Understanding the Behavior of Transactional Memory Applications

João Lourenço Ricardo Dias João Luís Miguel Rebelo Vasco Pessanha
CITI — Departamento de Informática
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa, Portugal
{Joao.Lourenco, rjfd}@di.fct.unl.pt
{jecluis, miguelrebelo6, vascopessanha}@gmail.com

ABSTRACT

Transactional memory is a new trend in concurrency control that was boosted by the advent of multi-core processors and the near to come many-core processors. It promises the performance of finer grain with the simplicity of coarse grain threading. However, there is a clear absence of software development tools oriented to the transactional memory programming model, which is confirmed by the very small number of related scientific works published until now.

This paper describes ongoing work. We propose a very low overhead monitoring framework, developed specifically for monitoring TM computations, that collects the transactional events into a single log file, sorted in a global order. This framework is then used by a visualization tool to display different types of charts from two categories: statistical charts and thread-time space diagrams. These last diagrams are interactive, allowing to identify conflicting transactions. We use the visualization tool to analyse the behavior of two different, but similar, testing applications, illustrating how it can be used to better understand the behavior of these transactional memory applications.

Categories and Subject Descriptors

D.1.3 [**PROGRAMMING TECHNIQUES**]: Concurrent Programming—*Parallel Programming*; D.2.5 [**SOFTWARE ENGINEERING**]: Testing and Debugging—*Diagnostics*

General Terms

Algorithms, Performance, Reliability, Experimentation

Keywords

Software Transactional Memory, Monitoring, Profiling, Visualization, Testing, Debugging, Concurrency

1. INTRODUCTION

The interest in parallel programming was boosted by the recent emergence of multi-core processors. In the past, performance improvement had a strong dependency on processor speed increase, but processor speed is not increasing anymore. The recent evident drop of prices and general availability of multiprocessors in desktop computers made these multi-core architectures available not only to the everyday user, but also to the software developers, who must now rely on parallelism to fully exploit computational systems and achieve performance improvements. One can estimate that soon desktop computers will include dozens of processors and, thus, the programming mechanisms and methodologies must consider scalability as a key issue. Transactional Memory (TM) promises to ease the development of scalable parallel applications with performance close to finer grain threading but with the simplicity of coarse grain threading.

Parallelism comes to application development at the expense of a dramatic increase in the program complexity and in the development efforts. Coding is harder due to many factors, such as tracking and coordinating the multiple concurrent control flows. Testing is also harder, as the parallel application may exhibit a multitude of behaviors, many of them unacceptable. Debugging is also much harder, as the exponential number of possible application states makes state-based debugging *per se* almost useless, and the intrusion effect introduced by monitoring (logging) approaches may change the application behavior and potentially masks errors previously observed and trigger new ones. Also, developers observe that parallel applications underperform for the available hardware. This is frequently due to design and/or coding decisions that limit the exploitation of concurrency internally by the application.

The increased complexity of parallel program development at all levels, including testing and correction and performance debugging, may be eased up by a good understanding of the effective application behavior in its specific hardware and software execution contexts, including understanding the transactional framework being used to model and control interactions between the multiple control flows. One way to achieve such an understanding is by collecting run-time information about the application behavior and later analyze this data. The collected run-time raw data can easily achieve hundreds of megabytes and, thus, become unmanageable by the common developer. A visual representation of the collected run-time data may aggregate large amounts of data in a single figure and, thus, may be very convenient for the program behavior analysis.

In this paper we propose a framework for analyzing the behavior of Transactional memory applications. This framework is composed by four components: the low overhead monitoring tool and trace file generator, the trace-file processor, the trace-file analyzers, and the graphical user interface. Each of these components will be further detailed in this paper.

The main contributions of this paper are:

- The proposal of a low overhead monitoring system for transactional memory programs that does not change

the global application behavior;

- A set of analyzers that extract relevant information from the trace file;
- An interactive graphical user interface that displays the information produced by the analyzers.

The remaining of this paper is organized as follows: the next Section will introduce the low overhead monitoring framework for transactional memory; Section 3 will describe the experimental context where the monitoring framework was used; Section 4 will describe our tool that displays a set of charts reporting on the information collected by the monitoring framework; Section 5 will show how the tool can be used to help in analyzing the behavior of two testing applications; and Section 7 presents some concluding remarks and line out some future work.

2. THE MONITORING FRAMEWORK

Monitoring transactional memory requires registering the *Start* and ending of a transaction, either with *Commit* or *Abort*, and all *Read* and *Write* accesses to shared memory locations that took place within the transaction. Reading or writing data from/to a memory location is usually accomplished with a single machine instruction. Logging these memory access events will probably require dozens or even hundreds of machine instructions, speeding down the memory accesses operations by one or two orders of magnitude. This level of overhead may be unacceptable to the computation. Disks are much slower than memory and saving the logged events into a file is also unacceptable in most situations. As an alternative, a limited number of events may be kept in a shared memory buffer, but the need to have exclusive access to the shared buffer for registering the events makes it a bottleneck in the logging system, eliminating much of the non-determinism inherent to the parallel computation and significantly changing the application behavior.

Three important properties must be considered when developing a transactional memory monitoring system: i) have the logged events kept in main memory represented with a small memory footprint; ii) do not introduce additional synchronizations between threads; and iii) do keep the global application behavior. The last property depends on its predecessor, as additional synchronizations between threads will most probably change the global program behavior.

To satisfy these three properties, we opted for an approach where each thread keeps the logging information in a private buffer in a compact binary format. All threads are thus working independently from each other, allowing the concurrent registration of events with no contention between threads. When the program finishes its execution, the tracing system merges all buffers and dumps the events into a single file, in text format for easier understanding. Merging the local thread buffers depends on defining a global order for the events. One possible solution would be to have an atomic counter incremented by each thread each time an event is registered. However, this approach would not comply to the second and third properties, which states that the logging system should not introduce additional synchronization requirements neither change the global application behavior. Our solution was to use a specific CPU register (the RDTS register) that gives the number of clock cycles since the last system reset. The value given by this register can be used to impose a global order to the events and can be accessed by all threads with no additional synchronization.

The value of the RDTSC register in each processor that may drift from the others. This clock drifting causes the global ordering of the events to be error prone and, thus, the resulting single file is not 100% accurate. However, the inaccuracy of this methodology does not compromise seriously the results for the statistical information; and this methodology provides very accurate information that could not be obtained otherwise, such as the real transaction duration time.

When using the tracing system, if all the operations being traced incur in the same overhead, the global application behaviour will be essentially the same, including the inherent non-determinism of the application. There will be a simple reduction of the overall system performance without significant impacts in the system behavior.

Figure 1 illustrates the performance of the two testing applications with and without the monitoring system activated. We studied the behavior in a read dominant context (the red/dark gray line), and a write dominant context (the green/light gray line). The left column always refers to the Linked List application and the right column to the Red-Black Tree application. For the top line, the testing applications were ran with no monitoring system. For the middle line, we used a (shared) atomic counter as a logical clock to timestamp the events and support their global ordering. For the bottom line, we used the CPU registers (RDTSC) to timestamp the events.

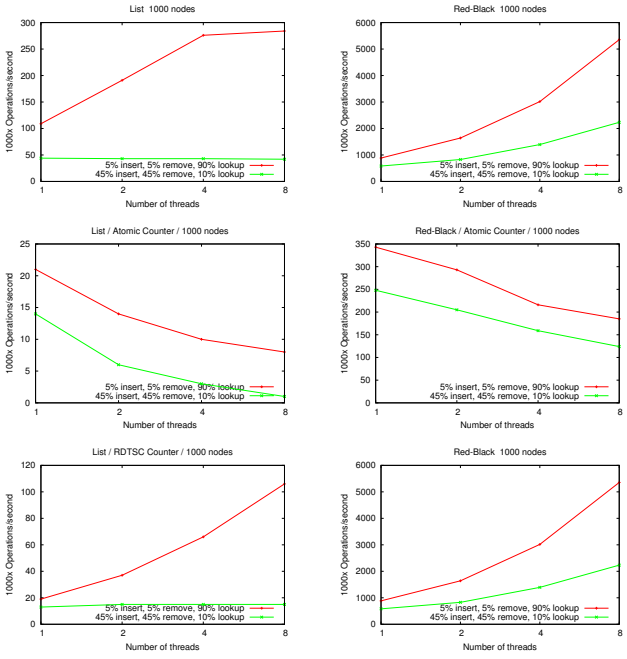


Figure 1: The performance of testing applications with and without the monitoring system.

By analysing the figure, one can conclude that the performance results for the applications without monitoring (top line) and with the atomic counter (middle line) are completely different, thus the applications exhibit different be-

haviors. On the other hand, comparing the graphs in the top and bottom lines, one can depict that the monitoring system in our approach reduces the overall performance to approximately 40% of the original, but displays similar scalability when the number of processors increase, keeping the global behavior for both applications. In the top-left and bottom-left charts, the performance curve for the read-dominant context are slightly different. This is due to scalability limitations of the testing application (List) when running without monitoring. This limitations are not triggered when monitoring is activated because the total number of operations per second is much lower (approximately 40%).

2.1 Event Types

There are many differences between the multiple transactional memory frameworks described in the literature. However, all of them rely in a small set of operations to provide its functionalities, namely: *Start* of a transaction (`Tx_Start`), end a transaction successfully with *Commit* (`TxCommit`) or unsuccessfully with *Abort* `TxAbort`, and access a shared data item for *Reading* (`TxRead`) or for *Writing* (`TxWrite`). Although with multiple alternative implementations, the above set of events is widely accepted as the minimum set necessary to describe Transactional Memory computations.

The programmer will use these operations in the following order: `TxStart (TxRead | TxWrite)* [TxAbort] TxCommit`, where "`*`" denotes repetition and "`[]`" denotes optional. However, considering that at runtime *TxAbort* and *TxCommit* are mutually exclusive and that a transaction may abort by many reasons; the actual behavior of the application can be represented as: `TxStart (TxRead | TxWrite)* (TxCommit | TxAbortUser | TxAbortCommit | TxAbortOther)`, where `TxAbortUser` denotes that the transaction was aborted by programmer request, `TxAbortCommit` denotes that the transaction tried but was unable to commit, and `TxAbortOther` denotes that the transaction aborted when accessing a shared data item, either for reading or for writing.

Each event must be registered upon the execution of the associated operation. In fact, we specify that all the events, except the *TxCommit*, must be registered right before the execution of the operation. The *TxCommit* event must be registered only when the transactional framework knows that it can commit the transaction. If a transaction willing to commit is aborted by the transactional framework, only a *TxAbort* event should be registered. This means that all transactions are delimited in the trace log by a *TxStart* event, and by either a *TxCommit* or a *TxAbort* event.

2.2 Event Structure

Each event is composed by a set of attributes. A subset of these attributes are common to all types of events while others are event specific. The following attributes are common to all events:

- `timestamp` — The time instant in which the event occurred;
- `eventId` — The identifier for the type of event, e.g., *TxStart*, *TxRead*, etc;
- `threadId` — The identifier of the thread that executed the operation;
- `transactionId` — The identifier of the transaction code block in which this operation took place.

All the attributes described above are self explanatory, except the last one. A single tread can execute multiple transactions in sequence, thus the `transactionId` attribute is used to identify the transaction code block where the operation took place. This allows to uniquely identify each transaction code block and to locate it in the source code. It also allows to map a set of operations into a single transaction.

The *TxAbort* event has an additional attribute, the `type` attribute, that is used to identify the reason for aborting the transaction. Because transactional memory frameworks implement different validation schemes, transactions can abort when preforming a read operation, a write operation, a commit operation, or when the user explicitly aborts the transaction. This attribute has three possible values: *commit*, when the transaction aborts on commit; *user*, when the transaction aborts by user request; *other*, when none of the previous apply and, thus, the transaction aborted in the sequence of a read or a write operation.

To keep the tracing system light, it should avoid post-processing the events online if it can be done later offline. There is no distinction between read and write aborts in the tracing file because it is easy to post-process the trace and, by looking back in time, to find which operation triggered the abort. In this case, one just needs to look back for the previous event from the same thread and check if it is either a read or a write operation.

The *TxRead* and *TxWrite* operations also have an additional attribute: the `varId` attribute. This attribute is used to identify the memory address or object ID (for an OO programming language) that was accessed by the operation. This identifier must be unique for each memory location or object.

2.3 Tracing System Instrumentation

We implemented a simple API so that TM frameworks could easily insert the tracing system call functions within the existing code. The current prototype only implements the API and the programmer must change the source code accordingly, but we intend to implement an mechanism to automatically insert the calls to the API. The API is composed by five functions, one to register each of the previously described events, namely *TxStart*, *TxCommit*, *TxAbort*, *TxRead*, and *TxWrite*.

All the calls to the monitoring API can requested by the TM framework. The exception goes to the *TxStart* event. This event must associate a unique ID to the transaction source code block that will latter be used to refer to that code block for, e.g., associate a transaction to a user-level operation. If IDs were generated automatically, a table that maps transactions IDs into source code blocks would have to be generated and made accessible to the programmer. A source-to-source compiler could easily generate the unique transaction IDs and dump this table in the end of the source code transformation process.

2.4 Tracing System Output

The tracing system dumps the content of all the buffers into a single file upon the termination of the application. All the events are ordered by increasing values of the timestamp attribute.

The format for each event in the trace file was defined aiming at allowing our analyzing tool to work with traces

generated by different TM runtime systems. This format definition is tus neutral to the TM and can be depicted in Figure 2, along with a small example of the output of a trace with two threads and two transactions.

3. EXPERIMENTAL CONTEXT

We performed a set of simple tests, logging the behavior with our monitoring system and then used our tool to analyze the behavior of these testing programs. The tests consisted on series of operations on a set. The set has two implementations, one as a Sorted Single Linked List and another as a Red Black Tree. The interface for both implementations provides three methods: *insert()*, *remove()* and *lookup()*. The set elements have a *key* and a *value* and all functions are indexed by the key. Duplicate keys are not allowed and adding an element with an already existing key will update its value. The tests were executed over CTL [3], a transactional memory framework for C and C++ programming languages derived from TL2 [4].

The tests are divided into three main categories, with three different load patterns. The test load pattern is defined by assigning different probabilities to each of the three methods. The first load pattern is meant to simulate a read dominant context, with 5% of inserts, 5% of removes and 90% of lookups. The second load pattern is meant to simulate a balanced system, with 20% of inserts, 20% of removes, and 60% of lookups. The third load pattern is meant to simulate a write dominant context, with 45% of inserts, 45% of removes, and 10% of lookups.

The tests were performed on a Sun Fire X4600 M2 x64 server with eight dual-core AMD Opteron Model 8220 processors @ 2.8 GHz with 1024 KB cache and 16 GB of RAM. The tests were executed to a maximum of 8 threads, to avoid having the operating system tasks interfere with the tests and introducing noise (arbitrary time delays) into the tracing system.

4. THE VISUALIZATION TOOL

One of the problems of tracing systems is that they tend to generate huge amounts of hard to digest information. In our monitoring system, besides registering the start and end of memory transactions, we must also register all the accesses to shared memory locations, and there may be millions of memory accesses per second. For the applications under testing, the monitoring system was generating approximately 50 Kbytes of tracing data per processor per millisecond, resulting in near 100 MByte of data for a time slot of 200 milliseconds with eight parallel threads.

Such volumes of information are clearly unmanageable with no aid from helping tools.

Our goal was to develop a tool to help processing the huge amounts of information generated by the monitoring system. The tool should serve two main purposes: i) provide a graphical representation for statistical information of the testing application behavior; and ii) provide a graphical representation of the application behavior along time.

The visualization tool we developed has thus two main features: visualization of statistical information by means of charts; and visualization of transactional computations in a timeline. Both features consume the same source of information which is the trace log generated by the tracing system (see Figure 2).

4.1 Application Components

The application is composed by four components: the monitoring framework, which was already described; the trace-file processor; the trace-file analyzers; and the graphical user interface.

4.1.1 The Graphical User Interface

The application graphical user interface (GUI) is in a very preliminary phase. The current set of analyzing modules will also be expanded in the near future. The application GUI is composed by two panes: one pane, on the left, allows to choose the type of visualization; the other pane, on the right, will display the selected chart/graph. Figure 3 illustrates the main view of the visualization tool.

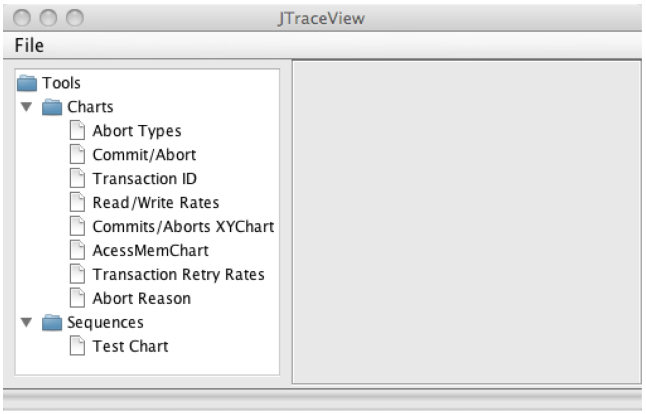


Figure 3: Main layout of the visualization tool

4.1.2 The Trace-File Processing

To ease the work of the analyzers when parsing the trace-file for generating charts or transaction behavior graphs, we developed a small component that allows to see the trace file as a list of events. Due to its size, we cannot load all the trace file data into main memory, thus this component provides the analyzers with an event iterator that operates over the events in the trace-file stored directly in secondary memory.

This iterator also supports the notion of *savepoint*. *Savepoints* work as bookmarks in the trace file and can be used to jump directly back and forth in the trace file without further processing.

4.1.3 The Trace-File Analyzers

There are two types of analyzers in our tool: visualization of statistical information by means of charts; and visualization of transactional computations in a timeline. The first one uses `JFreeChart`, a library to render many different types of charts (see Figure 5 as as example). The second one uses a new Java Swing component developed from scratch, to render the transactions behavior along time (see Figure 4 as an example).

Each analyzer must extend a well defined interface and must implement a method that returns the respective visual component. This component is rendered later on by the application GUI component. This approach allows analyzers to be considered as plugins to the GUI. All analyzers use

```
<timestamp> <eventId> T<threadId> <transactionId> [<TxAbort:type> | <varId>]

%% Example:
3043566053937770        tx_start  T1     2
3043566053938505        tx_read   T1     2       0x3871dbf8
3043566053938530        tx_start  T2     0
3043566053938569        tx_read   T1     2       0x805fa0
3043566053939240        tx_read   T2     0       0x805fa0
3043566053939378        tx_write  T2     0       0x805fa0
3043566053939505        tx_read   T1     2       0x3871dbf8
3043566053939725        tx_commit T2     0
3043566053940104        tx_abort  T1     2       commit
```

Figure 2: Event string format and output example.

the trace-file processor to extract the information needed to create the visual information.

The time-based transaction behavior analyzer is backed up by a Java Swing component that can show the executed transactions of each thread along time. Each transaction is represented by a color depending on the type of transaction and by the type of abort, and the size of the box representing the transaction is directly proportional to the real time duration of the transaction. This analyzer also allows the user to click in the abort event of a transaction A, and will automatically draw an arrow from that event to another transaction B that forced A to abort.

4.2 Visualization Modules

The visualization tool provides a graphical representations for statistical information of the application behavior as well as a graphical representation of transactional status of each application thread along time. These two main classes of charts will be further discussed in the following sections.

4.2.1 Statistical Information Charts

At the time of writing this paper, the visualization tool supports eight different charts for displaying statistical information:

Abort Types. Displays a pie chart with the relative (percentage) number of transactions aborted at different moments in the transaction life-cycle: by user request; when reading a memory cell/object; when writing a memory cell/object; and just prior to committing the transaction. Allows to have a feeling on the eagerness of conflict detection.

Commit/Abort. Displays the percentage of transactions that finished successfully versus those that had to abort. Allows to have a feeling on the amount of wasted computation cycles.

Transaction ID. Displays the relative number of each kind of user-level transactional operations. In our testing application, this will be the percentage of *insert()*, *remove()* and *lookup()* operations. Contributes to the understanding of the global application behavior.

Read/Write Rates. For each user-level transactional operation, displays a bar with the percentage of memory read and write operations. Contributes to the understanding of the behavior of the individual operations executed by the application.

Commits/Aborts XYChart. Reports on the number of committed and aborted transactions per execution time slice. Allows to infer the transactional throughput along time.

AccessMemChart. Reports on the access frequency for each transactional unit, e.g., how many times each memory cell was accessed. Allows to identify contention points. In the future we plan to split this chart into two charts, depending on the type of memory operation executed, i.e., a memory read or write.

Transaction Retry Rates. For each user-level transactional operation, reports average numbers for transaction retries. Allows to understand the level of contention exhibited by each user-level transactional operation.

Abort Reason. Reports on whether the aborts were caused by real conflicts or by false positives, i.e., the transaction was unnecessarily aborted by the transactional memory framework. Allows to understand if the contention management policies are adequate for the application under testing.

Wasted Work. Reports the percentage of time spent by aborted transactions in relation to the total time spent by all transactions. This contributes to the understanding of the time wasted in processing doomed transactions.

4.2.2 Time-based Behavior Information Charts

The visualization tool also supports the representation of the application behavior along time. This chart will represent in the Y-axis the multiple application threads, and in the X-axis the transactional status of those threads.

The example illustrated in Figure 4 refers to the evolution of the testing application with eight threads, with *Tx0* (dark blue) meaning the thread is executing an insert operation, *Tx1* (light blue) meaning the thread is executing a remove operation, and *Tx2* (yellow) meaning the thread is executing a lookup operation. Transactions terminate with either a commit (green) or abort (pink). In the bottom left there is a slider to change the zooming factor of the displayed information.

In opposition to the statistical visualization charts/modules, this time-based behavior information chart is interactive. If the user selects a time-slot in a transaction A corresponding to an abort, the module will locate and identify the transaction B that conflicted with transaction A and forced it to

abort. An arrow will be drawn connecting the abort time-slot of transaction A to the beginning of the time-slot of transaction B. This functionality can be depicted in Figure 4. The tool can also draw arrows from all abort events to the corresponding conflicting transactions. Behavioral patterns may be observed using this feature.

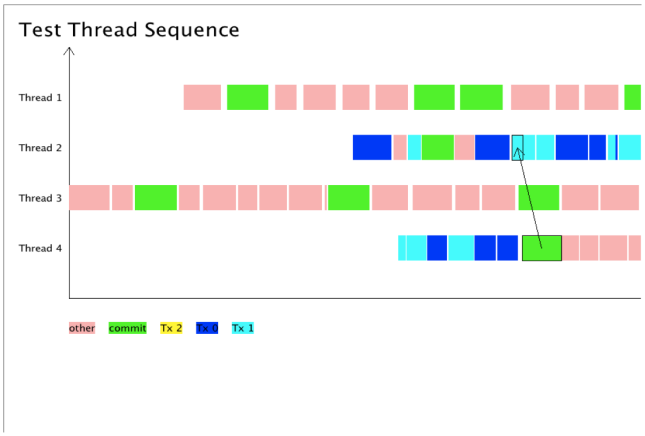


Figure 4: Example of a transaction conflict detection

In the future we plan to extend the user-interaction based functionalities, such as mapping the transaction time-slots to source-code locations and allowing the user to implicitly invoke the text editor in the source line associated with a specific time-slot.

4.2.3 Analyzer/Visualization Module Development

Developing a new analyzer is extremely easy in this tool. As described previously each type of analyzer corresponds to a Java class that extends an interface which defines the type of analyzer. This class implements the generation of the visual information, that can be a chart or a transaction behavior graph, by collecting information while iterating the trace-file. After implementing the analyzer class, it must be registered in the GUI component, in order to be listed in the left pane. These are the only two steps needed to create a new analyzer module.

In future work we will dynamically load the analyzers classes from a specified directory and list them in the GUI component without requiring an explicit registration. This will give the possibility to developers to create new analyzers even without having the tool source code.

5. APPLICATION BEHAVIOR ANALYSIS

In this section we will illustrate how the visualization charts can help understanding the behavior of an application that uses transactional memory. We recall that we actually have two similar testing applications implementing random-generated operations over a set. In each application the set resorts to different data structures: one uses a single linked list (LL), the other uses a red-black tree (RB). The syntax used to describe the testing conditions for the charts is as follows: `(App, n-threads, %inserts, %removes, %lookups, key_range)`. `App` will be either LL for the linked-list or RB for the red-black tree; `n-threads` should be between 1 and 8 and will identify how many threads were executing concurrently; `%inserts`, `%removes`, `%lookups` will identify the memory access pattern that can be read-dominant (5%, 5%, 90%), balanced (20%, 20%, 60%), or write-dominant (45%, 45%, 10%); `key_range` will define the size of the set.

5.1 Statistical Information Analysis

5.1.1 Commit/Abort Ratio

Figure 5 illustrates the Commit/Abort rates for different conditions of the Linked List testing application.

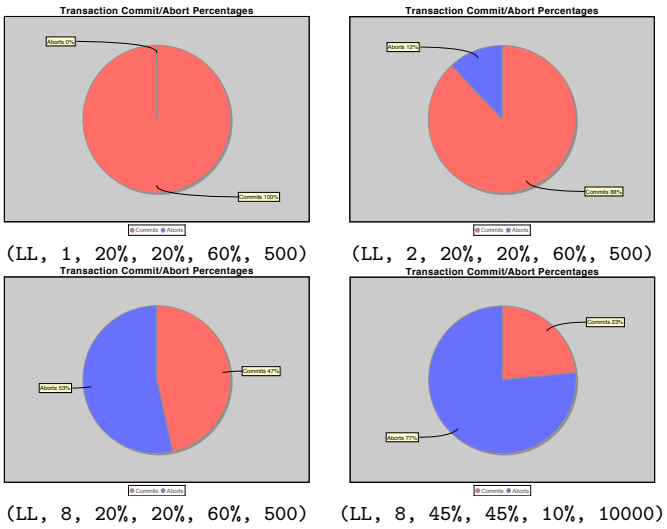


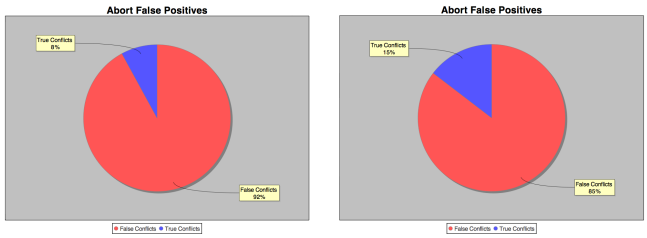
Figure 5: Commit abort rates for different testing application conditions.

As expected, with a single thread the abort rate is zero. This rate increases to 12% with two threads in a moderate update access pattern (40% of updates and 60% of lookups). With eight threads there is higher memory contention in accessing the list elements, and in the same balanced context the abort ratio increases to 53%. The worst case we could identify was with a very high rate of updates (90%) and with a very long list (10.000 elements). In this case, there is a very high probability that long transactions aiming at changing an element with a high key value, has to abort because it read a value that has been changed by a shorter transaction.

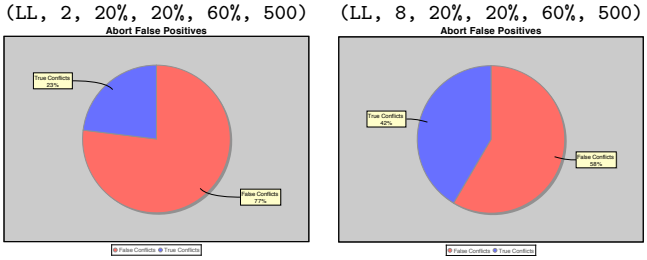
In the last two cases there is a considerable amount of wasted work done by aborted transactions. This means the transactional implementation of the underlying data structure is not adequate for current usage/parameters of the testing application.

5.1.2 False Positives

The TM framework used to generate the trace files was operating with memory words as the transactional unit and using a deferred update mechanism. Because each transaction may access thousands of memory cells, CTL maps memory addresses into a limited size table using a hash function. As multiple memory words may be mapped into the same position in the table, there is the chance to have undetected false conflicts. The size of this table has direct influence in percentage of the false conflicts.

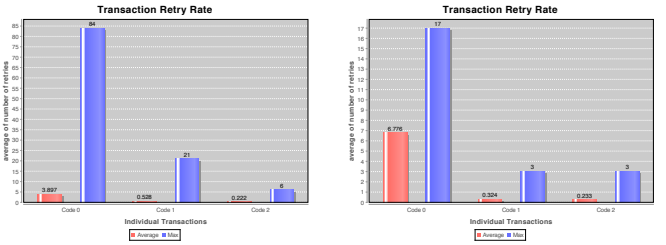


(LL, 2, 20%, 20%, 60%, 500) (LL, 8, 20%, 20%, 60%, 500)

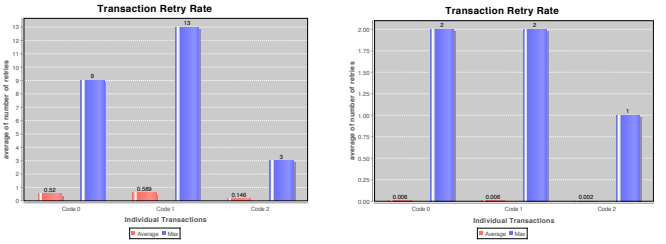


(RB, 8, 20%, 20%, 60%, 500) (RB, 8, 45%, 45%, 10%, 500)

Figure 6: Percentage of false positives for different testing application conditions.



(LL, 8, 20%, 20%, 60%, 50) (LL, 8, 20%, 20%, 60%, 10000)



(RB, 8, 45%, 45%, 10%, 50) (RB, 8, 45%, 45%, 10%, 10000)

Figure 7: Transaction retry rate for different testing application conditions.

Figure 6 shows the amount of false positives detected in different traces.

From the charts available, of which we present a subset here, we may infer that in general the list based (LL) solution has proportionally more false positives than the red-black tree based solution (RB). We may also infer that the number of false positives decrease when the contention level increases.

5.1.3 *Transaction Retry Rate*

Transactions frequently conflict with other transactions. The common approach to deal with conflicts is to abort one of the conflicting transactions. The transactional framework resorts to a contention manager to decide which of the conflicting transactions must abort. Depending on the contention manager policies, some types of transactions will be more prone to abort than others, e.g., the contention manager may give preference to shorter/longer transactions, or to younger/older transactions, or to read-only transactions, or to transactions that accessed the smaller/higher number of shared resources, etc.

Figure 7 illustrates the transaction retry rate for different testing application conditions. Different contention managers would originate different charts.

Finding an element in a list has an $O(n)$ complexity, while doing the same operation in a red-black tree has as $O(log_2 n)$. Thus, as expected, the transaction retry rate for the LL solution is much higher than for the RB solution. This can easily be depicted in the figure by comparing the top two charts with the lower ones (please note that the vertical scale differs in all the charts).

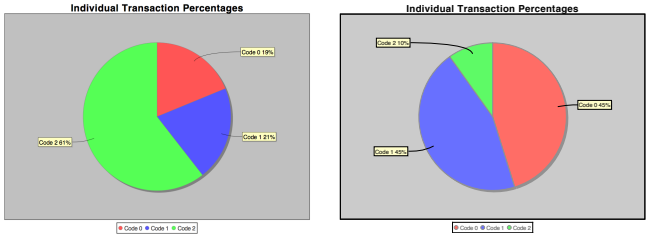
Another interesting effect, is that with a higher number of keys, the LL implementation has in average more aborts/retries and the RB implementation has less. This is due to the fact that the CTL contention manager does not privilege the longer, and the linear search in the LL implementation forces many long transactions to abort, because at commit time there is a good chance that a shorter transaction has updated an item that was previously read by the longer transaction and, thus, the longer transaction must abort.

The opposite applies to the RB implementation. The search space is split by half at each step and the probability of having two transactions in conflict is much lower. The contention level decreases when the set cardinality increases, as the search space is being slit even more.

5.1.4 *Application Level Operations*

Our testing applications receive a set of command line arguments (exactly the same, for both the LL and RB applications) that instantiate some of the configurable parameters, changing in this way the overall application behavior, including the memory access pattern.

In the case of our testing applications, there are only three high-level operations: `insert()`, `remove()` and `lookup()`. The Figure 8 indicates the relative frequency of those operations as registered in the trace file.



(LL, 8, 20%, 20%, 60%, 500) (LL, 8, 45%, 45%, 10%, 500)

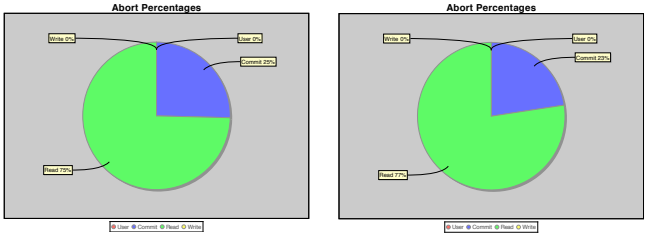
Figure 8: Relative frequency of user level operations in the application.

The chart in the right has a relative frequency exactly as expected for a write dominant context, i.e., 45% of inserts, 45% of removes and 10% of lookups. The chart in the left indicates 19% of inserts, 21% of removes and 61% of lookups, which even sum up to 101% and differs lightly from the values given in the command line, namely, 20%, 20% and 60% respectively. These small variations are due to the usage

of a pseudo-random number generator to select the operation to be executed and, on the other hand, to the rounding algorithm used to convert the percentages into integers.

5.1.5 Abort Types

The contention manager may order a transaction to abort when accessing a shared resource. If the transaction reaches the commit phase, a new validation phase is triggered, and the committing transaction read set is validated against the write set of all remaining concurrent transactions. The transaction write set must also be validated against the read and write sets of all concurrent transactions. Thus, four different situations may trigger the aborting of a transaction (whether the transaction is really aborted depends on the contention manager): reading a shared resource that has been changed since the current transaction has started; writing to a shared resource that has been changed since the current transaction has started; at the final validation of the read and write sets, just prior to committing the transaction; and by explicitly request from the user/programmer. Figure 9 represents visually this information.



(LL, 8, 20%, 20%, 60%, 500) (RB, 8, 20%, 20%, 60%, 500)

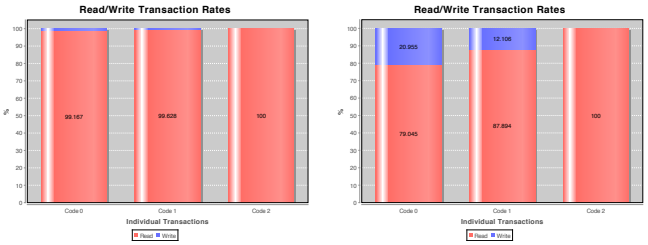
Figure 9: Relative frequency of abort types in the application.

In the case of our testing applications, the transactions are either read-only (lookup operations) or read-write. The later are a special case of read-write transactions, as the write operation is always done in the very final moments of the transaction life time. The LL version has more commit-time aborts than the RB version. This is due to the fact that LL transactions are very long and many conflicts will only be detected at commit time.

5.1.6 Read/Write Rates

Different application-level transactional operations exhibit different behavior in terms of memory access patterns. Some are read-only, some others are mixed read-write, and some others, as in the case of our testing applications, the write operations, when they exist, are in a small number (and take place in the very end of the transaction life-time). Figure 10 illustrates this ratio between read and write accesses to the shared memory cells.

From the figure it is possible to infer that the lookup operation (third column in both charts) does not update any memory location. It can also be depicted that the RB tree test does a lot more memory updates than the LL test. This is due to the internal re-balancing of the RB tree. In the case of the LL, the update operations (insert and remove) iterate the list nodes until the right node is found, and just then the node is updated and the transaction concluded.



(LL, 2, 20%, 20%, 60%, 500) (RB, 2, 20%, 20%, 60%, 500)

Figure 10: Relative frequency of memory accesses (read/write) within a transaction.

Thus, the proportion of reads/writes is much higher than in the case of the RB tree. Please note that both charts in the figure refer to tests with a range of only 50 different keys, which implies a list/tree with at most 50 different nodes. With larger lists/trees, the ratio of reads/writes will only increase.

6. RELATED WORK

As transactional memory is an emerging research area, few work has been done concerning tools to support the development of applications using transactional memory.

Yossi Lev in [7] presents a debugger which supports transactional memory. The work introduced new debugging mechanisms, shadowing the inner work of the transactional memory framework from the user.

Ansari in [1] presents a tool to profile the execution of applications that make use of transactional memory. The profiling tool was applied to non-trivial benchmarks, such as STAMP [2], to better understand what factors have more impact in the overall performance. Some of the statistical information provided by our tool has similar goals as those of Ansari's work.

Lourenço et al. in [8] presented some testing patterns that proved to be useful at testing and debugging transactional memory framework. Harmanci et al. in [6] developed a tool to help in design and optimization of transactional memory frameworks. The tool, TMUnit, provides a domain specific language for specifying workloads, and tests the performance and semantics of transactional memory frameworks.

The works reported in [1,7] and the works reported in [6,8] have different goals. The former aim at aiding the development of transactional memory applications, while the later aims at the development of transactional memory frameworks.

7. CONCLUDING REMARKS

Transactional memory is a new trend in concurrency control and there are not many tools available targeting software development using transactional memory. Not even in the research community. To our best knowledge up to the moment, all the TM-oriented known tools are reported in Section 6. In this paper we presented a novel tool aiming at helping software developers to understand the behavior of transactional memory applications.

Our tool resorts to a very low overhead monitoring framework, developed specifically for monitoring TM computations, that collects the transactional events (start of a transaction, commit, abort, read and write shared resources) in

each thread and logs them, together with a time-stamp, into a thread local memory buffer. When the application finishes, all the buffers are merged into a single one using a time-stamp to impose the global ordering.

The tool will use the global log to display different types of charts. Until now, all the charts may be grouped into one of two main categories, statistical and state-time diagrams. Statistical charts resort into analysis modules that process the contents of the log file aiming at displaying a specific kind of statistical information.

The global log may also be processed to extract time-relative information, such as which application-level operations are being executed concurrently, or how long did it take to execute a specific transaction. For our tool we developed and interactive module that exhibits a threads/time chart, with thread IDs in the Y-axis and time in the X-axis. The status of each thread changes visually along time. This module is interactive and the user may select an abort event in any thread and the tool will localize and point (with an arrow) the beginning of the conflicting transaction that forced the initial one to abort.

The tool is being actively developed and many other charts, both statistical and interactive, will may be developed. Other statistics that could be interesting to collect would be, for example, the effective amount of wasted work for each transaction type. Other interactive functionalities would include, for example, cross-referencing between the transactional operations in the log file and locations in the original source code.

We also plan to generate logs from other non-trivial TM benchmarking applications, such as STMBench7 [5], and the STAMP [2] and SPLASH2 [9] collections, and interpret the results using our visualization tool.

8. REFERENCES

[1] Mohammad Ansari, Kim Jarvis, Christos Kotselidis, Mikel Luján, Chris Kirkham, and Ian Watson. Profiling transactional memory applications. In *PDP '09: Proceedings of the 17th Euromicro International Conference on Parallel, Distributed, and Network-based Processing*. IEEE Computer Society Press, February 2009.

[2] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[3] Gonçalo Cunha. Consistent state software transactional memory. Master's thesis, Universidade Nova de Lisboa, November 2007.

[4] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. In *Distributed Computing*, volume 4167, pages 194–208. Springer Berlin / Heidelberg, October 2006.

[5] Rachid Guerraoui, Michal Kapalka, and Jan Vitek. Stmbench7: a benchmark for software transactional memory. *SIGOPS Oper. Syst. Rev.*, 41(3):315–324, 2007.

[6] Derin Harmanci, Pascal Felber, Vincent Gramoli, and Christof Fetzer. Tmunit: Testing transactional memories. In *4st ACM SIGPLAN Workshop on Transactional Computing (TRANSACT 2009)*, February 2009.

[7] Yossi Lev. Debugging with transactional memory. In *1st ACM SIGPLAN Workshop on Transactional Computing (TRANSACT 2006)*, June 2006.

[8] João Lourenço and Gonçalo Cunha. Testing patterns for software transactional memory engines. In *PADTAD '07: Proceedings of the 2007 ACM workshop on Parallel and distributed systems: testing and debugging*, pages 36–42, New York, NY, USA, 2007. ACM.

[9] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: characterization and methodological considerations. In *ISCA '95: Proceedings of the 22nd annual international symposium on Computer architecture*, pages 24–36, New York, NY, USA, 1995. ACM.