Trace Generation and Deterministic Execution for Concurrent Programs

Paulo S. L. Souza, Raphael N. Batista, Simone R. S. Souza, Rafael R. Prado, George G. M. Dourado, Julio C. Estrella

Institute of Mathematics and Computer Science – ICMC
University of Sao Paulo – USP
CEP 13.566-590 – São Carlos – SP – Brazil

{pssouza,rbatista,srocio,rafaelrp,georgemd,jcezar}@icmc.usp.br

Abstract. *This paper proposes new algorithms for generation of trace files and deterministic execution of concurrent programs under test. The proposed algorithms are essential to automate the coverage testing of concurrent programs and allow to execute new synchronizations automatically, increasing the source code coverage with focus on non-determinism, and edges of communication and synchronization. Our algorithms consider programs with multiple paradigms of communication and synchronization (collective, blocking and non-blocking point-to-point message passing, and shared memory). We validate our algorithms by means of experiments based on nine representative benchmarks, which exercise non-trivial aspects of synchronization found in real applications. Our algorithms have a robust behaviour and meet their objectives. We also highlight the overhead generated with the algorithms.*

1. Introduction

One of the most challenging issues while testing concurrent programs is dealing with the non-determinism resulting from the use of communication and synchronization[1] primitives and the consequent unpredictable progress of processes and threads. Multiple executions of a program using the same test case can exercise different pairs of synchronization being able to even generate different outputs, which are correct or not [Carver and Tai 1991]. This type of behaviour can hide defects, such as unprotected shared data, loss of messages or notifications, deadlocks, starvation, among others [Farchi et al. 2003]. Testing activity for concurrent programs is extremely challenging and prone to errors [Bocchino et al. 2009], as defects revealed in execution can disappear the next time: this makes the testing difficult and expensive.

Deterministic execution ensures that the same sequence of synchronization events is executed for the same test case. In order for deterministic execution to take place, sufficient information about the synchronizations must be extracted while the program is being executed and mechanisms must be created to enable the same synchronization sequence as the original execution to be replayed [Carver and Tai 1991]. *Trace files* store this information to keep the order in which the synchronization events were executed and also to determine which pairs of synchronization were carried out in a given execution. Besides re-executing, this information can be used to evaluate the coverage of synchronization

[1]For short, whenever possible, this paper uses the term synchronization to represent communication and synchronization events.

primitives [Souza et al. 2014]. Furthermore, it can be used as a basis to generate new synchronization pairs so that the coverage rate [Lei and Carver 2006] can be increased [Sarmanho et al. 2008].

This paper presents new algorithms that can generate trace files and execute deterministically, considering concurrent programs using multiple synchronization paradigms (blocking and non-blocking point-to-point message passing, collective messages and shared memory). The collective synchronization considered is one-to-all, all-to-one and all-to-all. Another significant difference of the proposed algorithms is that they were developed at the application level, i.e., they do not require alterations in their programming language or synchronization libraries.

The proposed algorithms were implemented and validated in a prototype using Java. The prototype was executed with nine benchmarks using different standards of synchronization. Our results demonstrate a stable behaviour of these algorithms.

2. Related Studies

Trace generation and deterministic execution of concurrent programs is a topic that has been addressed substantially in the literature. Some of the main studies found are presented as follows.

Carver e Tai [Carver and Tai 1991] proposed deterministic execution for semaphores and monitors. Their approach uses resources from the language itself to maintain the same order of the previous execution. During trace generation, only the order in which the threads gain access to a critical region (using monitor enter, semaphore down) is registered. During deterministic execution, threads verify if they can access the critical region, based on the original trace execution.

DejaVu (Deterministic Java Replay Utility) [Alpern et al. 2000] is a deterministic execution tool to facilitate the debugging of multi-thread programs executed in Jalapeño, a Java Virtual Machine (JVM) developed by IBM. Distributed DejaVu [Konuru et al. 2000] enables the trace file generation and replay for programs running in multiple JVMs. The tool uses a modified version of the JVM to obtain information on the Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) operations, carried out in Application Program Interface (API) Java sockets. These information allow the re-execution of message passing programs. The restriction of this approach is that the deterministic execution is specific to Java and depends on the Jalapeño modified virtual machine.

The Microsoft CHESS tool [Musuvathi et al. 2008] makes deterministic re-execution possible for Win32, .NET and Singularity programs. It uses a scheduler that observes all the calls made in the operating system's API, to obtain information on the execution and enable its replay. CHESS redirects all the calls from the synchronization operations to a library of wrappers, which captures sufficient information on the operational semantic and determines the order in which the events should occur.

Carver e Lei [Carver and Lei 2010] created a library called *Modern Multithreading* (MM) to develop, test and debug concurrent programs in C++ (using *Pthreads*) and Java. MM provides classes to create threads and synchronize objects, using both semaphores/monitors and synchronous/asynchronous message passing through channels. The non-deterministic testing executes the concurrent programs multiple times, expecting

different synchronizations. Random delays can also be used to increase the possibility of exercising different sequences of synchronization. Deterministic testing forces the execution of a synchronization sequence chosen by the tester. Reachability testing can explore possible pairs of synchronization in parallel.

Other studies propose the amount of information reduction necessary to generate trace files and reduce the overhead created by the deterministic execution [Lifflander et al. 2014] [Olszewski et al. 2009]. The main focus of these related studies is the shared memory primitives, although some of them address synchronous and asynchronous message passing. Another factor is that the proposals do not consider the use of simultaneous point-to-point message passing, collective and shared memory in the same concurrent program. In this same direction, other studies propose the deterministic execution in multithreaded systems without using trace files at all [Bergan et al. 2010] [Liu et al. 2011]. Some studies addressing both the synchronization paradigms need the programmer to change the source code [Carver and Lei 2010] or a specific JVM [Konuru et al. 2000]. Indeed, there are various proposals specific to a language, depending on the API of the operating system or changing the JVM. Our algorithms for trace file generation and deterministic execution are different from the others as they are broader, applied for coverage testing activity, offering solutions at the application level and consider various synchronization paradigms.

3. Algorithms for Trace Generation and Deterministic Execution

The algorithms developed operate at the user level, using synchronization resources provided by their own programming language. In order to do this, a static analysis of a concurrent program is carried out automatically to identify its synchronization primitives and afterwards the program is instrumented placing function calls before and after each primitive. The instrumentation transforms the program into a slightly modified program, semantically equivalent to the original, except for collecting information and mechanisms to control the program execution. The automatic static analysis and instrumentation use the *ValiInst* module from the *ValiPar* [Prado et al. 2015]. Due to our available space to write, this paper does not describe the particularities of the *ValiInst* module.

Algorithm 1 represents the modifications made for each primitive of the instrumented program. The algorithm can be executed freely (non-deterministically), as well as deterministically. For both of the executing ways, trace files are generated to identify the synchronization sequence and the synchronization pairs carried out during the execution of the concurrent program.

Algorithm 1 Instrumented primitive.

```
1: if execution_mode = DETERMINISTIC then
2:     CONTROLLER.VALIPAR_BEFORE(...)
3: end if
4: TRACER.VALIPAR_BEFORE(...)
5: PRIMITIVE(...)
6: TRACER.VALIPAR_AFTER(...)
7: if execution_mode = DETERMINISTIC then
8:     CONTROLLER.VALIPAR_AFTER(...)
9: end if
```

The two modalities of execution are divided into *before* and *after* the primitive execution. During the trace generation, the *tracer.valipar_before()* function collects infor-

mation from the sender and the *tracer.valipar_ after()* function retrieves both the information from the synchronization pair and the global order in which the event was executed.

In the deterministic execution, the *executionController.valipar_before()* function verifies if the primitive can be executed, basing itself on trace files from the original execution. If this is not the correct moment to execute, it blocks the primitive until it receives a notification that the previous primitives were executed (it verifies if it can proceed again). The *executionController.valipar_after()* function notifies events waiting to execute.

The algorithms developed are based on the static analysis model for concurrent programs proposed by [Souza et al. 2014]. The model organizes the primitives according to their semantics, which allows its use in different languages, such as Java or C/MPI/Pthreads. The model extracts information on the three flows - control, data and synchronization - representing them in a *Parallel Control Flow Graph* (PCFG). Each node in this graph corresponds to a set of instructions executed in sequence (without a conditional diversion) or to a synchronization primitive [Souza et al. 2014] [Souza et al. 2013].

The synchronization primitives are classified according to the type of event (sender, receiver or sender-receiver) and operation semantic (blocking and non-blocking). Thus, a primitive can be classified as *Blocking Sender* (BS), *Blocking Receiver* (BR), *Non-Blocking Sender* (NS), *Non-Blocking Receiver* (NR), *Blocking Sender-Receiver* (BSR) and *Non-Blocking Sender-Receiver* (NSR). Souza et al. (2014) define synchronization clusters to avoid synchronization pairs with primitives of different semantics. For example, a receive from a collective message passing cannot receive a message from a point-to-point message passing send or synchronize with a semaphore. Different semaphores also do not interact and use different clusters.

The synchronization primitives were grouped into message passing, semaphores (each semaphore is a different cluster) and collective message passing. The collective primitives can still be subdivided into: one-to-all, all-to-one and all-to-all. These clusters are used to identify the synchronization pairs during the trace generation and to take decisions during the deterministic execution. The main features of the trace file generation algorithms and deterministic execution are described in Sections 3.1 and 3.2.

3.1. Trace File Generation

Our algorithms for trace file generation consider that each thread of the concurrent program maintains a list of synchronization events that occur during an execution. Each event inserted in this queue has an unique identification so that a synchronization event of node n, from thread t and process p is uniquely identified by the tuple $\langle n^{p,t}, e\rangle$, where e is the number of the event listed in sequence by t. The event number distinguishes a node executed within a loop, i.e., a new event means a re-execution of a node.

Each event in the queue registers its type (sender, receiver or sender-receiver) and its cluster. Furthermore, each receiver executed also has information about the sender event that matches with it and also has a global logical time for the concurrent application. The global logical time is a *time-stamp* represented by *ts_global*. Each event has a state, which represents whether the event actually established a synchronization pair, when the receiver primitive is returned. In order to establish the synchronization pairs, a *First-In-First-Out* (FIFO) criterion is considered, assuming that the messages are received in the same order in which they are sent and that the semaphore queues wake up the threads in

the same order in which the threads requested access to the critical region.

The *ts_global* attributed to the receiving events represents the chronological order in which the events were executed. It is attributed by an external process called *Control Process* (CP). CP initiates together with the instrumented program and manages the value attribution for the *ts_global*. The attributions take place by message passing. The *ts_global* determines the order in which the receivers are executed during the deterministic execution. It begins at zero and is increased for each request made at the CP.

Algorithm 2 shows the steps needed to generate trace files considering blocking message passing primitives. Before the execution of a blocking message passing primitive, the function *valipar_before* will be called on both for the sender and receiver. For the sender the identification of the event will be added to the message to be sent; if it is a receiver, no additional step will be carried out and the primitive will be just executed.

Algorithm 2 Trace generation for blocking message passing.

```
 1: function VALIPAR_BEFORE(event, cluster, semantic, msg)
 2:     if semantic = BS then
 3:         msg ← event + message
 4:     end if
 5: end function

 6: function VALIPAR_AFTER(event, cluster, semantic, status, msg)
 7:     my_ts_global ← ∅
 8:     sender ← ∅
 9:     result.event ← 1
10:     result.status ← status
11:     if semantic = BR then
12:         DOWN(tsPairSem)
13:         SEND(CP, request for a new ts_global)
14:         RECEIVE(CP, my_ts_global)
15:         sender ← sender event from msg
16:         msg ← msg without sender event
17:         UP(tsPairSem)
18:     end if
19:     SAVE_EVENT(event,cluster,semantic,sender,my_ts_global, result)
20: end function
```

After executing the sender primitive and the receipt of the message by the receiver, the valipar_after functions will be executed. For the sender event, the function will just save the event in the thread's queue, which will be stored in the thread's trace file. If the event is a blocking receiver, the thread receiver gains the semaphore *tsPairSem* to ensure that it will also retrieve its *ts_global* without interference from a receiver belonging to another thread. When gaining the semaphore, the thread receiver sends a message to the CP, requesting a *ts_global*. If it is the first receiver event to execute, it will receive a message with the value zero. The thread receiver retrieves the information from the send message and leaves the critical region. Then the receiver is added to the thread queue.

In general, non-blocking message passing primitives can work in two ways: (a) they are executed in background and return identifiers so that it is possible to consult the state of the operation (case of MPI); or (b) they live (i.e. are active and executing) just during the time of the call and return the amount of sent/received bytes. In this last way, if the message has not been totally transmitted or received, the primitive should be called again (case of *SocketChannels* in Java). The algorithm proposed for non-blocking message passing considers these two types of non-blocking events (Algorithm 4). For the first case (a), besides the message passing cluster, two new clusters are introduced: *wait* and

Algorithm 3 Trace generation for shared memory.

```
 1: function VALIPAR_BEFORE(event, cluster, semantic, msg)
 2:     if semantic = BS then
 3:         DOWN(queueSem)
 4:         QUEUE_INSERT(cluster, event)
 5:         UP(queueSem)
 6:     end if
 7: end function

 8: function VALIPAR_AFTER(event, cluster, semantic, status, msg)
 9:     my_ts_global ← ∅
10:     sender ← ∅
11:     result.event ← 1
12:     result.status ← status
13:     if semantic = BR then
14:         DOWN(tsPairSem)
15:         SEND(CP, request for a new ts_global)
16:         RECEIVE(CP, my_ts_global)
17:         DOWN(queueSem)
18:         sender ← QUEUE_REMOVE(cluster)
19:         UP(queueSem)
20:         UP(tsPairSem)
21:     end if
22:     SAVE_EVENT(event,cluster,semantic,sender,my_ts_global, result)
23: end function
```

test. These clusters represent the primitives that verify the state of a non-blocking primitive based on the returned identifier. The wait primitive has a blocking behaviour, while the primitive test has a non-blocking behaviour. The *ts_global* and the synchronization pair are only attributed to receiver events (including test and wait of received primitives) that will actually receive a message (Algorithm 4.15[2]).

As in the blocking message passing algorithm, the trace generation for non-blocking primitives uses its own message to insert the sender's identification. The difference takes place during the *valipar_after* function, where the identifier returned by the non-blocking primitive is inserted into a hash table, as the key to retrieve the tuple with the identification and the semantic of the original primitive (Algorithm 4.24). When a primitive *test* or *wait* is executed, this information will be retrieved from the hash table so that it is saved in the trace (Algorithm 4.11–14). For this case, the state of the trace file is a tuple with both the event identifier of the origin and the primitive state.

The synchronization made using collective primitives were organized in the following clusters: one-to-all, all-to-one and all-to-all. A queue is used to register the synchronization pairs that participate in these synchronizations. This queue needs to be accessible for all the concurrent processes/threads, which participate in some synchronization. Thus, it was allocated in the CP. Exchanging messages enables the insertion and retrieval of data in the queue. The algorithms for collective primitives use a *communicator* to identify subgroups within the same cluster. This subgroup separates, for example, two barriers or different broadcast groups that, despite belonging to the same cluster, do not interact. The creation of these subgroups is dynamic and non-static using the source code, because their requirement is determined on-the-fly, according to the execution flow of the threads. The CP maintains a queue for each cluster of the collective primitives. When receiving an insertion request, a data structure is added to the queue of the cluster containing the identification of the sender event and its communicator.

[2] Algorithm $x.y$ indicates the line y of algorithm x.

Algorithm 4 Trace generation for non-blocking message passing.

```
1:  function VALIPAR_BEFORE(event, cluster, semantic, msg)
2:      if semantic = NS then
3:          msg ← event + message
4:      end if
5:  end function

6:  function VALIPAR_AFTER(event, cluster, semantic, status, msg)
7:      my_ts_global ← ∅
8:      sender ← ∅
9:      result.event ← ∅
10:     result.status ← status
11:     if cluster = test or cluster = wait then
12:         result.event ← HASH_RETRIEVE_SENDER(identifier)
13:         semantic ← HASH_RETRIEVE_SEMANTIC(identifier)
14:     end if
15:     if semantic = NR and result.status = 1 then
16:         DOWN(tsPairSem)
17:         SEND(CP, request for a new ts_global)
18:         RECEIVE(CP, my_ts_global)
19:         sender ← sender event from msg
20:         msg ← msg without sender event
21:         UP(tsPairSem)
22:     end if
23:     if (semantic = NR or semantic = NS) and cluster ≠ test and cluster ≠ wait then
24:         HASH_INSERT(identifier, event, semantic)
25:     end if
26:     SAVE_EVENT(event,cluster,semantic,sender,my_ts_global, result)
27: end function
```

3.2. Deterministic Execution

To replay a specific sequence of synchronization from a concurrent program, the deterministic mode should be activated at the beginning of the execution, so that the control functions are called on before and after each primitive (Algorithm 1). During the initialization of each thread, it loads the content of the respective trace files from the original execution and initializes with zero a local variable, which represents the *ts_global*. This variable controls the progress of the receiver events.

Before a receiver event is executed, it verifies if the *ts_global* correspondent to the trace is greater or equals to the value of the *ts_global* of the thread. The receiver primitive can only be executed if this condition is satisfied; if it is not, the thread will wait until it receives a notification with the updated value of the *ts_global*. When this message is received, the thread updates its *ts_global* and verifies if the primitive can be re-executed.

The *valipar_after* function, called on soon after executing the receiver primitive, increases the *ts_global* value and propagates it by broadcast to all the threads that are blocked (waiting), freeing the next receiver.

Sender events have their execution controlled in two ways: (a) they wait for the receiver event to be freed in a blocking receive or (b) they are executed according to an ordered queue that represents the order in which the events take place in a determined cluster from the original execution. The first case is used to control point-to-point message passing events, as these events can be freed only when the receiver event is ready, avoiding a race condition. The second case is used for shared memory and for collective primitives.

Algorithm 5 presents the deterministic execution solution for blocking point-to-point message passing. Before the sender events execute the primitive, they wait for a confirmation message from their original receiver (Algorithm 5.5), enabling the receiver

to decide from whom it will receive the message, thus avoiding non-determinism. When the receiver is ready to execute, i.e., *ts_global* of the receiver event, obtained from the original trace, is equal to the *ts_global*, a message is sent to its sender (Algorithm 5.10). This message frees the sender to execute its primitive. The receiver waits for a confirmation from the sender that the message was sent, to be able to execute the receive primitive (Algorithm 5.11), to which it is sent soon after the primitive execution (Algorithm 5.16).

Algorithm 5 Deterministic Execution for blocking message passing.

```
 1: ts_global ← 0

 2: function VALIPAR_BEFORE(event, cluster, semantic, msg)
 3:     e ← current event from trace
 4:     if semantic = BS then
 5:         receiver ← RECEIVE(any receiver, a release msg)
 6:     else if semantic = BR then
 7:         while e.ts > ts_global do
 8:             RECEIVE(any_thread, ts_global)
 9:         end while
10:         SEND(e.sender, a release msg)
11:         RECEIVE(e.sender, a release msg)
12:     end if
13: end function

14: function VALIPAR_AFTER(event, cluster, semantic, status, msg)
15:     if semantic = BS then
16:         SEND(receiver, a release msg)
17:     else if semantic = BR then
18:         ts_global++
19:         BROADCAST(all other threads, ts_global)
20:     end if
21:     current event ← next event from trace
22: end function
```

Besides the original trace, the blocking shared memory algorithm loads a queue for each cluster, including all the sender events that occur in the same order in which they were executed in the original execution. Before a shared memory sender primitive is executed, the event loaded from the trace will be compared to the top of the queue. If the event is not equal to the top event, it will sleep in a condition variable. If it is, two actions will be carried out after executing the primitive: the top of the cluster queue is moved and all the sender events that were waiting are woken up. The process for the receivers happens in a similar way to the blocking message passing, except for freeing the sender and confirmation message.

During the deterministic execution of the non-blocking message passing events, the sender events wait for the original receivers to be free in the same way as the blocking message passing. The receiver primitives that do not establish a synchronization pair are free to execute, and as the sender events wait to be freed, all the receiving events (receive, test and wait) present the same behavior as the original execution. Before the execution of a receiver event that established a synchronization, a message is sent to the sender to free it and a confirmation is expected from this sender before the receiver primitive is executed. When the sender is freed to execution, a confirmation is sent to the receiver event, which then executes the reception of the message. This confirmation is used so that the receiver primitive is only executed after the message is sent, ensuring that it is received by the same receptor of the original execution. One non-blocking message passing receiver event waits for the *ts_global* only if it is the first event in its thread.

The deterministic execution for all-to-all collective primitives is based on the order in which the events were executed inside the cluster. Therefore, a queue containing the order of events is loaded in each thread during the beginning of the replay. Before an event is executed, it verifies the top of the queue. If it is not the event at the top, a message is sent to the CP to notify it that this event is waiting for an alteration at the top of the queue. Before executing the event that is at the top of the queue, the top of the queue is moved and a message is sent to the CP. When this message is received, the CP sends a message to each thread that is waiting, so that their events can attempt to execute again.

The deterministic execution for one-to-all and all-to-one collective primitives is controlled with the support of the CP using a queue. Each cluster has a queue of sender events in the same order in which they were saved in the trace. A sender event can only be executed if it is at the top of the queue. If it is not, it waits for a notification from the CP that the top was moved. At this moment it can try to execute again. After executing a sender event, the queue is moved and a notification is sent to the CP so that it frees the threads that are waiting. The receive events are controlled by the *ts_global* in a similar way to the blocking message passing algorithm.

4. Evaluation and Analysis of the Results

Prototypes of the algorithms presented in this paper were developed in Java to evaluate: (a) if the algorithms are capable of re-executing a concurrent program deterministically; (b) the overhead added by the deterministic execution; and (c) the size of the trace files.

Five microbenchmarks were used while the algorithms were being developed and evaluated: Blocking Message Passing (BMP), Non-blocking Message Passing (NMP), Shared Memory (SM), All-to-All (ATA) and One-To-All (OTA). All the benchmarks used to validate our proposal are described and available for download in *http://testpar.icmc.usp.br/benchmarks* [Dourado et al. 2016]. Each microbenchmark has only one type of primitive and thus validates a specific behaviour of the algorithm. Table 1 shows the main features of the microbenchmarks: paradigm (Message Passing (MP) or Shared Memory (SM)), semantic (Point-To-Point (P2P); Collective (COL); Blocking (BK); Non-Blocking (NBK); All-To-all (ATA) or One-to-all (OTA)), number of processes and threads (Procs and Threads), send events (Sndrs), receive events (Rcvrs), send-receive events (Sndr/Rcvr), Lines Of Code (LOC) and Cyclomatic Complexity (CC). We did not find examples of all-to-one primitives in Java.

Table 1. Microbenchmarks used during the development.

Bench	Paradigm	Semantic	Procs	Threads	Sndrs	Rcvrs	Sndr/Rcvr	LOC	CC
BMP	MP	P2P/BQ	3	3	4	4	0	66	1.2
SM	SM	P2P/BQ	1	3	9	8	0	42	1.0
NMP	MP	P2P/NBQ	3	3	3	3	0	99	2.8
ATA	SM	COL/ATA	1	4	0	0	6	65	1.5
OTA	MP	COL/OTA	4	4	2	4	0	47	2.5

Four other larger benchmarks with a combination of primitives were also used. They were Producer-Consumer (PC), Prime-Server (PS), Multiplication of Matrices (MM) and Token-Ring (TR). PC only has shared memory primitives and the other three have both paradigms (message passing and shared memory); PS and TR have blocking and non-blocking point-to-point primitives and the rest use blocking primitives. Table

4 presents the main characteristics of these four new benchmarks. TR also has broadcast (one-to-all) and barrier (all-to-all) collective primitives.

Table 2. Benchmarks used to evaluate if the deterministic re-execution is correct, the overhead of the deterministic execution and the trace size.

Bench	Proc	Thread	Sndr	Rcvr	Snd/Rcv	LOC	CC
PC	1	5	19	16	0	140	1.2
PS	4	7	29	18	0	210	1.5
MM	5	13	37	30	0	227	1.4
TR	3	7	25	17	20	296	2.6

The benchmarks were executed in a computer using an Intel Core i7 - 36100M 2.3GHz CPU, 8GB of RAM, Ubuntu Server 14.04 and OpenJDK Java Virtual Machine 1.7.0_55. Each benchmark was executed 30 times non-deterministically and 30 times deterministically. We collected for each execution: response time, standard deviation (SD), amount of generated trace files and trace file size (bytes).

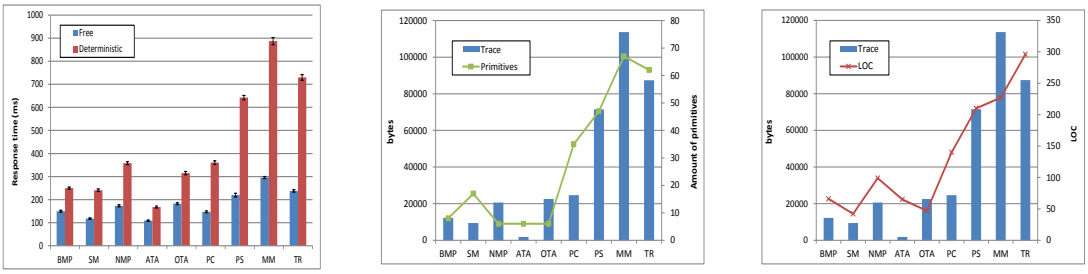
All the deterministic executions generated with success new trace files, these equivalent to the respective generated trace files from the free executions. Table 3 presents the results, highlighting the cost added by the deterministic execution. As observed in the data presented, the additional cost to the response time by the deterministic execution was between 154% and 307% with relation to the free execution. For the microbenchmarks, the overhead was lower than 173% and for the benchmarks, the cost was slightly higher, between 244% and 307%. These results were expected and show that the cost of the response time is higher, according to an increase of the amount of primitives. Despite this increase in the response time, the trace file generation and the deterministic executions met their objectives: to describe the concurrent execution behaviour of an application and enable it to be replayed, considering the structural testing of concurrent programs. It should be mentioned here that this overhead considers traces with information about all the executed nodes, as well as the definitions and variable uses. Without this information, the overhead would be smaller.

Table 3. Overhead of deterministic execution and trace size.

Bench	Free exec time [SD]	Determ exec time [SD]	Overhead	Trace amount	Trace size (bytes)
BMP	149.63 ms [11.57]	249.87 ms [13.26]	166.99%	6	12198
SM	117.73 ms [8.19]	240.63 ms [15.59]	204.39%	7	9418
NMP	173.10 ms [13.14]	358.30 ms [17.30]	206.99%	6	20631
ATA	108.80 ms [9.14]	167.93 ms [12.03]	154.35%	2	1839
OTA	182.60 ms [11.35]	315.03 ms [18.54]	172.53%	9	22526
PC	147.20 ms [11.99]	360.57 ms [22.98]	244.95%	13	24622.8
PS	219.70 ms [21.54]	642.30 ms [26.95]	292.35%	17	71499
MM	296.00 ms [12.14]	887.37 ms [43.17]	299.79%	30	113671
TR	237.67 ms [14.56]	729.83 ms [27.76]	307.07%	21	87384

Figure 1(a) illustrates a comparison between the average response times (ms) for the non-deterministic execution (free) and the deterministic execution, with their standard deviations (SD) and with a confidence interval of 95% (based on a normal distribution).

As observed in Figure 1(b), the size of the trace file is strongly related to the amount of primitives in the benchmarks. The second factor that influences the size of the trace is the amount of code lines of the benchmark (Figure 1(c)). This happens because



(a) (b) (c)

Figure 1. (a) Average response time of free and deterministic executions, (b) Trace size in relation to amount of primitives, and (c) Trace size in relation to code lines.

other events (such as: the beginning of a node, definitions and use of variables) are also provided by ValiInst and recorded.

5. Final Considerations

In this paper, algorithms for trace file generation and deterministic execution for six types of synchronization primitives were presented: blocking and non-blocking point-to-point message passing, blocking shared memory (semaphore), one-to-all, all-to-one and all-to-all collective primitives. Furthermore, the algorithms can be used when these different paradigms are used in concurrent programs at the same time.

For all the experiments carried out, the algorithms met their objectives of replaying a concurrent program in a deterministic way. Information such as synchronization pairs and the order of events are provided in the trace files. This information is important for the structural testing of concurrent programs, during the evaluation of the coverage of synchronization edges and to determine the definition and use of shared variables.

The experiments carried out demonstrate that the overhead of the deterministic execution and the trace file size increase proportionally to the amount of primitives used in the concurrent program. Despite being costy, both the trace file generation and the deterministic execution have many benefits. The deterministic execution will be applied directly to the structural testing of concurrent programs, enabling replay of a revealed fault and helping in the debugging activity. Still concerning about the structural testing of concurrent programs, the deterministic execution will also be used by the automatic generation of new synchronization sequences (or variants), enabling new synchronizations from a concurrent program to be covered from a trace file.

6. Acknowledgements

The authors thank FAPESP for the financial support (2012/14285-4, 2013/01818-7, 2013/05750-8 and 2013/07375-0).

References

[Alpern et al. 2000] Alpern, B., Ngo, T., Choi, J.-D., and Sridharan, M. (2000). Dejavu: Deterministic java replay debugger for jalapeño java virtual machine. In *OOPSLA '00*, pages 165–166, NY. ACM.

[Bergan et al. 2010] Bergan, T., Anderson, O., Devietti, J., Ceze, L., and Grossman, D. (2010). Coredet: a compiler and runtime system for deterministic multithreaded execution. In *ASPLOS 2010*, volume 38, pages 53–64. ACM.

[Bocchino et al. 2009] Bocchino, Jr., R. L., Adve, V. S., Dig, D., Adve, S. V., Heumann, S., Komuravelli, R., Overbey, J., Simmons, P., Sung, H., and Vakilian, M. (2009). A type and effect system for deterministic parallel java. *SIGPLAN Not.*, 44(10):97–116.

[Carver and Lei 2010] Carver, R. and Lei, Y. (2010). A class library for implementing, testing, and debugging concurrent programs. *International Journal on Software Tools for Technology Transfer*, 12(1):69–88.

[Carver and Tai 1991] Carver, R. and Tai, K.-C. (1991). Replay and testing for concurrent programs. *Software, IEEE*, 8(2):66–74.

[Dourado et al. 2016] Dourado, G. G. M., Souza, P. S. L., Prado, R. R., Batista, R. N., Souza, S., Estrella, J., Bruschi, S., and Lourenco, J. M. S. (2016). A suite of java message-passing benchmarks to support the validation of testing models criteria and tools. In *ICCS 2016*, volume 80, pages 2226–2230. Elsevier.

[Farchi et al. 2003] Farchi, E., Nir, Y., and Ur, S. (2003). Concurrent bug patterns and how to test them. In *Parallel and Distributed Processing Symposium*, pages 7 pp.–.

[Konuru et al. 2000] Konuru, R., Srinivasan, H., and Choi, J.-D. (2000). Deterministic replay of distributed java applications. In *IPDPS 2000*, pages 219–, Washington. IEEE.

[Lei and Carver 2006] Lei, Y. and Carver, R. (2006). Reachability testing of concurrent programs. *IEEE T. on Software Engineering*, 32(6):382–403.

[Lifflander et al. 2014] Lifflander, J., Meneses, E., Menon, H., Miller, P., Krishnamoorthy, S., and Kale, L. (2014). Scalable replay with partial-order dependencies for message-logging fault tolerance. In *IEEE Cluster Computing 2014*, pages 19–28.

[Liu et al. 2011] Liu, T., Curtsinger, C., and Berger, E. D. (2011). Dthreads: efficient deterministic multithreading. In *SOSP 2011*, pages 327–336. ACM.

[Musuvathi et al. 2008] Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P. A., and Neamtiu, I. (2008). Finding and reproducing heisenbugs in concurrent programs. In *OSDI 2008*, pages 267–280, Berkeley. USENIX.

[Olszewski et al. 2009] Olszewski, M., Ansel, J., and Amarasinghe, S. (2009). Kendo: Efficient deterministic multithreading in software. *SIGPLAN Not.*, 44(3):97–108.

[Prado et al. 2015] Prado, R. R., Souza, P. S., Dourado, G. G. M., Senger, S. R. S., Estrella, J. C., Bruschi, S. M., and Lourenco, J. (2015). Extracting static and dynamic structural information from java concurrent programs for coverage testing. In *CLEI 2015*, pages 1–8, Arequipa. IEEE.

[Sarmanho et al. 2008] Sarmanho, F. S., Souza, P. S., Souza, S. R., and Simão, A. S. (2008). Structural testing for semaphore-based multithread programs. In *ICCS 2008*, pages 337–346, Berlin. Springer-Verlag.

[Souza et al. 2014] Souza, P. S., Souza, S. R., and Zaluska, E. (2014). Structural testing for message-passing concurrent programs: an extended test model. *Concurrency and Computation: Practice and Experience*, 26(1):21–50.

[Souza et al. 2013] Souza, P. S., Souza, S. S., Rocha, M. G., Prado, R. R., and Batista, R. N. (2013). Data flow testing in concurrent programs with message passing and shared memory paradigms. *Elsevier*, 18(0):149 – 158. ICCS 2013.