On Monitoring C/C++ Transactional Memory Programs[*]

Jan Fiedor[1], Zdeněk Letko[1], João Lourenço[2], and Tomáš Vojnar[1]

[1] IT4Innovations Centre of Excellence, FIT, Brno University of Technology, Czech Republic
{ifiedor, iletko, vojnar}@fit.vutbr.cz
[2] CITI, Universidade Nova de Lisboa, Portugal
joao.lourenco@fct.unl.pt

Abstract. Transactional memory (TM) is an increasingly popular technique for synchronising threads in multi-threaded programs. To address both correctness and performance-related issues of TM programs, one needs to monitor and analyse their execution. However, monitoring concurrent programs (including TM programs) may have a non-negligible impact on their behaviour, which may hamper the objectives of the intended analysis. In this paper, we propose several approaches for monitoring TM programs and study their impact on the behaviour of the monitored programs. The considered approaches range from specialised lightweight monitoring to generic heavyweight monitoring. The implemented monitoring tools are publicly available to the scientific community, and the implementation techniques used for lightweight monitoring of TM programs may be used as an inspiration for developing other specialised lightweight monitors.

1 Introduction

Due to the widespread use of multi-core and multi-processor computers in the last decade, the number of programs utilising many threads working in parallel is rising significantly. This switch from sequential to multi-threaded programming aims at achieving maximum speed-up by utilising all of the available cores of a multi-core computer. However, the development of multi-threaded programs is far more demanding than the development of common single-threaded programs, as the programmer must ensure a proper synchronisation of all the threads running in parallel. Failing to do so may lead to various problems including performance degradation and program malfunction. Therefore, there is ongoing research on developing new techniques for thread synchronisation that ease the development of multi-threaded programs.

One of the current approaches aiming at facilitating the development of multi-threaded programs is transactional memory (TM) [4, 5], which is both easy to use and

[*] The work was supported by the ESF COST Action IC1001 (Euro-TM), the COST project LD14001 and the Kontakt II project LH13265 of the Czech ministry of education, the BUT project FIT-S-14-2486, the EU/Czech IT4Innovations Centre of Excellence project CZ.1.05/1.1.00/02.0070, the EU/Czech Interdisciplinary Excellence Research Teams Establishment project CZ.1.07/2.3.00/30.0005, and the research project PTDC/EIAEIA/113613/2009 of the Portuguese National Science Foundation (FCT).

provides good performance. When using TM, the threads are synchronised by defining transactions that may be executed optimistically in parallel and will succeed if they do not interfere with each other. Even though using TM may be easier, there are still various opportunities to make mistakes that lead to performance degradation and errors, which rises a clear demand for tools for analysing and debugging TM programs.

In order to be able to implement various dynamic analyses of the behaviour of TM programs, one first needs to monitor their execution. However, the monitoring code may influence the monitored program's behaviour and hamper the results of some analyses. That is why, in this paper, we propose several different ways of monitoring C/C++ TM programs and then experimentally study their influence on the behaviour of the monitored programs. Our monitoring approaches range from lightweight to heavyweight monitoring. The monitored programs are taken from the well-known STAMP benchmark [1].

As our primary metric for evaluating the influence of the different monitoring approaches, we use the number of transactions that aborted during the execution of the monitored TM programs as this metric gives a good insight into their contention level, i.e., into the number of conflicting concurrent transactions. The more conflicts and aborts the more work for the TM system.

In this paper, we present an experimental evaluation of the influence of different kinds of lightweight and heavyweight monitoring approaches for TM programs, both in terms of global numbers of aborts as well as numbers of aborts for different types of transactions. Moreover, we also show that the obtained results can be significantly influenced by the environment in which the monitoring is performed.

The results presented in this paper can be used in several ways. First, they can show researchers or developers interested in monitoring TM programs how the behaviour of these programs can be influenced by different monitoring techniques as well as the environment. Second, the proposed and implemented monitoring techniques are available to the scientific community and can be used in other settings, which is especially easy for the case of heavyweight monitoring since we implemented a quite generic TM monitoring platform on top of the ANaConDA framework [3]. The lightweight monitoring approaches are rather specialised; however, the described implementation techniques can be useful if there is a need for implementing yet another lightweight monitor.

Related work. To the best of our knowledge, there are only a couple of works dealing with monitoring of TM programs, namely the works [2, 6]. These works aim at providing the users with a variety of interesting data about the execution of a TM program by tracing its operations. However, only the authors of [2] discuss how their monitoring influences the monitored programs, and this discussion is rather brief and addresses only the global number of aborts. We provide a much more detailed study of the influence of monitoring on the monitored programs, using more and/or different monitoring approaches and considering other metrics besides the global numbers of aborts.

2 Monitoring Transactional Memory Programs

In this section, we briefly recall general principles and properties of both lightweight and heavyweight monitoring techniques, and we propose several ways to use these approaches in monitoring TM programs. The influence of these techniques on the monitored programs is then experimentally studied in the next section.

2.1 Lightweight and Heavyweight Monitoring

Lightweight monitoring [6] strives to minimize the impact of the monitoring activity on the behaviour of the monitored TM program. To achieve this goal, only a limited amount of information is collected, mainly the kind of information that can be obtained fast enough and with minimal intrusion. This makes lightweight monitoring particularly suitable for analysing a program for performance issues. To achieve the highest performance, the monitoring code is usually embedded into the monitored program itself by modifying its source or intermediate code, or even its binary. In all these cases, the monitored program is modified and differs from the original one.

Besides the limited amount of information provided, another disadvantage of the lightweight approach is its lack of automation and/or versatility. The program must be modified again and again for each change in the information to be collected, no matter how small that change is. Sometimes, the required information can be acquired by modifying only some of the libraries used by the program (such as the TM run-time libraries in our case), but then the monitoring will be restricted to those programs that use this specific library. Moreover, embedding monitoring code into a library may be problematic if it is being shared with other programs running on the system, requiring one to manage and maintain multiple versions of the same library.

Heavyweight monitoring [7] trades performance for versatility. It frequently uses a specific run-time environment, such as some kind of a low-level virtual machine, to execute the code of the given program and to monitor its execution. Executing the program in such an environment slows down its execution considerably but enables the acquisition of nearly any information required about the execution of the program. Moreover, environments supporting dynamic instrumentation are able to insert (or remove) the monitoring code during the execution of the program, leaving its original code untouched. Finally, by having full control of the code being executed, these environments are able to monitor even self-modifying or self-generating code.

2.2 Lightweight Monitoring of TM Programs

In order to study the impact of monitoring on the behaviour of monitored TM programs, we proposed and implemented several lightweight monitoring approaches. These approaches differ in how much information they are collecting and how they are collecting this information. TM libraries usually provide information about the global numbers of started, committed, and aborted transactions. We take the possibility of obtaining this information as a starting point, and our monitoring approaches allow one to obtain various refinements of this information.

Our lightest monitoring approach (denoted as the *statistics collector* or *sc* in the experiments) allows one to obtain not only the global numbers of started, committed, and aborted transactions, but also all of these numbers separately for each thread and each type of transaction. In order to be as lightweight as possible, this information is obtained in such a way that the monitoring code maintains two counters for each thread and each type of transaction: the first one tracking the number of started transactions and the second one recording the number of committed transactions. These counters are stored in a two-dimensional array so that each combination of a thread and a type of transaction has its own exclusive set of counters. As each thread is accessing a different part of the array, no additional synchronization is introduced. Further, to achieve the best performance, the array is static with a defined maximum number of supported threads and types of transactions, and no boundary checks are done during the monitoring—the monitoring code just accesses a counter and increments it. The numbers of aborts are then computed from the numbers of started and successfully committed transactions.

Our next monitoring approach (denoted as the *event logger* or *el* in the experiments) is based on registering TM operations (events) in an event log (list) during a program execution, followed by a *post mortem* processing of these events. An event is generated (and stored in the event log) only when a transaction starts or successfully commits, and the number of aborts is computed later. In order to minimize the probe effect, each thread has its own event log which resides in the main memory, and hence no additional synchronization between the threads or interaction with the file system is needed[3].

Finally, we have implemented several variants of the event logger. The *el-a* variant differs from the basic event logger in that it is explicitly tracking the aborts and does not compute them from the number of started and successfully committed transactions. The *el-arw* variant does additionally track transactional reads and writes, which significantly increases the number of events collected. Further, we extend all the three above mentioned event logger approaches by collecting and associating a time stamp for each logged event (leading to variants denoted as *el-ts*, *el-a-ts*, and *el-arw-ts* in the experiments). The time stamp is retrieved from the Intel TSC (Time Stamp Counter) register, and storing the time stamp doubles the data size of each event.

The implementation of all of our monitoring approaches is available[4] and can be used either directly or serve as an inspiration for implementing other specialized monitors. The current implementation is restricted to the TL2 library and requires a modification of the source code of the program to be monitored. Since the TL2 library provides a set of macros representing the TM operations and these macros are used by the testing programs, our implementation inserts the monitoring code into the programs by modifying these macros. Thus, the source code of the programs is modified at compile time when the modified macros are being expanded by the compiler. Still, we need to recompile the programs with a different set of macros every time we need to change the way the monitoring is done or the type of information to be acquired.

2.3 Heavyweight Monitoring of TM Programs

For versatile heavyweight monitoring of TM programs, we have proposed and implemented an extension of the ANaConDA framework [3]. The ANaConDA framework is based on PIN [7], a dynamic binary instrumentation tool from Intel. ANaConDA

[3] Eliminating the interaction with the file system is very important as writing to a file introduces a significant intrusion to the execution of a program.

[4] http://github.com/fiedorjan/lightweight-stm-monitoring

enables monitoring of multi-threaded C/C++ programs and allows one to obtain information about common synchronisation operations, such as memory accesses or lock acquisitions and releases. In order to support (heavyweight) monitoring of TM programs, we extended the ANaConDA framework to include a support for monitoring TM operations as described below.

The C/C++ programming languages usually include a support for TM by making use of a software library. In this setting, monitoring the TM operations implies intercepting the calls of the functions in this library. As there are many libraries implementing TM for C/C++, our extension is not restricted to a specific library and may be easily instantiated for any TM library. This allows one to analyse a broad variety of TM programs, not only a subset of programs using a specific library. Regardless of the concrete implementation/library used, TM is supported by five basic operations: three operations for managing transactions (*txStart*, *txCommit*, and *txAbort*); and two operations for managing the transactional accesses to the main memory (*txRead* and *txWrite*).

To be able to monitor the five basic TM operations of a concrete TM library with ANaConDA, the user has to identify which library functions implement these operations and which of their parameters reference memory locations. After that, the extended ANaConDA framework is able to monitor any TM program that uses that particular TM library. Currently, we instantiated the extended ANaConDA framework with a support for monitoring programs that use the TL2-x86[5] or the TinySTM[6] libraries.

We implemented all of the approaches described in the previous sections as plug-ins for the extended ANaConDA framework. The framework monitors the execution of a TM program and sends notifications of the relevant TM events to the plug-in. The plug-in then processes the events in the same way as the lightweight monitoring approaches. Unlike in the case of lightweight monitoring, the heavyweight monitoring does not require customized versions of the monitored program specifically tailored for a particular monitoring strategy. Based on the type of information requested by each plug-in, the framework instruments the original code of the monitored program upon loading it into the main memory with the code which collects the required information.

3 Experimental Evaluation of the Impact of Monitoring

We will now present a set of experiments that evaluate the influence of the monitoring approaches described in the previous section on the behaviour of a set of benchmark TM programs from several different points of view. For our experiments, we used 6 out of 8 programs from the STAMP benchmark suite [1], namely `genome`, `intruder`, `kmeans`, `scca2`, `vacation`, and `yada`. These programs utilise transactional memory to solve a wide variety of problems. In case of the `kmeans` and `vacation` programs, we also distinguish the `high` and `low` variants that use respectively the high and low contention configurations available in the benchmark. The remaining two benchmarks, `bayes` and `labyrinth`, were excluded due to technical problems unrelated with the work described in this paper.

[5] http://stamp.stanford.edu/releases.shtml#tl2-x86
[6] http://tmware.org/tinystm

Table 1. Average number of aborts in original runs and runs with lightweight monitoring.

		genome	intruder	kmeans		ssca2	vacation		yada
	variant			high	low		high	low	
Lightweight	orig	**2.6** $\cdot 10^4$	**4.3** $\cdot 10^7$	**5.6** $\cdot 10^6$	**5.2** $\cdot 10^6$	**2.6** $\cdot 10^2$	**4.9** $\cdot 10^5$	**2.6** $\cdot 10^4$	**2.7** $\cdot 10^6$
	sc	**2.8** $\cdot 10^4$	**4.3** $\cdot 10^7$	**5.4** $\cdot 10^6$	**5.1** $\cdot 10^6$	**3.5** $\cdot 10^2$	**4.9** $\cdot 10^5$	**2.7** $\cdot 10^4$	**2.6** $\cdot 10^6$
	el	**2.3** $\cdot 10^4$	**3.8** $\cdot 10^7$	**4.3** $\cdot 10^6$	**4.0** $\cdot 10^6$	**2.7** $\cdot 10^2$	**4.6** $\cdot 10^5$	**2.5** $\cdot 10^4$	**2.6** $\cdot 10^6$
	el-ts	**2.2** $\cdot 10^4$	**3.5** $\cdot 10^7$	**3.7** $\cdot 10^6$	**3.4** $\cdot 10^6$	**2.0** $\cdot 10^2$	**4.4** $\cdot 10^5$	**2.4** $\cdot 10^4$	**2.3** $\cdot 10^6$
	el-a	**2.3** $\cdot 10^4$	**3.7** $\cdot 10^7$	**4.0** $\cdot 10^6$	**3.7** $\cdot 10^6$	**2.0** $\cdot 10^2$	**4.4** $\cdot 10^5$	**2.4** $\cdot 10^4$	**2.5** $\cdot 10^6$
	el-a-ts	**2.1** $\cdot 10^4$	**3.4** $\cdot 10^7$	**2.9** $\cdot 10^6$	**2.7** $\cdot 10^6$	**2.2** $\cdot 10^2$	**3.9** $\cdot 10^5$	**2.1** $\cdot 10^4$	**2.1** $\cdot 10^6$
	el-arw	**2.1** $\cdot 10^4$	**1.1** $\cdot 10^7$	**3.2** $\cdot 10^6$	**3.4** $\cdot 10^6$	**1.9** $\cdot 10^2$	**0.5** $\cdot 10^5$	**0.8** $\cdot 10^4$	**1.8** $\cdot 10^6$
	el-arw-ts	**2.5** $\cdot 10^4$	**0.8** $\cdot 10^7$	**2.3** $\cdot 10^6$	**2.7** $\cdot 10^6$	**2.5** $\cdot 10^2$	**0.5** $\cdot 10^5$	**0.8** $\cdot 10^4$	**1.5** $\cdot 10^6$

For the experiments, we used two different environments. The first environment, which we will refer to as *x5355-64GB*, consists of a single machine with 4-core Intel Xeon X5355 2.66 GHz CPU and 64 GB of memory, running Linux with the 3.2.0 kernel. The second environment, which we will refer to as *x3450-8GB*, is a cluster containing three identical nodes with 4-core Intel Xeon X3450 2.66 GHz CPUs and 8 GB of memory, running Linux with the 2.6.26 kernel. As all of the CPUs which we used support Hyper-threading, up to 8 threads may run seemingly simultaneously on any of these machines. To achieve maximal concurrency, all of the benchmarks were configured to use 8 threads. For lightweight monitoring, programs were compiled with `-g` and `-O3` flags.

3.1 Comparison of Lightweight Monitoring Approaches

First, we evaluate the impact of the different variants of lightweight monitoring that we proposed on the behaviour of the monitored programs. As a metric, we use the global number of transactions aborted during the program run. The presented experiments were performed in the *x5355-64GB* environment.

Table 1 shows the average global number of aborts (out of 100 runs) for each of the tested programs when executed with the different variants of lightweight monitoring described in Section 2.2. The variant *orig* represents a run without any monitoring, i.e., the execution of the original program with no modifications. The parameters of each of the programs were set to the values recommended for the so-called standard runs of the programs in the STAMP benchmark suite[7].

When performing the most lightweight monitoring (*sc*), the global number of aborts does not change much and stays almost always within a range of 5 % from the original runs. The only exception is the `ssca2` benchmark which gets near 35 % more aborts than in the original runs. This is caused by the so-called outliers, i.e., rare runs that achieve a number of aborts much higher than usual, which distorts the results. This

[7] These parameters are recommended by the STAMP authors when running the benchmarks natively, i.e., directly on a concrete operating system, not in a simulator or another tool negatively affecting its performance.

Table 2. Average aborts in original runs and runs with lightweight monitoring without outliers.

		genome	intruder	kmeans		ssca2	vacation		yada
	variant			high	low		high	low	
Lightweight	orig	**2.6** $\cdot 10^4$	**4.3** $\cdot 10^7$	**5.6** $\cdot 10^6$	**5.0** $\cdot 10^6$	**2.6** $\cdot 10^2$	**4.9** $\cdot 10^5$	**2.5** $\cdot 10^4$	**2.6** $\cdot 10^6$
	sc	**2.7** $\cdot 10^4$	**4.4** $\cdot 10^7$	**5.4** $\cdot 10^6$	**5.0** $\cdot 10^6$	**2.5** $\cdot 10^2$	**4.9** $\cdot 10^5$	**2.6** $\cdot 10^4$	**2.6** $\cdot 10^6$
	el	**2.2** $\cdot 10^4$	**3.8** $\cdot 10^7$	**4.2** $\cdot 10^6$	**3.9** $\cdot 10^6$	**1.7** $\cdot 10^2$	**4.6** $\cdot 10^5$	**2.5** $\cdot 10^4$	**2.6** $\cdot 10^6$
	el-ts	**2.1** $\cdot 10^4$	**3.5** $\cdot 10^7$	**3.7** $\cdot 10^6$	**3.3** $\cdot 10^6$	**1.6** $\cdot 10^2$	**4.3** $\cdot 10^5$	**2.4** $\cdot 10^4$	**2.3** $\cdot 10^6$
	el-a	**2.3** $\cdot 10^4$	**3.7** $\cdot 10^7$	**3.9** $\cdot 10^6$	**3.6** $\cdot 10^6$	**1.9** $\cdot 10^2$	**4.4** $\cdot 10^5$	**2.4** $\cdot 10^4$	**2.5** $\cdot 10^6$
	el-a-ts	**2.1** $\cdot 10^4$	**3.4** $\cdot 10^7$	**2.9** $\cdot 10^6$	**2.6** $\cdot 10^6$	**1.6** $\cdot 10^2$	**3.9** $\cdot 10^5$	**2.1** $\cdot 10^4$	**2.1** $\cdot 10^6$
	el-arw	**2.1** $\cdot 10^4$	**1.1** $\cdot 10^7$	**3.2** $\cdot 10^6$	**3.2** $\cdot 10^6$	**1.8** $\cdot 10^2$	**0.5** $\cdot 10^5$	**0.8** $\cdot 10^4$	**1.8** $\cdot 10^6$
	el-arw-ts	**2.4** $\cdot 10^4$	**0.9** $\cdot 10^7$	**2.3** $\cdot 10^6$	**2.6** $\cdot 10^6$	**1.7** $\cdot 10^2$	**0.5** $\cdot 10^5$	**0.8** $\cdot 10^4$	**1.5** $\cdot 10^6$

effect is more noticeable in the cases where the global number of aborts is relatively low and even one of such outlying runs may change the average values considerably. For example, the results for the `ssca2` benchmark using the *sc* monitoring approach contained two runs with 4300 and 3800 global numbers of aborts. When we look at the global number of aborts and remove the 10 runs identified as outliers, we get close to the original global number of aborts even for the `ssca2` benchmark. These results can be seen in Table 2. In particular, we take as outliers the runs which achieved a significantly different global number of aborts than the rest of the runs based on their Euclidian distance from the 10 runs with the closest global number of aborts.

When we try to obtain the same information as above using the event logger approach (*el*), we see that the global number of aborts drops much more than when using the *sc* approach—changing up to 25 % of the original value. This is because logging the events in a list is more intrusive than just incrementing a counter. This demonstrates that it is indeed quite important how the monitored information is acquired and registered as even slightly different methods that obtain the same information may have considerably different impact on the behaviour of the monitored TM programs.

When we start collecting more information (events) than just the number of started and committed transactions, we get an even lower global number of aborts. When logging the number of aborts as well (using the *el-a* approach), the drop in the number of aborts is not that significant yet (up to 30 % of the original value) as the number of events of this type is not that high. However, when we start tracking the read and write operations as well (using the *el-arw* approach), the global number of aborts often suffers large drops (the change is up to 90 % of the original value). This is related to the fact that the number of reads and writes is usually much higher than the number of starts and commits.

If we also start collecting the time stamps (using the *el-ts*, *el-a-ts*, and *el-arw-ts* approaches), the global number of aborts does also drop when compared with the variants not collecting the time stamps. However, in general, despite collecting time stamps is usually more intrusive than tracking the aborts, it is less intrusive than tracking the reads and writes.

Table 3. A comparison of average number of aborts for lightweight and heavyweight monitoring.

		genome	intruder	kmeans		ssca2	vacation		yada
	variant			high	low		high	low	
Lightweight	orig	67.6	22850.0	3804.7	1626.1	6.5	23.4	4.9	9362.3
	sc	73.3	22013.1	4115.7	1721.5	7.2	23.3	5.3	11659.3
	el	63.1	17663.5	2722.9	1245.9	12.2	25.2	5.3	9354.7
	el-ts	61.3	16797.2	2402.7	1236.4	13.0	22.6	4.7	8118.7
	el-a	65.8	16504.1	2204.3	1091.0	16.6	22.6	4.0	8096.3
	el-a-ts	64.3	16112.9	1696.8	942.8	15.6	19.7	3.8	6846.7
	el-arw	72.7	8238.9	2891.2	1877.0	18.0	19.9	3.7	5804.0
	el-arw-ts	107.1	9499.4	3463.6	2121.3	22.0	22.6	4.7	4458.0
PIN	orig	3.7	85.8	0.2	0.1	0.0	2.1	0.2	595.1
	sc	3.4	81.1	0.4	0.1	0.0	2.0	0.3	584.4
	el	8.6	92.2	7.2	6.7	0.5	2.4	0.5	589.3
	el-ts	9.4	106.9	9.0	7.8	0.7	2.5	0.3	571.2
	el-a	7.0	101.6	14.9	12.2	0.5	2.1	0.2	580.2
	el-a-ts	7.4	95.7	17.5	14.6	0.6	2.4	0.3	576.6
	el-arw	13.2	476.8	36.6	28.6	0.9	10.1	1.6	715.2
	el-arw-ts	24.1	1567.1	213.2	139.3	1.0	14.6	2.8	902.4
ANaConDA	orig	10.8	71.4	0.3	0.1	0.0	1.9	0.2	595.6
	sc	9.3	109.8	0.2	0.1	0.0	3.4	0.6	729.6
	el	13.7	109.7	8.6	7.8	0.6	4.0	0.5	704.3
	el-ts	11.3	119.2	9.8	8.6	0.8	4.0	0.4	687.4
	el-a	12.3	126.0	20.8	16.7	0.9	3.6	0.7	702.4
	el-a-ts	11.0	133.8	24.5	18.0	0.9	4.0	0.5	682.3
	el-arw	20.8	1653.4	178.5	126.9	1.3	17.4	2.8	1100.1
	el-arw-ts	34.4	3132.9	480.8	305.8	1.5	19.1	3.7	1260.8

3.2 Comparison of Lightweight and Heavyweight Monitoring

In this section, we compare the impact of the lightweight and heavyweight implementations of the considered monitoring approaches. Since heavyweight monitoring greatly slows down the tested programs, for these experiments the parameters of the benchmarking programs were set to the values recommended by the STAMP authors for the so-called simulation runs, which are suitable when executing a program in a simulator or another tool that negatively affects its performance. Since the simulation runs generate much less aborts than the standard ones, meaning that the results might be negatively influenced by the outliers, we remove 10 (out of 100) runs marked as the outliers during the evaluation. Due to the higher time cost of these tests, the experiments were performed in the *x3450-8GB* environment.

Table 3 shows the average global number of aborts for each of the tested programs for the lightweight and heavyweight implementations of the monitoring approaches described in Section 2.2. The heavyweight implementations come in two different versions. The first version, called *PIN*, does the monitoring by executing the lightweight monitoring implementation, i.e., the modified versions of the programs, in the PIN framework without doing any instrumentation of the program. The purpose of this version is to show how the use of PIN's low-level virtual machine changes the behaviour of the monitored program even without the influence of the instrumentation needed to capture the monitored events. The second version, denoted as *ANaConDA*, is the true heavyweight implementation where the counter incrementation and event collection is done through the callbacks provided by the extended ANaConDA framework.

First of all, let us note that compared with the results of the standard runs (Table 2), the results of the simulation runs exhibit the same tendencies when monitored using the lightweight approaches (and hence we can consider their use instead of the standard runs meaningful). The main difference is that the simulation runs are more prone to problems with outliers as their execution time is quite short and even a very short disruption during the execution may change significantly the overall results. For example, the results obtained for the `yada` benchmark using the *sc* monitoring approach contain several runs with significantly greater global number of aborts even after the 10 outliers have been removed (in fact, in this batch of runs there were 14 runs with a very high global number of aborts).

When we start monitoring the programs using the heavyweight versions of the monitoring approaches, we can see a massive drop in the global number of aborts (more than 95 %). This drop is mainly caused by PIN's low-level virtual machine as just running the original (non-modified) version (*orig*) of a program in PIN leads to an extreme drop in the global number of aborts (more than 95 %). The additional disruption introduced by the monitoring code does not influence much the behaviour. In fact, rather than having the effect of decreasing the global number of aborts, like in the case of the lightweight monitoring, inserting the monitoring code actually helps to increase the number of aborts a little in the heavyweight monitoring. This effect increases as we collect more information while monitoring, which is a completely opposite tendency compared to the lightweight monitoring. Also, the monitoring code inserted by ANaConDA has a greater effect on increasing the global number of aborts than using the lightweight monitoring code executed in PIN.

Another effect that the heavyweight monitoring has on the considered programs is that it suppresses the outliers. Table 3 contains the results evaluated from the runs not marked as outliers, but the results are nearly identical even when considering all of the runs.

3.3 Impact of the Monitoring on Different Types of Transactions

The global number of aborts is an important performance metric and hence also a good basic metric of how the behaviour of the monitored programs is influenced by the monitoring layer. However, one may want to get a more detailed information about the behaviour of a program and also about the way how it is influenced by monitoring. To go one step further in this direction, we now consider monitoring numbers of aborts of different types of transactions and the influence of monitoring on these numbers. Since TM libraries do not give us statistics for different types of transactions, we use the information obtained using the *sc* monitoring approach as a baseline behaviour of a program in this case. As the global number of aborts when using the *sc* monitoring approach is very similar to the original global number of aborts, we may safely assume

Table 4. Average number of aborts for different types of transactions.

		intruder			kmeans-high		
	variant	Tx1	Tx2	Tx3	Tx4	Tx5	Tx6
Lightweight	sc	**13.9** $\cdot 10^6$	**91.0** $\cdot 10^5$	**20.5** $\cdot 10^6$	**51.7** $\cdot 10^5$	**24.9** $\cdot 10^4$	**51.0** $\cdot 10^0$
	el	**9.5** $\cdot 10^6$	**85.2** $\cdot 10^5$	**19.9** $\cdot 10^6$	**40.9** $\cdot 10^5$	**22.1** $\cdot 10^4$	**44.0** $\cdot 10^0$
	el-ts	**8.1** $\cdot 10^6$	**83.5** $\cdot 10^5$	**18.9** $\cdot 10^6$	**35.1** $\cdot 10^5$	**21.8** $\cdot 10^4$	**36.0** $\cdot 10^0$
	el-a	**9.5** $\cdot 10^6$	**86.0** $\cdot 10^5$	**19.0** $\cdot 10^6$	**37.8** $\cdot 10^5$	**21.9** $\cdot 10^4$	**37.0** $\cdot 10^0$
	el-a-ts	**8.7** $\cdot 10^6$	**83.0** $\cdot 10^5$	**17.0** $\cdot 10^6$	**26.8** $\cdot 10^5$	**22.2** $\cdot 10^4$	**33.0** $\cdot 10^0$
	el-arw	**5.1** $\cdot 10^6$	**23.6** $\cdot 10^5$	**3.3** $\cdot 10^6$	**31.3** $\cdot 10^5$	**8.3** $\cdot 10^4$	**12.0** $\cdot 10^0$
	el-arw-ts	**5.1** $\cdot 10^6$	**22.3** $\cdot 10^5$	**1.1** $\cdot 10^6$	**22.6** $\cdot 10^5$	**7.7** $\cdot 10^4$	**11.0** $\cdot 10^0$

that this behaviour is very close to the original one. The presented experiments were again performed in the *x5355-64GB* environment.

Table 4 shows the average number of aborts for each type of transactions present in the `intruder` and `kmeans` benchmarks (in the latter case, for the variant with high contention). As can be seen, the various kinds of monitoring influence each type of transactions differently. When looking at transactions of Type *Tx2* and *Tx3* for the `intruder` benchmark or at transactions of Type *Tx5* for the `kmeans` benchmark, one can see that utilizing the event logger with or without direct tracking of aborts (*el* and *el-a*, respectively) does not influence the average number of aborts much. The drop in the number of aborts is around 10 % here. Also, the collection of time stamps (the *el-ts* and *el-a-ts* approaches) changes these numbers minimally. However, when we start tracking the reads and writes (the *el-arw* approach), the number of aborts drops considerably (by around 65–85 %).

On the other hand, some types of transactions, like transactions of Type *Tx1* for the `intruder` benchmark and transactions of Type *Tx4* for the `kmeans` benchmark are more affected by the event logger (*el*) approach and exhibit a significant decrease in the number of aborts (by around 20–30 %). The number of aborts does not drop much when we add the direct tracking of aborts (*el-a*), but it lowers again (by around 10–20 %) when we include the collection of time stamps (the *el-ts* and *el-a-ts* approaches). When we start tracking the reads and writes in these types of transactions, the number of aborts drops again (by around 10–30 %), but this drop is not that significant as in the case of the previously described transaction types.

One may think that the abrupt drop in the number of aborts that we saw in the transactions of Type *Tx2*, *Tx3*, or *Tx5* when we started tracking the reads and writes is connected to the number of memory accesses in these types of transactions since the influence of the monitoring should be different for transactions with a high and low number of memory accesses, respectively. However, our analysis of the data showed no clear dependency between the number of accesses and the drops in the number of aborts. For example, transactions of Type *Tx2* perform on average 110 accesses to the TM, while transactions of Type *Tx3* just 3 and transactions of Type *Tx5* only 2. Still, the tendencies they exhibit for the various monitoring approaches are the same. The exact cause of this behaviour remains an interesting direction for future work.

Table 5. Average aborts in runs with lightweight monitoring in the *x3450-8GB* environment.

		genome	intruder	kmeans		ssca2	vacation		yada
	variant			high	low		high	low	
Lightweight	orig	**3.0** $\cdot 10^4$	**3.0** $\cdot 10^7$	**5.7** $\cdot 10^6$	**4.1** $\cdot 10^6$	**6.3** $\cdot 10^2$	**3.6** $\cdot 10^5$	**3.1** $\cdot 10^4$	**5.0** $\cdot 10^6$
	sc	**3.1** $\cdot 10^4$	**3.0** $\cdot 10^7$	**6.0** $\cdot 10^6$	**4.4** $\cdot 10^6$	**11.7** $\cdot 10^2$	**3.6** $\cdot 10^5$	**3.2** $\cdot 10^4$	**5.0** $\cdot 10^6$
	el	**2.7** $\cdot 10^4$	**2.9** $\cdot 10^7$	**4.9** $\cdot 10^6$	**3.7** $\cdot 10^6$	**3.4** $\cdot 10^2$	**3.4** $\cdot 10^5$	**3.0** $\cdot 10^4$	**4.6** $\cdot 10^6$
	el-ts	**2.6** $\cdot 10^4$	**2.9** $\cdot 10^7$	**4.5** $\cdot 10^6$	**3.3** $\cdot 10^6$	**1.9** $\cdot 10^2$	**3.3** $\cdot 10^5$	**2.8** $\cdot 10^4$	**4.4** $\cdot 10^6$
	el-a	**2.8** $\cdot 10^4$	**2.8** $\cdot 10^7$	**4.2** $\cdot 10^6$	**3.1** $\cdot 10^6$	**5.2** $\cdot 10^2$	**3.3** $\cdot 10^5$	**2.7** $\cdot 10^4$	**4.3** $\cdot 10^6$
	el-a-ts	**2.6** $\cdot 10^4$	**2.5** $\cdot 10^7$	**3.1** $\cdot 10^6$	**2.3** $\cdot 10^6$	**2.3** $\cdot 10^2$	**3.0** $\cdot 10^5$	**2.5** $\cdot 10^4$	**3.6** $\cdot 10^6$
	el-arw	**2.4** $\cdot 10^4$	**0.8** $\cdot 10^7$	**3.4** $\cdot 10^6$	**3.7** $\cdot 10^6$	**5.1** $\cdot 10^2$	*timeout*	**3.5** $\cdot 10^4$	**2.9** $\cdot 10^6$
	el-arw-ts	**2.8** $\cdot 10^4$	**0.7** $\cdot 10^7$	**2.5** $\cdot 10^6$	**2.2** $\cdot 10^6$	**2.4** $\cdot 10^2$	*timeout*	*timeout*	*timeout*

3.4 Influence of the Environment

In the previous sections, we discussed that even a slight disturbance of the monitored TM program's execution by the monitoring code could impact its behaviour. However, changes in the monitoring code are not the only factor that may influence the behaviour of the monitored program. Other factors include changes of the environment in which the monitoring is done. That is why we now compare both of our execution environments used for acquiring the experimental results.

In particular, Table 5 shows results of the same experiments with lightweight monitoring as Table 1 but this time from the *x3450-8GB* environment instead of *x5355-64GB*.[8] We can see that the tendencies for the various monitoring approaches are similar to the ones presented before. However, the average global number of aborts changed for some of the benchmarks. For example, the `intruder` benchmark achieved around 30 % less aborts on this machine regardless of the monitoring approach used. On the other hand, the `yada` benchmark got twice as many aborts with any monitoring approach used.

Moreover, interestingly, some of the benchmarks seem to behave the same way as on the previously used machine when looking at the global number of aborts only. However, when looking at aborts for different types of transactions, one finds out that the program is in fact behaving differently. When looking at the `kmeans` benchmark, the average global number of aborts for the original run (*orig*) is nearly the same, but this is not true when we compare the number of aborts per transactions type.

In particular, Table 6 contains the average number of aborts for each type of transactions present in the `intruder` and `kmeans` (high contention variant) benchmarks. When we look at the *sc* monitoring approach and compare transactions of Type *Tx4* and *Tx5* with the results presented in Table 4, we see that the number of aborts for transactions of Type *Tx4* increases by about 20 % while the number of aborts for transactions of Type *Tx5* drops by more than 85 %. Moreover, the tendencies exhibited by transactions of type *Tx5* change: now, the number of aborts starts actually increasing when

[8] The missing values for some of the benchmarks for the *el-arw* and *el-arw-ts* monitoring approaches in Table 5 are caused by all of the runs timing out due to the extensive swapping as the main memory was rapidly filled out with the collected events.

Table 6. Average aborts for different types of transactions in the *x3450-8GB* environment.

		intruder			kmeans-high		
	variant	Tx1	Tx2	Tx3	Tx4	Tx5	Tx6
Lightweight	sc	**3.2** $\cdot 10^6$	**88.9** $\cdot 10^5$	**17.5** $\cdot 10^6$	**59.8** $\cdot 10^5$	**3.6** $\cdot 10^4$	**6.0** $\cdot 10^0$
	el	**3.8** $\cdot 10^6$	**84.1** $\cdot 10^5$	**16.5** $\cdot 10^6$	**48.7** $\cdot 10^5$	**6.3** $\cdot 10^4$	**7.0** $\cdot 10^0$
	el-ts	**4.2** $\cdot 10^6$	**85.9** $\cdot 10^5$	**16.5** $\cdot 10^6$	**44.0** $\cdot 10^5$	**7.6** $\cdot 10^4$	**8.0** $\cdot 10^0$
	el-a	**3.9** $\cdot 10^6$	**85.9** $\cdot 10^5$	**15.4** $\cdot 10^6$	**41.0** $\cdot 10^5$	**6.4** $\cdot 10^4$	**7.0** $\cdot 10^0$
	el-a-ts	**4.0** $\cdot 10^6$	**83.9** $\cdot 10^5$	**13.1** $\cdot 10^6$	**29.9** $\cdot 10^5$	**7.7** $\cdot 10^4$	**8.0** $\cdot 10^0$
	el-arw	**3.7** $\cdot 10^6$	**15.3** $\cdot 10^5$	**2.3** $\cdot 10^6$	**33.4** $\cdot 10^5$	**6.9** $\cdot 10^4$	**7.0** $\cdot 10^0$
	el-arw-ts	**4.4** $\cdot 10^6$	**14.6** $\cdot 10^5$	**1.1** $\cdot 10^6$	**23.5** $\cdot 10^5$	**10.1** $\cdot 10^4$	**14.0** $\cdot 10^0$

more intrusive monitoring approaches are used. Also, the time stamp collection greatly increases the number of aborts here.

We see a similar change in the behaviour in the `intruder` benchmark for transactions of Type *Tx1*. While the other two types of transactions exhibit similar tendencies and number of aborts, the number of aborts in transactions of Type *Tx1* drops by more than 75 % when using the *sc* monitoring approach. Using the more intrusive monitoring approaches then increases the number of aborts.

4 Analysis of the Impact of Heavyweight Monitoring

It is hard to explain all the above presented changes in the behaviour of the monitored TM programs since, for that, one would typically need some additional information about their original behaviour. However, gathering such information is usually impossible without monitoring and hence without again changing the behaviour.

Nevertheless, the situation is a bit different for the specific case when one wants to analyse differences between what happens within lightweight and heavyweight monitoring. In this case, the environment used for heavyweight monitoring has more influence on the behaviour than the actual collection of information about the monitored program. Hence, one may come with a hypothesis why the behaviour changes in a certain way in heavyweight monitoring and then try to support the hypothesis by analysing differences of suitable data collected about the behaviour of the monitored program during lightweight and heavyweight monitoring processes. We follow this path below.

Our hypothesis why the behaviour of the monitored TM programs changes so significantly during heavyweight monitoring is as follows. The run-time environment used in heavyweight monitoring has to execute not only the code of the monitored program but also the monitoring code that collects desired information about the execution of the program as well as other essential code for managing the running threads, for determining when and where to execute the monitoring code, etc. As a result, there is more code to be executed inside each transaction block, but there is even more code to be executed outside of the transactions. This, of course, influences the timing of the transactions as their execution is moved further apart in the program's execution, and even though their execution is longer, their chances to overlap and possibly abort are

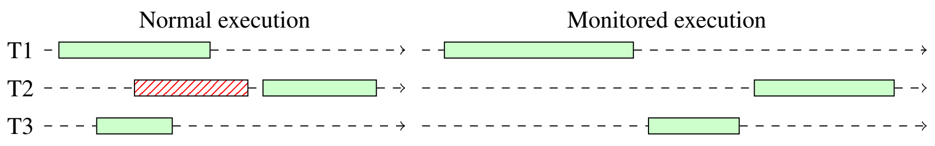

Fig. 1. Differences between normal and monitored execution.

Table 7. Average percentage of time spent in transactions.

		genome	intruder	kmeans		ssca2	vacation		yada
	variant			high	low		high	low	
Light	el-a-ts	45.4%	71.6%	33.1%	26.9%	50.8%	96.2%	95.4%	89.0%
Light	el-arw-ts	60.3%	95.3%	78.6%	75.0%	63.8%	99.0%	98.9%	97.2%
Heavy	el-a-ts	13.9%	15.6%	8.1%	6.3%	3.4%	29.7%	27.8%	56.3%
Heavy	el-arw-ts	24.9%	29.9%	22.7%	23.4%	5.0%	65.1%	61.7%	74.1%

decreased. This phenomenon is illustrated in Figure 1 (where an abort of a transaction within the normal execution is highlighted in red hatching).

To support the above hypothesis, we computed how much time is spent inside and outside the transactional blocks (using recorded timestamps of starts, aborts, and commits of transactions). The results are shown in Table 7. One can clearly see that the relative time spent inside transactions is much lower when using heavyweight monitoring than when using lightweight monitoring. This confirms our hypothesis and explains why we get significantly less aborts during heavyweight monitoring. Moreover, the table also shows that when we start registering transactional reads and writes, we spend more time in transactions, and, correspondingly, we also get more aborts (cf. Table 3).

5 Conclusion

We have presented several approaches of lightweight and heavyweight monitoring of TM programs. The proposed monitoring techniques are publicly available and can be used directly or serve as an inspiration for implementing other specialized monitors. We have also presented an experimental evaluation of the influence of these monitoring approaches on the number of aborts, both at the global level and for each type of transactions present in the monitored programs. Further, we have shown that not only the monitoring process influence the number of aborts, but also the environment in which the monitoring is performed has a great impact on the overall behaviour.

From our experiments we concluded that when using lightweight monitoring strategies, the more information we monitor the less aborts we usually get, both globally and per transaction type as well. However, one has to be careful of the role of outliers and of the fact that the number of aborts does not decrease in the same way across different types of transactions. Moreover, sometimes, the number of aborts can even increase when we increase the amount of monitoring. Such a behaviour is easily observed when the environment used causes a massive initial drop in the number of aborts. This is, in particular, visible when using environments for heavyweight monitoring.

In the future, it would be interesting to find analytical explanations for the various phenomena observed during the experiments reported in this paper. Such explanations could then perhaps be used as a basis for finding means for neutralizing the influence of the monitoring approaches on the monitored runs. Furthermore, one can use the developed monitoring layer as a basis for developing various dynamic analyses allowing one to detect errors in the monitored programs.

Acknowledgment

We would like to thank H. Pluháčková and B. Křena for the valuable discussions on the topic of this paper as well as for a help with statistical processing of the considered data.

References

1. C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford Transactional Applications for Multi-Processing. In *Proc. of IISWC'08*, 2008.
2. M. Castro, K. Georgiev, V. Marangozova-Martin, J.-F. Mehaut, L. G. Fernandes, and M. Santana. Analysis and Tracing of Applications Based on Software Transactional Memory on Multicore Architectures. In *Proc. of PDP'11*. IEEE CS, 2011.
3. J. Fiedor and T. Vojnar. ANaConDA: A Framework for Analysing Multi-threaded C/C++ Programs on the Binary Level. In *Proc. of RV'12*. LNCS 7687, Springer, 2012.
4. R. Guerraoui and M. Kapalka. *Principles of Transactional Memory*. Morgan and Claypool Publishers, 2010.
5. T. Harris, J. Larus, and R. Rajwar. *Transactional Memory, 2nd Edition*. Morgan and Claypool Publishers, 2010.
6. J. M. Lourenço, R. J. Dias, J. a. Luís, M. Rebelo, and V. Pessanha. Understanding the Behavior of Transactional Memory Applications. In *Proc. of PADTAD'09*. ACM, 2009.
7. C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proc. of PLDI'05*. ACM, 2005.