, instead of using a random strength, it is better to use fixed strengths 20 for reads and 5 for writes, which leads to detection of a data race in nearly 90 % of runs. Moreover, the approach also helps in many other cases, e.g., in case of `t06`, `t07`, and `t08` where using sleep noise with strength 5 for reads and strength 20 for writes gives similar or better results than using strength 10 for all accesses. In some cases where lowering the frequency decreases the probability of detecting a data race, like in `t04`, we can use yields for reads and sleeps for writes with the frequency being 100 only and still keep the success ratio as for frequency 500, which speeds up the execution of the program considerably. Sometimes combining various strengths or different types of noise is the only way to detect some data races in a fair number of testing runs as, e.g., in case of `t05`.

Further, we also verified our assumptions from Section 3.2 that blocking the more rare types of accesses should give us better results. For example, `t04` contains data races on members of a structure which are written to several times, but read from frequently. The results are very good when using the sleep noise with a high frequency and strength, but they are considerably worse when the amount of noise is lower. However, using frequency 100 with the sleep noise for the rare writes and yield noise for the common reads gives us nearly the same results as using the strong noise. On the other hand, when we use the opposite combination of noise for the different accesses, the results are very poor. In cases where the read and write accesses to the variable on which a data race is found are equally common, like in case of `t06` and `t07`, it is still better to use the sleep noise for writes and yield noise for reads. This is due to when considering all accesses to all variables in the program, the read accesses are typically more frequent, and so we will not block the remaining threads too much. On the other hand, consider the `t09` and `t10` programs. They contain

a data race on a shared variable which is accessed mostly in a critical section except several reads. So all the writes are guarded and there are only several reads that may access the shared variable when it is written to. If we block the threads performing the write, it is highly improbable that we will find the rare unguarded read in some of the remaining threads, but blocking the threads performing the rare reads allows us to find the data race in a fair number of runs.

The statement that blocking the rarer accesses gives us better results seems not to hold in case of `t05` where with frequency 500, using the sleep noise for reads and yield noise for writes is clearly better, but with frequency 100, it is just the opposite. The problem here is that the write accesses are so rare that if we use a small frequency, we will not block the execution of the thread performing the write access to find the conflicting read in the other threads in the meantime. So when using a high frequency, the probability to inject the noise before some write is relatively high, and it is better to use the sleep noise for writes. On the other hand, the read accesses are performed in a loop, and so there is a large number of them. Causing too much noise before the reads does not help here at all as it often hides the data races, but if we inject the noise before the reads with a low frequency, we will still have a good chance to encounter the rare writes in the other threads.

5.3 Detecting Assertion Errors

We have also tested whether the noise injection can help us in detecting wrong usages of the pthread library such as cases when a thread releases a lock which it does not own and the like. Such scenarios are detected directly by assertions built into the pthread library. We used the same set of 116 programs as before and checked their output for assertion errors originating from the pthread library. Among the 116 programs, we found 3 that break the built-in assertions.

Again, we studied how the noise injection settings influence the overall success of detecting the wrong usages of the pthread library. The results are shown in Table 2, each column representing one of the tested programs and each row one of the configurations of the noise injection in the same format as in Table 1. The values in the body of the table then express the percentage of runs (out of 500) which ended with an assertion error.

The first two programs (`t02` and `t12`) are both initialising the `tmutex` lock in each of the created threads, resetting the lock's ownership information and status when a new thread is started. This allows more than one thread to acquire the lock as one thread may acquire the lock, and then another thread may start, reset the status of the lock to *not acquired* and afterwards acquire the lock itself, hence becoming the owner of the lock. If the program now switches to the first thread, and this thread will release the lock, the pthread library will raise an assertion error saying that some thread is trying to release a lock which it does not own. This also leads to data races as more than one thread may access the critical section guarding the `next_ticket` variable and access it. The third program (`t14`) releases the `mmutex` lock twice in the `advance()` function, instead of acquiring it and then releasing it. Due to this, while some thread is inside the critical section of `advance()`, another thread can acquire the not locked `mmutex`, and then the former thread may release it despite it never acquired it, which causes the same assertion error as in case of the first two programs. Note, however, that this situation is quite rare as the threads may usually acquire the lock only a moment before the problematic release is done.

Table 2: Success ratio of finding assertion errors for various configurations of the noise injection (the values represent the percentage of runs which ended with an assertion error)

Noise configuration \ Program	t02	t12	t14
normal run	0.0	0.0	0.0
instrumented, no sleep or yield noise	48.0	50.8	8.0
sleep 500 10	0.0	0.0	1.2
yield 500 10	**62.4**	**51.0**	**8.8**
rs-sleep 500 10	3.2	1.0	3.8
rs-yield 500 10	41.2	48.8	8.0
sleep 100 10	2.0	27.8	7.2
yield 100 10	49.6	**51.2**	6.6
rs-sleep 100 10	16.4	32.8	6.8
rs-yield 100 10	44.2	**56.2**	**8.8**
sleep 500 20	0.0	0.0	2.6
yield 500 20	**64.6**	**55.2**	6.6
sleep 500 5	0.0	0.0	3.2
yield 500 5	58.6	48.4	8.2
sleep 100 20	4.2	26.4	2.0
yield 100 20	56.6	47.4	7.4
sleep 100 5	21.2	25.6	4.6
yield 100 5	51.0	49.4	8.0
sleep 500 10 / read 20 / write 5	0.0	0.0	5.2
sleep 500 10 / read 5 / write 20	0.0	0.0	6.0
yield 500 10 / read 20 / write 5	**62.4**	0.0	7.6
yield 500 10 / read 5 / write 20	**64.0**	0.0	**10.4**
sleep 100 10 / read 20 / write 5	7.4	0.0	5.4
sleep 100 10 / read 5 / write 20	9.2	0.0	4.6
yield 100 10 / read 20 / write 5	49.6	0.0	6.2
yield 100 10 / read 5 / write 20	54.6	0.0	7.0
sleep 500 10 / read sleep / write yield	2.2	0.0	3.4
sleep 500 10 / read yield / write sleep	0.0	0.0	3.0
yield 500 10 / read sleep / write yield	0.2	0.0	2.8
yield 500 10 / read yield / write sleep	0.0	0.0	1.6
sleep 100 10 / read sleep / write yield	50.2	0.0	6.4
sleep 100 10 / read yield / write sleep	22.4	0.0	4.4
yield 100 10 / read sleep / write yield	60.6	0.0	**9.4**
yield 100 10 / read yield / write sleep	47.4	0.0	3.4

As can be seen from Table 2, running the instrumented program without any noise gives us already good results when trying to detect a wrong usage of the pthread library. However, this is because we are in fact injecting a very weak noise into the execution as we let the framework execute the noise injection code (although it inserts no noise) which by itself disturbs the scheduling of the threads. In runs with no instrumentation (even when the program is running within the PIN framework), the success ratio is practically zero. The yield noise may sometimes help us to achieve slightly better results while the sleep noise is mostly rather hiding the errors. In case of `t14`, the success of encountering an assertion error is very dependent on when and where the noise is injected and not so much on the noise settings used, since the wrong usage manifests only in a very specific situations. So if we manage to inject the noise in the right place and at the right time, even weak noise will help.

5.4 Testing C/C++ vs. Java Programs

We also tried to compare how much the various types of noise help in case of C/C++ programs compared to Java programs. Since implementing the same program in two different programming languages in a way that the implementation is as close as possible is not an easy task, we have chosen a simple `bank` program for the tests, which is one of the typical and often used case studies [5]. This program contains a data race on a shared variable accessed usually in a critical section, but sometimes also outside of it.

Table 3: Success ratio of the AtomRace detector for various configurations of the noise injection (the values represent the percentage of runs in which a data race was found)

Noise Type	rs-sleep					
Frequency	500			100		
Strength	20	10	5	20	10	5
C++	98.8	99.0	100.0	91.6	91.8	91.2
Java	100.0	100.0	99.6	99.4	99.8	99.0

Noise Type	rs-yield					
Frequency	500			100		
Strength	20	10	5	20	10	5
C++	67.6	63.8	58.8	52.0	41.8	52.4
Java	36.2	29.0	24.2	22.4	21.8	28.6

To test the Java version of the program, we used ConTest [3] together with a Java implementation of AtomRace [7]. Since the ConTest tool uses random strengths, we used the `rs-sleep` and `rs-yield` noise in the C++ version of the program to be able to compare them to the corresponding ConTest's types of noise. The results are shown in Table 3. The values express the percentage of runs (out of 500) in which the data race detector found a data race, i.e., the percentage of executions which actually lead to an error.

We can see that while the sleep noise is a little more helpful in case of the Java version of the program compared to the C/C++ version, giving the nearly 100% success ratio even for lower frequencies, the yield noise is clearly better in case of the C++ version, helping to find a data race in twice as many runs as in the Java version. A question that remains is whether the differences are caused by the programming language itself or whether they depend on the concrete thread management and synchronisation library used.

6. CONCLUSIONS

We have discussed several typical problems which arise when monitoring multi-threaded C/C++ programs at the binary level in order to allow for their testing and/or dynamic analysis, and we have proposed solutions to these problems. We have also proposed an improvement of the noise injection technology to be used to increase chances of spotting an error when testing or dynamically analysing a multi-threaded C/C++ program. We have experimentally validated the proposed solutions on a set of C/C++ programs, and we have also discussed the effect of various noise settings when dealing with the considered programs.

For the future, there are several interesting directions that can be taken. First, we would like to improve our implementation of the proposed ideas, extend it by a support of more C/C++ concurrency libraries, and test the resulting tool on larger concurrent C/C++ programs. We would also like to implement more dynamic analyses on top of our framework and include their evaluation into the further experiments. Next, although the two simple types of noise that we currently support belong among the most commonly used ones and they might often be sufficient, it is interesting to perform experiments with other known types of noise too. Moreover, an interesting topic for future research is to try to introduce some more sophisticated types of noise, e.g., tailored for a specific detector or type of concurrency errors. Finally, since using different noise injection configurations for the read and write accesses proved to be useful in our experiments, it may be interesting to look more into the fine-grained use of noise and find more rules how to use it, perhaps supported by some preliminary analysis of the program at test or allowing the noise settings to be automatically suitably adjusted during a test execution based on the so-far obtained results.

Acknowledgement. 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 BUT projects FIT-11-1 and FIT-12-1.

7. REFERENCES

[1] R. Agarwal and S. D. Stoller. Run-time Detection of Potential Deadlocks for Programs with Locks, Semaphores, and Condition Variables. In *Proc. of PADTAD'06*, pages 51–60, ACM Press, 2006.

[2] Y. Ben-Asher, Y. Eytani, E. Farchi, and S. Ur. Noise Makers Need to Know Where to be Silent - Producing Schedules That Find Bugs. In *Proc. of ISOLA'06*, pages 458–465, IEEE CS, 2006.

[3] 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.

[4] A. Jimborean, L. Mastrangelo, V. Loechner, and P. Clauss. VMAD: An Advanced Dynamic Program Analysis and Instrumentation Framework. In *Proc. of CC'12*, LNCS 7210, pages 220–239. Springer, 2012.

[5] B. Křena, Z. Letko, R. Tzoref, S. Ur, and T. Vojnar. Healing Data Races On-The-Fly. In *Proc. of PADTAD'07*, pages 54–64, ACM Press, 2007.

[6] M. Laurenzano, M. Tikir, L. Carrington, and A. Snavely. PEBIL: Efficient Static Binary Instrumentation for Linux. In *Proc. of ISPASS'10*, 2010.

[7] Z. Letko, T. Vojnar, and B. Křena. AtomRace: Data Race and Atomicity Violation Detector and Healer. In *Proc. of PADTAD'08*, pages 1–10, ACM Press, 2008.

[8] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1999.

[9] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. In *Proc. of ASPLOS-XII*, ACM, 2006.

[10] 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*, pages 190–200, ACM Press, 2005.

[11] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs. In *Proc. of OSDI'08*, pages 267–280, USENIX, 2008.

[12] N. Nethercote and J. Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proc. of PLDI'07*, pages 89–100, ACM Press, 2007.

[13] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multi-threaded Programs. In *Proc. of SOSP'97*, pages 27–37, ACM Press, 1997.

[14] E. Trainin, Y. Nir-Buchbinder, R. Tzoref-Brill, A. Zlotnick, S. Ur, and E. Farchi. Forcing Small Models of Conditions on Program Interleaving for Detection of Concurrent Bugs. In *Proc. of PADTAD'09*, pages 1–6, ACM Press, 2009.