Analysis and Monitoring of Transactional Memory Programs

STSM Scientific Report

Jan Fiedor

FIT, Brno University of Technology, Czech Republic
ifiedor@fit.vutbr.cz

Reference Number	COST-STSM-IC1001-14636
Grant Number	ECOST-STSM-IC1001-150913-034425
COST Action	IC1001
STSM Topic	Analysis and Monitoring of Transactional Memory Programs
STSM Period	2013-09-15 to 2013-12-14
Participant	Jan Fiedor
	FIT, Brno University of Technology
Host	Prof. Joao Lourenco
	FCT, Universidade Nova de Lisboa

1 Introduction

This report presents the work done during my stay at FCT UNL. The objective was to adapt ANaConDA, a framework for monitoring and analysing multi-threaded C/C++ programs, to be able to monitor and analyse C-based transactional memory programs. This objective can be divided into two separate topics. The first topic deals with the question how to monitor C-based transactional memory programs while minimally influencing their original behaviour and still get all the information needed for various analyses. The second topic is related to how to find errors in C-based transactional memory programs, mainly using a dynamic analysis which can be performed on-the-fly during the actual monitoring.

2 Purpose of the STSM

Due to the rapid expansion of multi-core and multi-processor computers in the last decade the amount of programs utilizing many threads working in parallel is rising significantly. The main reason of switching to multi-threaded programs is to achieve maximum speed-up by utilizing all of the available resources, e.g., utilize all available cores of a multi-core computer. However, developing multi-threaded programs is far more demanding than the development of the usual single-threaded programs. The programmer must ensure a proper synchronization of all of the threads running in parallel. It is very easy to make an error when writing a multi-threaded program. On the other hand, detecting these errors can be quite difficult as they may manifest only in a very rare situations, e.g., only when a specific timing of all of the operations performed by the threads running in parallel is achieved. Nowadays, there are attempts to develop new techniques for thread synchronization which would make the development of multi-threaded programs easier and thus lower the amount of errors introduced during their development.

Currently, the most common approach to synchronize threads of a multi-threaded program is to use some sort of locking. There are several ways how to do the locking. However, all of them are either too complex to use or provide a very poor performance, often similar to the performance of a single-threaded program. One of the promising approaches, which is both easy to use and at the same time provides a very good performance, is transactional memory [6, 5]. In transactional memory, the synchronisation is done by defining transactions which might be executed in parallel as long as they do not interfere with each other. Even though it is really easy to use transactional memory, there are still various opportunities to make errors. There are also some issues that might influence the performance of transactional memory, which many of the programmers might not be aware of. Therefore, tools for detecting errors and performance issues in transactional memory programs are much needed.

In this report, we propose ways how to monitor C-based transactional memory programs and how to minimize the impact of the monitoring on the behaviour of the monitored program. We perform experiments to evaluate how the proposed ways behave on different programs and discuss possible improvements that can be a subject of future research. Further, we study some fo the approaches for detecting errors in transactional memory programs and propose a modification to one of the algorithms which allows one to detect high-level data races in large real-world programs. We also discuss some of the limitations of the algorithm which we found when analysing more complex real-world programs.

3 Description of the work carried out during the STSM

In this section, the work done during the STSM is described. After a brief introduction of transactional memory, which is the main topic of the STSM, the two main objectives of the STSM, the monitoring and analysis of transactional memory programs, are described in more detail.

3.1 Transactional Memory

This section provides a brief overview of transactional memory. This includes a description of how transactional memory work, what are the differences between transactional memory and the usual approaches to synchronise threads, and what are the challenges that need to be dealt with when using transactional memory.

As was mentioned in the introduction, the usual approach to synchronize threads of a multi-threaded program is to use some sort of locking. There are various ways how to do the locking, however, the two most prevalent ones are the use of a global lock and the so-called *fine-grained* locking. A global lock is a single lock which is used to synchronize all of the threads of a multi-threaded program. This approach is quite simple to use and thus less prone to errors. However, using a global lock usually degrades greatly the overall performance of a multi-threaded program as only one thread might be performing the computation at a time like in case of a single-threaded program. Fine-grained locking uses different locks to synchronise specific parts of a multi-threaded program, often granting superior performance compared to using a global lock at the cost of greatly increasing the complexity of a multi-threaded program. The programmer must be sure that the specific parts of the multi-threaded program might be executed concurrently, which is a very difficult task considering the complexity of a multi-threaded computation. Wrong assumptions about the computations which might be performed in parallel might lead to various errors, e.g., data races, atomicity violations, etc., which are very hard to detect.

Transactional memory [6, 5] takes the best of both of the approaches mentioned above providing an easy to use way to synchronise threads (similar to the use of a global lock) while still keeping the performance of a multi-threaded computation as high as possible (like when using the complex fine-grained locking). When using transactional memory, the thread synchronisation is done by defining transactions, which is of similar complexity as using a global lock. Instead of acquiring and releasing a global lock before and after some code, respectively, the programmer just puts the code into a transaction. The performance is then achieved by executing the transactions optimistically, i.e., assuming that in most cases the transactions might be executed in parallel without interfering with each other. When two transactions finish their execution without any interference with each other, we achieve the best performance possible as everything ran in parallel. If we use the fine-grained locking and know that the two pieces of code never interfere with each other and thus can be safely executed in parallel, we would protect each of the pieces of code using a different lock and achieve the same performance. However, note that this requires much more effort from the programmer than just simply putting the code into a transaction. Also if the two pieces of code might interfere only in some rare cases, we would have to use the same lock to protect them and degrade the performance, while the transactional memory would allow the pieces of code to be safely executed in parallel if no interference occurs. From this point of view, the transactional memory can be seen as a more reactive approach, which assumes the best and fix the problems when they arise, compared to the more proactive locking approach, which have to assume the worst case scenarios and prevent even the rare situation, where the threads are interfering with each other, from happening.

Nevertheless, although transactional memory might be easy to use and prevent the programmers from making many potential errors, it does not mean that there could not be errors in transactional memory programs. Some errors arise when combining transactional memory with other approaches for synchronising threads, e.g., locks, or when part of the code is not synchronised at all. Errors like weak-isolation data races [10] belongs to this category. Other errors are specific to transactional memory itself, e.g., write-skew errors, which arise when the transactions are not properly isolated from each other [2]. Therefore, tools and algorithms for detecting these kinds of errors are much needed.

Errors are not the only problem to deal with when using transactional memory. As transactional memory is designed to provide high performance of a multi-threaded computation, achieving good performance is often as important as ensuring that there are no errors in the program. However, transactional memory cannot guarantee that the performance of a multi-threaded computation will be good enough. It depends on many things from the underlying hardware to the specific nature of the multi-threaded computation in the program. Therefore, tools and algorithms for analysing the performance of transactional memory programs and concrete implementations of transactional memory itself are also needed.

3.2 Monitoring Transactional Memory Programs

This section discusses several ways of how to monitor an execution of a transactional memory program. For each of these approaches, we mention its advantages and disadvantages. Then, we describe in more detail one of the approaches we think is the most suitable for our purposes and provide some results of using this approach in practice. At last, we discuss possible future work in this area.

3.2.1 Approaches for Monitoring Transactional Memory Programs

In order to perform any kind of analysis, one first needs the information about the execution of a program. There are various ways to extract information from the execution of a program which mainly differs on how much they influence the monitored program, how much automatic they are, and how much information they are able to provide.

When taking into account the influence on the monitored program, we distinguish between lightweight and heavyweight approaches. Lightweight monitoring does not influence the original behaviour of a program much and the slowdown of the execution when the monitoring is performed is usually minimal. This makes the lightweight monitoring very convenient when analysing the performance of a program where even a slight slowdown of the execution might lead to inaccurate analysis results as we get a different behaviour that usual. The problem with lightweight monitoring is that we only get a very limited amount of information as some information is just too time consuming to obtain for a lightweight monitoring. On the opposite, we have the heavyweight monitoring, which impose a significant slowdown on the execution of a program. Because of the slowdown, the behaviour of the program when the monitoring is performed is usually very different which might be a problem for some kinds of analyses. The biggest advantage of this approach is that we can get nearly any information we need which might be the main difference between being able to perform some analysis or not.

Another difference between the various monitoring approaches is at which level of abstraction they obtain the information needed to perform some analysis. In order to obtain the information about the execution of a program, the program must be instrumented in some way. The instrumented code then performs the monitoring and extracts the information needed when the program is running. One possibility is to instrument the source code of the program. This way, we can get a very accurate information, however, the instrumentation is usually far from automatic and the programmer must instrument the source code himself manually. Another problem is when we need to monitor not only the program itself, but also the libraries it is using. Changing the source code of the libraries is often impossible due to various reasons (no source code, the library is used by other programs, etc.). Also there is a risk that the compiler might optimize the source code in various ways, removing some parts of the code not needed or adding some code like temporary variables if it speeds-up the execution. These changes might influence some kinds of analyses.

On the other hand, we may instrument the binary of the program directly. This approach has a big advantage of not requiring the source of the program to be monitored and also any of its libraries we might want to monitor too. As we are working directly with the code which is executed by the processor, we do not have any troubles with the possible optimizations as we see them directly. The biggest problem here is the slowdown and a more problematic access to some higher level information (like names of variables accessed etc.). The slowdown is in particular visible when using the dynamic binary instrumentation, which is very generic and can handle even self-modifying and self-generation code, but requires a low level virtual machine to execute instructions of the monitored program which imposes a significant slowdown. Also accessing information like the name of variables accessed is much harder as we have to get this information from the debugging information in the program (which might not even be there).

3.2.2 Heavyweight Monitoring Minimally Influencing the Program

As our primary focus is the detection of errors in transactional memory programs, we need a monitoring approach which is able to give us very detailed information about the execution of a program while requiring minimal input from the programmer, i.e., be as automatic as possible.

A lightweight monitoring seems to be the best option here as we would not slow down the program being monitored much and keep its original behaviour. This would allow us to use the monitoring framework not only for error detection, but also for various performance analyses. There is already a tool which does this kind of lightweight monitoring [9]. However, it suffers from several critical flaws. First is that it does not allow us to get all the information needed to perform the kinds of analyses we need. For example, when detecting weak-isolation data races, we need to get information not only about accesses to the transactional memory, but also to the normal memory. The tool is not able to provide us the information about the second type of accesses, only about the accesses to the transactional memory. The problem is that obtaining this information would make the monitoring considerably more heavyweight and we would lose all the advantages. Another disadvantage is that the instrumentation of the program being monitored is done at the source code level and the code must be instrumented manually by the programmer.

As we need to monitor even things like accesses to a (non-transactional) memory and monitoring all accesses to a memory in a program would make any monitoring quite heavyweight, we have chosen a different approach. Instead of using a lightweight monitoring and losing many crucial information for our analyses, we use a heavyweight monitoring, which changes the original behaviour of the monitored program, and try to revert the behaviour of the program back to the original one during the monitoring, i.e., we try to minimize the disturbance to the program we have caused by the heavyweight monitoring. In order to do that, we use the noise injection techniques. The noise injection techniques are usually used to disturb the scheduling of threads in order to force the program to exhibit some rare behaviour which might lead to an error. However, there are also works [7] which use the noise injection techniques to fix the errors instead of detecting them. In our case, the idea is to use the noise injection techniques to somehow negate the disturbance caused by the heavyweight monitoring, which can be seen as a form of noise itself. So, in other words, we are trying to reduce the disturbance caused by one type of noise by injecting a different type of noise, which should minimize the effects of the first one. The advantage of this approach is that we will get all the information needed for the analyses while observing a behaviour very similar to the original one. The monitoring will slow down the execution of the monitored program, but as long as it will exhibit a similar behaviour to the one without the monitoring, we will be able to do even performance analyses without worrying much that the results might be too inaccurate.

Instead of making a completely new tool to implement the approach states above, we extended the ANaConDA framework [4], a framework for monitoring the execution of multi-threaded C and C++ programs on the binary level, to support monitoring of transactional memory. As transactional memory implementations for C and C++ are usually in a form of a library, the extension in not restricted to a specific library, but it may be instantiated for any library used. This is important as it allows us to analyse a broad variety of transactional memory programs and not only a subset of programs using a specific library. Using the ANaConDA framework is also very convenient for us as it has a built-in support for noise injection. Moreover, the noise injection in ANaConDA might be configured in a very detailed way allowing us to test a broad variety of noise injection settings.

3.2.3 Experimental Results

To evaluate if the noise injection techniques can be used to fix the behaviour of a program when its behaviour is altered because of a heavyweight monitoring, we performed a large number of tests. We have used the programs from the STAMP benchmark [1] and compared the number of aborts in the original run with the number of aborts in the runs where the program was monitored using the ANaConDA framework. The number of aborts characterize the behaviour of transactional memory programs very well. It tells us how much the threads of a program interfere with each other which directly affects the actual performance of the transactional memory, i.e., it might be used to determine if we really get a significantly better performance when using the transactional memory compared to, e.g., using a global lock.

The results of our experiments are shown in Table 1, each column representing one of the tested programs, namely the *genome*, *intruder*, *kmeans*, *vacation* and *yada* programs, and each row one of the configurations of the noise injection in the following format: First, a configuration of noise generation is given, consisting of the type of noise, its frequency, and its strength. The values of the frequency say how probable it is

[1] http://stamp.stanford.edu

that some noise will be generated every time the given transactional memory operation 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, actively or passively, wait in case of the busy-wait and sleep noise, respectively. Then, it is specified where the noise might be inserted, if before any transactional memory operation or only before an operation of a specific type (start, abort, commit, read or write). Also, beside the special configuration representing the original run of the program, four other configuration where no noise is injected are present. The first two configurations represent runs which where executed only using the Intel's PIN framework [11], which performs a dynamic binary instrumentation and on top of which the ANaConDA framework is built. In the first configuration we instrumented each instruction with a simple code incrementing a counter and counted the number of instructions executed. In the second configuration, we did not instrument anything, just let the program to run in the low-level virtual machine PIN is using. The second two configurations represent runs where we monitored specific transactional memory operations without injecting any noise.

The results show that when performing a heavyweight monitoring of transactional memory programs, we get a very different behaviour as the number of aborts drops rapidly. When we use noise injection, we increase the number of aborts and get closer to the original behaviour, however, how close we get depends on the program and also on the noise configuration which we use. For some of the programs, e.g., genome or intruder, we can get very close to the original behaviour, when we use the right configuration. For other programs, e.g., kmeans, vacation or yada, we are quite far from the original behaviour whether we use any of the configurations we tried.

From the results, one can also see that if we instrument each instruction with the same code, we actually get a huge slowdown, far bigger than when performing the standard monitoring, but often get quite close to the original behaviour. This is a very interesting and valuable observation as it tells us that even with a massive slowdown, we can achieve a behaviour similar to the original one. The main difference is that when instrumenting each instruction we slow every part of the program equally. On the other hand, when performing the usual monitoring, we instrument only a specific, interesting, parts of the program and thus slow down specific parts of the program significantly while keeping the rest of the program to execute normally [2]. Another interesting thing is that when we get really close to the original behaviour while instrumenting each of the instructions, we do not achieve a very good results when using the noise injection, and vice versa. Compare the results of, e.g., vacation or yada, with the results of, e.g., genome and intruder. When counting the instructions (instrumenting all instructions equally) of vacation or yada, we get very close to the original behaviour (number of aborts). However, when using any of the configurations we have tried, the results were better compared to the standard monitoring without the noise injection, but still quite far from the original behaviour. On the other hand, when counting the instructions of

[2] When using dynamic binary instrumentation which executes each instruction in a virtual machine, we are in fact slowing down every instruction we execute. However, this slowdown is exactly the same for each instruction in the program, so we may consider this as a normal execution.

Table 1. Number of aborts in transactional memory programs for various noise configurations.

Configuration	genome	intruder	kmeans-high	kmeans-low	vacation-high	vacation-low	yada
original run	29255.9	28801.5	6227575.4	4373338.3	370051.7	32959.5	4951769.4
PIN (counting instructions)	6297.7	1249.8	1371294.5	3123090.1	282820.1	26851.2	4954548.0
PIN (no instrumentation)	1514.2	79.3	9.2	21.7	14909.0	1342.0	256262.8
monitoring starts, commits, aborts	1947.0	83.8	40.2	118.8	20017.3	2207.8	310817.9
monitoring all operations	8142.5	798.9	38246.2	81588.7	87406.5	14354.0	731750.7
busy-wait 100 1 before commits	2976.8	1104.9	388882.6	801797.7	37492.5	8393.9	371061.1
busy-wait 100 1 before all operations	-	11392.2	-	-	-	-	-
busy-wait 250 1 before commits	2173.9	2797.5	899273.6	1497110.9	33490.3	8930.5	443921.8
busy-wait 500 1 before commits	1822.1	7072.6	1531912.6	-	31700.2	9047.8	813450.5
sleep 10 1 before reads	27030.7	2281.8	758608.2	1017451.2	-	-	-
sleep 10 1 before read and writes	26434.1	2638.8	1126005.8	-	-	-	-
sleep 10 1 before writes	3997.1	678.0	747058.2	1059902.0	41211.6	8660.4	-
sleep 100 1 before commits	3050.0	1220.1	415349.8	801339.6	38678.6	8861.8	325461.0
sleep 100 1 before all operations	-	10873.5	-	-	-	-	-
sleep 100 2 before commits	2154.3	1111.2	396614.5	859134.8	35446.1	9274.5	287055.3
sleep 250 1 before commits	2099.2	2879.6	858333.0	1663104.8	34741.2	9129.5	432395.1
sleep 250 2 before commits	1829.6	2889.0	874936.3	-	32514.7	9321.3	376473.4
sleep 50 1 before commits	4053.5	806.9	228278.7	443974.1	40982.8	8651.3	394146.1
sleep 50 2 before commits	2793.0	708.3	218307.1	497465.9	38865.7	9002.5	331012.1
sleep 500 1 before commits	1802.2	7257.9	1495551.6	-	32828.8	8858.3	831001.8
sleep 500 2 before commits	1689.5	7305.2	-	-	-	-	812413.5
yield 100 10 before aborts	4506.7	323.4	7539.9	29898.7	43185.0	6716.8	348787.1
yield 100 10 before commits	3799.8	379.2	20357.3	38434.5	42934.8	6755.9	341957.7
yield 100 10 before reads	15140.0	2784.1	249778.7	226270.3	50244.1	10260.4	612571.5
yield 100 10 before reads and writes	15283.7	3160.7	489785.4	497181.9	50575.5	10228.8	-
yield 100 10 before starts	3713.2	330.6	14853.5	27769.0	42943.9	6738.2	346416.0
yield 100 10 before writes	3393.4	664.6	235478.5	257401.1	43504.6	7277.0	433923.7
yield 100 10 before all operations	30085.5	6717.6	960007.9	977732.4	99212.8	20279.6	-
yield 100 100 before all operations	25330.8	11647.1	1646973.2	1779226.2	-	-	-
yield 100 50 before all operations	45350.4	21593.5	2950867.4	3200529.2	99132.3	20436.6	-
yield 1000 10 before all operations	25575.7	12140.3	1624598.1	1848199.1	-	-	-
yield 500 10 before all operations	45390.2	22635.0	3006748.5	3207251.9	99500.9	20483.6	-

genome or intruder, we are very far from the original behaviour, yet when we use the noise injection, we are able to achieve a similar behaviour as the original one.

3.2.4 Current and Future Work

Although we were able to fix the behaviour (achieve a similar behaviour to the original one) for some of the programs, there are still other programs where we were not that successful. Based on what we learned from the experiments, we came with several ideas how to improve the proposed monitoring approach which we would like to try in the near future.

Some of the noise configurations give us clearly better results that others. In our experiments, we have used a large amount of different noise configurations. We have chosen these configurations based on our previous experience with using the noise injection techniques to detect errors in multi-threaded programs and based on the results from the previous batches of tests we got. Still, all of the configurations were chosen manually and represent only a very small portion of all of the possible configurations. That means that even when we were not able to find a configuration which would fix the behaviour for some of the programs, it does not mean that there is no configuration able to do that. As many the configurations working well for error detection proved not to be much useful here, there might be configurations which proved to be ineffective for error detection which might we very suitable for the problem we are solving here. The problem is that it is not possible to check all of the possible configurations manually. However, we can take advantage of search-based testing and genetic algorithms and use them to automatically find a more suitable noise configurations for solving our problem. We have a tool called SearchBestie [8], a tool for search-based testing with a support for genetic algorithms, which we can use with the ANaConDA framework to search the space of possible noise configurations for the ones which gives us the best results, e.g., which will give us a similar number of aborts than in the original run.

Another thing we want to try is to develop new types of noise which might be more suitable for fixing the behaviour or to use the noise injection techniques in a different way. The types of noise we have used in our experiments all delay the execution of a thread in some way. These types of noise are usually very effective when used for error detection as many errors manifest when a specific timing of actions performed by the threads occurs. The transactions behave a little differently so we want to try new types of noise more suitable for them. One example is the *abort* noise, which forces a transaction to abort and directly influence the number of aborts. In the experiments, we were also injecting the noise in the parts of the program we were monitoring, which is the usual way of using the noise injection techniques. When using the noise to fix the behaviour of a program, this might not be the best way to do it. We have already discussed that slowing down (delaying) only some parts of the program greatly alters its behaviour. Injecting noise into these parts might just make the situation even worse. It would be interesting to try to inject the noise on the contrary to the parts not slowed down by the monitoring and see if we will achieve a similar results as when instrumenting all the instructions equally.

3.3 Detecting Errors in Transactional Memory Programs

This section describes some of the errors which can be found in transactional memory programs. After a brief description of these errors and some related work around them, we focus on one of these errors, the high-level data races. For this kind of errors, we propose a dynamic analysis algorithm able to detect the high-level data races. Finally, we discuss some problems of this algorithm which cause false alarms.

3.3.1 Errors in Transactional Memory Programs

As was mentioned in Section 3.1, there are two categories of errors one may find in transactional memory programs. To the first category belongs the errors which arise when combining transactional memory with other approaches for synchronising threads, e.g., locks, or when part of the code is not synchronised at all, i.e., not protected by neither a transaction, nor any other type of synchronisation. The second category consists of errors specific to the transactional memory itself. These errors are often related to the specifics of the implementation of transactional memory.

Probably the most well-known type of errors in transactional memory programs is the weak-isolation data race [10] error. This error is very similar to the (low-level) data race error frequently appearing in programs using, e.g., locks, for synchronisation. A data race occurs when two threads access the same memory location without any synchronisation and at least one of these accesses is a write access. A data race usually occurs when some of the accesses are in a critical section guarded by a lock, but the remaining accesses are not guarded at all. When using transactional memory, the programmer might cause a similar situation where he fails to put all of the accesses into transactions leaving some of them unguarded. This problem might also occur when combining transactional memory with other means of synchronisation, e.g., locks. If part of the accesses are in transactions and the other part in critical sections guarded by locks, a data race might still occur as the transactions are executed independently from the code in the critical sections. This situation is similar to guarding part of the accesses to the same memory location with one lock and the remaining accesses with a different lock.

However, fixing all the (low-level) data race errors in a program might not be enough, as the program might still contain the so-called high-level data race [10, 3] errors. A high-level data race occurs when several memory locations (variables), which must always be updated atomically (to ensure consistency), are updated separately. For example, take a 2D point containing the x and y values. When updating the point, both x and y must be changed at once, without any interference from other threads. Even when all accesses to x and y are situated in transactions, and no (low-level) data race is possible, if they are in two separate transactions, we might still get an inconsistent state if two threads try to update the point at the same time. This kind of error is not specific to transactional memory as it may also occur when using other types of synchronisation like locks. However, as one of the ways of improving the performance of a transactional memory program is to split its long transactions into a several smaller ones, which might cause high-level data races, they are more frequent in transactional memory programs.

Another type of errors one might encounter in transactional memory programs is the stale value [3] error. This error is more specific to the transactional memory programs and occurs when a value of a variable stored in a transactional memory leaves the scope of a transaction, e.g., is stored to a temporary variable outside the transactional memory, and is later used in another transaction instead of the current value of the variable stored in a transactional memory.

The last type of errors we will mention is the write-skew [2] error. This error is very specific to the transactional memory as it occurs when the transactions are not properly isolated from each other. Whether this error might arise in a transactional memory program greatly depends on the actual implementation of transactional memory. This also shows that it is not only important to test transactional memory programs, but also the concrete implementations of transactional memory.

3.3.2 Detecting High-level Data Races

There are several works dealing with the detection of high-level data races. When the high-level data races were introduced in [1] for the first time, the authors proposed an algorithm based on collecting and analysing the so-called *views*. A view is a set of accesses performed in an atomic region, which might be a critical section guarded by a lock, but it may also be a transaction for our purposes. These views are checked against the views of other threads if they do not violate the so-called view compatibility. When a violation is found, a high-level data race is reported. The authors tested the algorithm on a small set of Java programs, first monitoring their execution and logging the views encountered and then analysing the logs post mortem to check if there is a high-level data race. The problem with this approach is that when the program is data intensive, the logs become huge (more then 0.5 GB) and are then hard to analyse in a reasonable time. Also, as was later shown, the algorithm may miss some high-level data races and report false alarms.

A similar approach based on the views can also be found in [3], where the algorithm from [1] is refined to distinguish between read and write accesses and extended with the information given by the so-called causal dependencies, which tells the algorithm if a value read from one variable influence the value written into the other. Because of this modifications, the algorithm is more precise and detect some high-level data races which was the original algorithm unable to find. On the other hand, the refined version of the algorithm reports more false alarms than the original one. Also one drawback of the refined version of the algorithm is that it is a static analysis algorithm and is not very suitable for larger real-world programs, where it will not scale well.

Our approach is also based on the algorithm from [1] and refined to distinguish between the read and write accesses as proposed in [3]. To to able to analyse real-world programs, we implemented the algorithm as a dynamic analysis algorithm and restricted it to look only at the last N views of each thread, which we call a window. This allows us to deal with the problem of having a huge amount of views which will take ages to analyse. The first experiments have also shown that even when remembering only the several last views for each thread we do not miss any high-level data races. However, we were able to check this fact only for simple examples where we know the high-level data races contained in them. Now we are trying to determine how the size of the window influences the high-level data races found in a more complex examples. Another difference, compared to the approaches mentioned previously, is that we analyse C and C++ programs, where the tools for the detection of high-level data races are still lacking. Still, the algorithm reports many false alarms in real-world programs, because of various reasons discussed in the next section.

3.3.3 Sources of Detection Inaccuracies

There are several problems which influence the precision of the detection algorithm. The first source of inaccuracies are the join operations. The algorithm checks if there is a violation of a view consistency if a pair of views of one thread is interleaved with a view of another thread. However, the algorithm performs an approximation and checks even the interleavings not witnessed in the actual execution. In most of the cases, the approximated interleavings are all possible, however, the join operation clearly tells us that a view from a joined thread cannot interleave any pairs of views which occurred after the join in other threads.

Another problem is the ordering of some operations. Some of the false alarms we encountered were related to the ordering of queue operations. If we have a set of operations performed in an atomic region (critical section, transaction, etc.) and these operations are insert an item to a queue, get an item from a queue and update the item, we get a view each time the operation is executed. Then the algorithm might check interleavings where getting an item from a queue and updating it is interleaved with the insertion of this item into the queue. This sequence of operations is clearly not possible, however, the approximation causes the algorithm to check this situation and possibly report a false alarm.

Next problem is related to the use of nested atomic regions. The simple examples used in [1, 3] never contained any nested atomic regions so the algorithm itself is not designed to deal with them to begin with. In our case, we flatten the nested atomic regions, i.e., we merge all of the nested atomic regions into a single large atomic region. However, this approach leads to a number of false alarms as the algorithm assumes that all the accesses in an atomic region should be performed atomically everywhere in the program and the merging causes the algorithm to think that some accesses must be performed atomically even when they are in fact not in the same atomic region.

We are currently working on ways to solve these problems as the previous works did not provide any solutions to them, mostly because these problems never occurred in the simple programs which were used for the experiments.

4 Conclusion of the work carried out during the STSM

To summarize the work done during my stay at FCT UNL, I mainly focused on problems of monitoring and analysing transactional memory programs.

Regarding the first objective, the monitoring of the execution of transactional memory programs, I extended the ANaConDA framework to be able to monitor transactional memory. The extension is quite generic and allows monitoring of programs using various transactional memory implementations, which greatly increases the amount of programs that may be monitored. Then we studied the possibilities of using noise injection techniques to fix the behaviour of a program when we are monitoring it using a heavyweight approach, which often alters the behaviour. We performed a large number of experiments on a remote cluster to evaluate our ideas. In order to do that, I have written a set of scripts to automate and simplify everything from installation of the ANaConDA framework to test automation and the evaluation of the tests performed. While we were able to get close to the original behaviour for some of the programs, for others we are still quite far from it. However, based on the results of the experiments we performed, we learned a lot about the behaviour of transactional memory programs and came with several new ideas of how to improve our approach which we plan to try in a very near future.

Considering the second objective, the analysis of transactional memory programs, we proposed a dynamic analysis algorithm based on the algorithm from [1, 3], which can be used to detect high-level data races in large real-world transactional memory programs. I implemented this algorithm as a plug-in for the ANaConDA framework and we tested it on a set of C/C++ programs containing high-level data races. Compared to the algorithms in [1, 3], the algorithm is fairly precise, however, for real-world programs it still reports many false alarms because of a various inaccuracies we already mentioned in Section 3.3.3. We are currently working on ways how to solve these inaccuracies and make the algorithm more precise.

5 Foreseen publications resulting from the STSM

We have not published the results of the work done yet, as we want to implement some of the ideas mentioned in Section 3.2.4 and solve some of the inaccuracies mentioned in Section 3.3.3 first. After that, we plan to publish a paper at an international conference.

6 Future collaboration with the host institution

As was already mentioned in the previous section, there are several problems we want to solve and ideas we want to try, so both participant sides (FIT and FCT) have decided to continue the mutual collaboration. We are also planning another visit at FCT UNL somewhere in the middle of this year and further extend the collaboration initiated by this STSM.

7 Confirmation by the host institution of the successful execution of the STSM

The Host Joao Lourenco from FCT UNL confirms that Jan Fiedor achieved all the targets that were defined for this collaboration. He developed a tool able to monitor and analyse C-based transactional memory programs, implemented and evaluated a new approach of using noise injection techniques to reduce the influence of a heavyweight monitoring on the behaviour of a monitored program, and proposed an algorithm for detecting high-level data races that can be used to analyse large real-world programs.

References

1. C. Artho, K. Havelund, and A. Biere. High-level data races. *Software Testing, Verification and Reliability*, 13(4):207–227, 2003.
2. R. J. Dias, D. Distefano, J. a. C. Seco, and J. a. M. Lourenço. Verification of snapshot isolation in transactional memory java programs. In *ECOOP'12*, pages 640–664, Berlin, Heidelberg, 2012. Springer-Verlag.
3. R. J. Dias, V. Pessanha, and J. a. M. Lourenço. Precise detection of atomicity violations. In *HVC'12*, pages 8–23, Berlin, Heidelberg, 2013. Springer-Verlag.
4. J. Fiedor and T. Vojnar. Anaconda: A framework for analysing multi-threaded c/c++ programs on the binary level. In *RV'12*, volume 7687 of *Lecture Notes in Computer Science*, pages 35–41. Springer, 2012.
5. R. Guerraoui and M. Kapalka. *Principles of Transactional Memory*. Morgan and Claypool Publishers, 2010.
6. T. Harris, J. Larus, and R. Rajwar. *Transactional Memory, 2nd Edition*. Morgan and Claypool Publishers, 2nd edition, 2010.
7. B. Křena, Z. Letko, R. Tzoref, S. Ur, and T. Vojnar. Healing data races on-the-fly. In *PADTAD'07*, pages 54–64, New York, NY, USA, 2007. ACM.
8. B. Křena, Z. Letko, T. Vojnar, and S. Ur. A Platform for Search-based Testing of Concurrent Software. In *PADTAD'10*, pages 48–58, New York, NY, USA, 2010. ACM.
9. J. a. Lourenço, R. Dias, J. a. Luís, M. Rebelo, and V. Pessanha. Understanding the behavior of transactional memory applications. In *PADTAD'09*, pages 3:1–3:9, New York, NY, USA, 2009. ACM.
10. J. Loureno, D. Sousa, B. C. Teixeira, and R. J. Dias. Detecting concurrency anomalies in transactional memory programs. *Computer Science and Information Systems*, 8(2):534–548, 2011.
11. 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 *PLDI'05*, pages 190–200, New York, NY, USA, 2005. ACM.