Performance Evaluation of Concurrent Data Structures

Hao Wu[1(✉)], Xiaoxiao Yang[2], and Joost-Pieter Katoen[1]

[1] Software Modelling and Verification Group, RWTH Aachen University, Aachen, Germany
hao.wu@cs.rwth-aachen.de

[2] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China

Abstract. The speed-ups acquired by concurrent programming heavily rely on exploiting highly concurrent data structures. This has led to a variety of coarse-grained and fine-grained locking to lock-free data structures. The performance of such data structures is typically analysed by simulation or implementation. We advocate a *model-based* approach using *probabilistic model checking*. The main benefit is that our models can also be used to check the correctness of the data structures. The paper details the approach, and reports on experimental results on several concurrent stacks, queues, and lists. Our analysis yields worst- and best-case bounds on performance metrics such as expected time and probabilities to finish a certain number of operations within a deadline.

1 Introduction

Background and Motivation. Multi-core computers are ubiquitous. However shared concurrent data structures [11] are an important obstacle. The downside of lock-based data structures is that they are a sequential bottleneck. Lock-free data structures are resilient to failures, are more complex, and require special synchronisation primitives. Modern multi-core architectures support compare-and-swap operations to allow threads to read, modify and write atomically. Correctness of concurrent data structures is a key issue and typically addressed by a semi-formal pencil proof; performance is typically assessed by simulation or implementation [2,4]. We propose to carry out both correctness and performance analysis using a single, *model-based*, approach. This has the advantage that both analyses use the same model, and that results are coherent. In addition, it allows for using *a single technique*—model checking—*for both types of analysis*.

Modelling approach. The starting point for our approach is to model the concurrent data structure at hand, together with the threads that perform operations on it. We use the LOTOS NT language (LNT[1], for short) which is a

Supported by the CDZ project CAP (GZ 1023) and the A. von Humboldt-Foundation.

[1] The LNT language is a formal description technique standardized by ISO OSI (1989), please refer to: http://www.iso.org/iso/catalogue_detail.htm?csnumber=16258.

M. Fränzle et al. (Eds.): SETTA 2016, LNCS 9984, pp. 38–49, 2016.
DOI: 10.1007/978-3-319-47677-3_3

compositional modelling language with process algebraic roots, that supports abstract data specifications. The structure of the entire system has the form $(T_1 \,|||\, \ldots \,|||\, T_n) \,||_G\, D$ where the n threads T_1 through T_n which are independent (hence indicated by $|||$, a shorthand for $||_\emptyset$) and communicate with the concurrent data structure D via the communication gates G. To keep the state space finite, we bound the number of operations by applying a monitor process M that keeps track of the number of read and write operations. Using the CADP toolbox [7], the underlying state space can be generated and be analysed for checking functional correctness. We focus here on performance evaluation.

Performance modelling and evaluation. To enable performance evaluation, we assume that delays between reads and writes are random in nature and are governed by negative exponential distributions. We insert these random delays into the model by renaming such read and write actions to Markovian delay with different rates in CADP. As a result, CADP yields (after a mild post-processing) a *Markov automaton* [3,6]. These automata have random delay transitions, and allow for non-determinism, a feature that we exploit to model the concurrency among the threads. Prior to the performance evaluation, the state space is minimised using branching bisimulation minimisation in CADP, which preserves the important properties in the performance evaluation [15]. The analysis of Markov automata is enabled using recently developed algorithms by Guck *et al.* [8,9], which allow for determining the expected time until a certain state is reached—like "what is the expected time until each thread has completed 10 reads and writes?"—and the likelihood to reach a state within a given deadline—like "what is the probability that all reads and writes finish within 10 min?". Due to the inherent non-determinism (due to concurrency), the analysis does not obtain *the* expected time, or *the* probability, but rather obtains *bounds*. These bounds represent the best- and worst-case scenarios. The quantitative analysis of MA is supported by the MAMA tool-set[2]. The entire approach is given in Fig. 1.

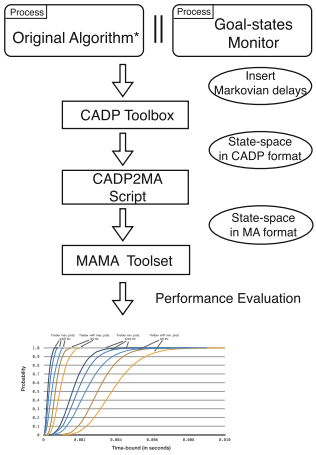

Fig. 1. Our approach

Experimentation. We have applied our approach to the modelling and performance evaluation of several concurrent stacks, queues, and lists. We treat the Treiber stack [13] and its variant with hazard pointers, see e.g., [16]. In addition, we cover the Michael-Scott two-lock (MS 2L) queue [14], its lock-free variant (MS LF) [14], and an improvement on this lock-free variant [5] (known as DGLM). Finally, we consider the coarse-grained synchronisation list [11, Chap. 9.4], a fine-grained synchronization list [11, Chap. 9.5], the lazy list [10] and an optimistic

[2] http://www.home.cs.utwente.nl/~timmer/mama/.

list [11, Chap. 9.6]. Our performance evaluation treats models of up to 378 million states. The experiments show that—as expected—lock-based data structures have a rather deterministic performance, whereas lock-free and fine-grained lock-based show more variance in their performance. In addition, fine-grained lists may yield a lower throughput as under intense race conditions many unsuccessful operations are carried out. To the best of our knowledge this is the first work that formally models concurrent data structures and uses this for a performance evaluation. Related works are the probabilistic model checking of low-level OS kernels including spin-locks [1] and the modelling and performance evaluation of mutual exclusion algorithms [12].

```
1 type Qnode is
    Qnode (next: Nat)
    with "get", "set"
  end type

3 process Thread[M: Queue_Ops,
   HL, TL: Lock_Ops, complete,
   T: Thread_Ops](pid: Pid) is
   loop
    select
     Enq[M, TL, T](pid); complete
     []
     Deq[M, HL, T](pid); complete
    end select
   end loop
  end process

4 process Enq [M: Queue_Ops, TL: Lock_Ops,
               T: Thread_Ops] (pid: Pid) is
   var locked : Bool in
    TL(lock_tail, pid);
    loop G in
     TL(test_and_set, ?locked, pid);
     if (locked == false) then break G
     else T(thr_delay, pid)
     end if
    end loop;
    M(set_tail_next, pid);
    M(set_tail, pid);
    TL(unlock_tail, pid)
  end var end process

5 process H_Lock[HL: Lock_Ops] is
   var locked : Bool, pid: Pid in
    locked := false;
    loop select
     HL(lock_head, ?pid)
     []
     HL(test_and_set, false, ?pid)
        where (locked == false);
     locked := true
     []
     HL(test_and_set, true, ?pid)
        where (locked == true)
     []
     HL(unlock_head, ?pid);
     locked := false
    end select end loop
  end var end process

2 type Memory is
    array [0 .. 10] of Qnode
    with "get", "set"
  end type

6 process Queue[M: Queue_Ops] is
   var m: Memory, head, tail, size, hd,
    pos, pos_next: Nat, pid: Pid in
    size := 10;  head := 0; tail := 0;
    m := Memory(Qnode (0 of Nat));
     loop select
      M(read_head, head, ?any Pid)
      []
      M(read_next, ?pos, ?pos_next, ?pid)
          where (m[pos].next == pos_next)
      []
      M(set_tail_next, ?pid);
       m[tail] := m[tail].{next => (tail + 1)}
      []
      M(set_tail, ?pid); tail := m[tail].next
      []
      M(set_head, ?hd, ?pid);
      head := m[head].next
     end select end loop
  end var end process

7 process MAIN [M: Queue_Ops, HL, TL: Lock_Ops,
   finish, complete, T: Thread_Ops] is
   par M, HL, TL, complete, T in
    par  M in
     par HL, TL in
      par
       Thread[M, HL, TL, complete, T] (1 of Pid)
       ||
       Thread[M, HL, TL, complete, T] (2 of Pid)
      end par
      ||
      par
       H_Lock[HL]
       ||
       T_Lock[TL]
      end par
     end par
     ||
     Queue [M]
    end par
    ||
    Monitor [M, HL, TL, finish, complete, T]
  end par end process
```

Fig. 2. The (partial) LNT code of the MS 2L queue

2 Modeling Concurrent Data Structures in LNT

For space sake we only show the LNT model[3] of the MS 2L queue [14] here in Fig. 2. Furthermore, the references of LNT language and various tools (e.g. state space minimisation) provided by CDAP can be found under http://cadp.inria.fr.

Figure 2 - 1 2 define the data structure of the MS 2L queue. It consists of a bounded array of `Qnodes`, since mutable dynamic data structure is not supported by LNT. Moreover, since to model a pointer is also not possible here, the `next` field of the `Qnode` stores the index (a natural number) of its next node in the array. A thread (Fig. 2 - 3) identified by `pid` needs to synchronize with the lock (process) for head (i.e., `H_Lock`), the lock for tail (i.e., `T_Lock`) and the queue (process) to perform enqueue and dequeue operations, hence the gates (`M, HL, TL`) required for synchronization with these processes are declared. Further, the gates (complete, T) indicate the process's own operations. The behavior of a thread is to repeatedly perform (enclosed by `loop`) either an enqueue or a dequeue operation and emits a `complete` signal (synchronized with monitor process for counting) if the operation is finished. Figure 2 - 4 defines the enqueue operation (for dequeue similarly) of a thread. First it tries to acquire the tail lock (via gate `TL`) before performing operations on the queue. The two locks are assumed to be simple *test-and-set* locks with *fixed* back-off delay. The delay (`T(thr_delay, pid)`) are inserted after each unsuccessful try of test-and-set, then the test-and-set is restarted. If it is succeed, the loop is exited and the operations to enqueue (e.g., `set_tail_next` (set tail's next to new node) and `set_tail`) are performed via the synchronization with gate `M`. Finally, the lock is released (`unlock_tail`). Figure 2 - 5 defines the head lock, which consists of a boolean variable `locked` and operations to lock with the `test_and_set` operation (it returns previous value of `locked` and set it to true atomically) and unlock via the gate `HL`. Note that complex locks can be modelled similarly. Figure 2 - 6 is the queue process which consists of the queue (array) and the head and tail (as indexes in the array) with auxiliary variables for synchronization. Note that enqueue a new node is simply represented by setting the tail's next to current tail's index + 1 in our model. Figure 2 - 7 defines the whole MS 2L queue: the threads, the two locks, the queue and the monitor process are composed in parallel with corresponding gates. Note that the threads are independent hence they do not synchronize.

3 Towards Performance Evaluation

In performance evaluation, we bound number of operations on the data structures to keep the state space finite. The monitor process synchronises with the

[3] The complete LNT models and their corresponding scripts of all aforementioned concurrent data structures follow the same principle described here and can be found under the link: https://moves.rwth-aachen.de/wp-content/uploads/LNT-models.zip.

`complete` actions from threads and counts the successfully completed operations. Goal states indicate when the number of operations has reached a certain bound. The performance of the concurrent data structure is then evaluated based on reaching a goal state. Since we assume that performing the elementary operations (e.g., read/write, test_and_set, compare_and_swap) cause a random delays governed by negative exponential distribution. We use the renaming rules to replace such operations with Markovian transitions with rates in the state space.

Experimental setup. To conduct experiments, we used the workflow as depicted in Fig. 1. The CADP tool [7] is used to generate the state space of the parallel composition of the LNT models of the data structures, the threads, the monitor. CADP also supports branching bisimulation minimisation that we exploit to reduce the models prior to performance analysis. We developed a CADP2MA script that transforms the state space as generated by CADP into a Markov automaton (MA) [3,6]. MA are state-transition systems that cater for non-determinism and support random delays. The performance evaluation of MA is done using the recent MAMA tool-set [8]. This tool supports the numerical computation of several quantitative objectives on MA. Our experiments focus on two measures: (1) the expected time until the system completes a certain number of operations requested by the overall threads and (2) the probability of finishing all these operations within a given deadline. As a thread repeatedly performs enqueue (push, add) and dequeue (pop, remove) operations in a *non-deterministic* manner[4], our analysis does not yield a single number but yields two *bounds*. A lower bound on the expected time gives the minimal time that is needed on average to complete all operations; an upper bound gives the maximal time. The former can be understood as the best achievable scenario; the latter as the worst one.

Assumptions and parameter settings. As concurrent programs with shared data structures (and possibly pointers) in principle have an unbounded state space, we make the following assumptions and modelling choices so as to enable a performance evaluation on a finite state space: (1) The number of threads is fixed. These threads invoke pre-defined operations (like pop and push) of the concurrent data structure. The invocation of such operations is modelled by means of synchronisation. (2) We bound the number of performed operations. (3) As LNT does not natively support pointers, fixed-size arrays with pre-defined elements are used, an index (a natural number) is used to locate the node, and a pointer is treated as an index record. The delays that are used in our experiments are adopted from [12] where the performance of mutual exclusion algorithms was analysed. The analysis is not based on a specific processor architecture, it is assumed that local caches are absent, and all operations are carried out on global memory. The rates of the exponential distributions are: a read operation from global memory (rate 3000, i.e., on average $1/3000$ time units), write to a global memory (2000), and complex operations (1200) on variables in global memory

[4] Thus, the thread behavior is not biased to certain scenarios.

e.g., compare_and_swap or test_and_set actions. The experiments are conducted on a computer with 4 × 12-core AMD CPU @ 2.1 GHz and 192 GB memory under 64-bit Debian 7.6.

4 Experimental Results

This section reports on applying our approach to the modelling and performance evaluation of several concurrent stacks, queues, and lists. For each data structure, we modelled and compared several variants from the literature. We report on the state spaces, the performance analysis results, and discuss them.

Concurrent queues. We cover the MS 2L queue [14], its lock-free variant (MS LF) [14], and an improvement on this lock-free variant (known as DGLM) [5].

State space size and analysis times. Table 1 shows the state spaces of different configurations of the three queues, the reduced state space by probabilistic branching bisimulation minimisation [7], the reduction factor, and the times (in seconds) for generating + reducing the state space and analysing expected time objectives. For four threads, three operations are considered (due to state space explosion). As expected, the state space grows exponentially in the number of threads and number of operations. State spaces up to about 378 million states have been generated (MS LF, 3 threads and 15 operations). The bisimulation reduction times for large state spaces are about 50 % of the generation times. Since the actions in the resulting system cannot be delayed by any other actions, all actions (except delay transitions) are turned into τ-transitions. This gives rise to state space reductions of up to 99.9 %. The MS 2L queue has a relatively small state space, due to its nature of low concurrency.

Expected time results. Figure 3 shows the analysis results of the expected time to finish a number of operations on the queues. Observe that the MS 2L queue is rather deterministic as the minimal and maximal values are quite close. This does not hold for the lock-free queues. The performance of the lock-based queue thus

Table 1. State space and the analysis time of expected time for concurrent queues

	MS 2L queue				MS LF queue				DGLM queue			
#Thr.-#Oper.	state space	red. st. space	red. rate	gen.+red./comp. time (in s)	state space	red. st. space	red. rate	gen.+red./comp. time (in s)	state space	red. st. space	red. rate	gen.+red./comp. time (in s)
2-3	1623	273	0.8318	25/0.03	6206	502	0.9191	26/0.07	6079	483	0.9205	21/0.06
2-5	4862	561	0.8846	26/0.1	27023	1376	0.9491	30/0.47	25552	1350	0.9472	23/0.38
2-10	25942	1895	0.9270	27/0.97	292k	5296	0.9818	28/6.1	247522	5251	0.9788	27/5.5
2-15	74712	3956	0.9471	29/4.7	1.39M	11471	0.9917	52/28	1.06M	11440	0.9893	47/29
2-20	163k	6765	0.9584	32/11.7	4.49M	20066	0.9955	497/124	3.13M	19991	0.9936	104/95
2-30	502k	14585	0.9709	35/63	25.3M	44186	0.9983	964/580	15.2M	44031	0.9971	656/608
3-3	22832	753	0.9670	27/0.17	350k	3905	0.9888	29/3.5	326k	3796	0.9883	29/2.8
3-5	65540	1857	0.9717	29/1.1	2.21M	11649	0.9947	70/44	1.87M	11674	0.9938	65/36
3-10	332k	6346	0.9809	31/14.7	47.9M	46993	0.9990	2018/1183	31.7M	46452	0.9985	1398/1624
3-15	934k	13267	0.9858	46/83	378.3M	103k	0.9997	21135/10088	200M	101k	0.9995	9826/10130
4-3	286k	1814	0.9936	29/1.0	21.78M	23042	0.9989	853/204	19.3M	23631	0.9988	780/317

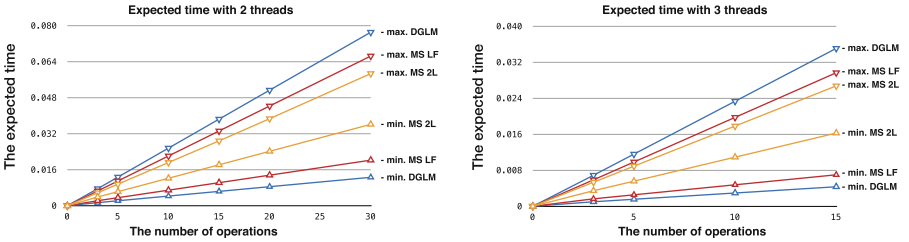

Fig. 3. Min./max. expected time versus # operations for concurrent queues (Color figure online)

provides a more stable service than its lock-free variants. Comparing Fig. 3 (left) and (right), we see that the expected time of finishing 3 operations with 2 threads is improved by 11 %/7 % in best/worst case of MS 2L queue, 26 %/10 % of MS LF queue, 27 %/9 % of DGLM queue with 3 threads (due to more concurrency), receptively. In best case the expected time for lock-free queues is much better than for the MS 2L queue comparing to the difference between lock-free queues and the MS 2L queue in worst case. The lock-free queues have—as expected—a much higher overhead than the lock-based queue. Thus, lock-free queues have a lower throughput in worst case than the lock-based one. The DGLM queue outperforms the MS LF queue in best case. Finally, we observe that expected times grow linearly in the number of operations.

Probability of timely completion. To get more insight into the performance of queues, we analyse the probability of finishing a certain number of operations within a (varying) deadline, see Fig. 4. Note that the faster the curve goes to one, the better the queue's performance. Since the number of operations only causes a linear increase in the expected delay, we consider three operations. We vary the number of threads from two to four. The three leftmost groups of curves indicate the maximal probabilities of the three queues with 4/3/2 threads, respectively. We can observe that the algorithms have a very strong impact on these results: lock-free queues perform much better than the lock-based one in best case. However in worst case, the performance of

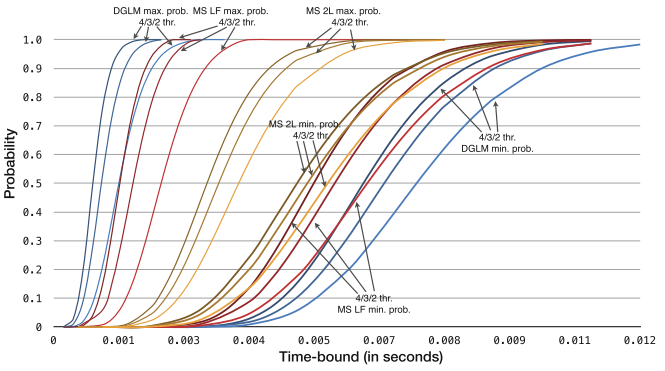

Fig. 4. How likely do concurrent queues complete three operations on time? (Color figure online)

algorithms is not that distinguishable (the rightmost curves), since they are close to each other. The number of threads in all queues influence these curves quite consistently, the more threads the much quicker the queue will finish the operations. As for the expected times, the lock-based algorithm behaves rather deterministic and its performance is quite stable when varying the number of threads.

Evaluation. In the best-case scenario, the two lock-free queues behave much better than the lock-based queue, in worst case however the lock-based queue outperforms the lock-free queues. Lock-based queues have a stable performance and are less vulnerable to the number of threads. The DGLM queue has better expected times than the MS LF in best case, but does not process operations within a deadline more likely.

Concurrent stacks. We consider two variants of the (lock-free) Treiber stack: one with hazard pointers (HPs) and one without. Hazard pointers [13] prevents the well-known ABA problem. HPs is used to keep certain locations as hazard and prohibit other threads to deallocate them. During garbage collection, only locations not pointed to by HPs can be freed. Our model of the Treiber-HP stack is based on [16] and includes a memory allocation thread. Our analyses focus on evaluating the performance influence of hazard pointers.

State space size and analysis times. Table 2 shows the state spaces for different parameter settings. The Treiber-HP stack causes a state space explosion due to the frequent scanning of HPs to find free locations after each `pop()`-operation. The branching bisimulation reduces the state space significantly (up to 0.999 in case of 4 threads). In cases for which the state space could be generated, the analysis time for Treiber-HP is prohibitive (taking hours).

Table 2. The state spaces and analysis times of expected time of Treiber stacks

	Treiber Stack				Treiber Stack + Hazard Pointers			
#Thr.-#Oper.	state space	red. st. space	red. rate	comp. time (in s)	state space	red. st. space	red. rate	comp. time (in s)
2-3	1081	57	0.947	16/0.005	7303	320	0.956	26/0.01
2-5	2876	165	0.943	17/0.02	70763	2445	0.965	38/1.5
2-10	13597	960	0.929	19/0.3	8.04M	258k	0.968	841/23591
2-15	37411	2978	0.920	21/2	M.O	M.O	-	-
2-20	79582	6744	0.915	25/9.7	M.O	M.O	-	-
2-30	240k	21941	0.909	22/112	M.O	M.O	-	-
3-3	16989	145	0.991	17/0.01	712k	2154	0.997	61/0.97
3-5	53320	527	0.990	19/0.1	18.35M	83972	0.995	1998/3580
3-10	358k	5146	0.986	21/7.3	M.O	M.O	-	-
3-15	1.29M	21011	0.984	39/113	M.O	M.O	-	-
4-3	265k	313	0.999	18/0.03	M.O	M.O	-	-

Expected time results. Figure 5 shows that the HPs cause significant overhead both in best and worst case. This is due to the additional operations of the HPs including setting/comparing with the HPs in `pop()`. It is interesting to observe that minimal expected times vary less than maximal ones. This is possibly due to the fact that HPs are not effective for `push()`-operations.

Probability of timely completion. As for the concurrent queues, we compute the time-bounded reachability of finishing three operations with 2/3/4 threads. We notice that increasing the number of threads in the system affect the minimal probability more than the maximal probability for both stacks. Moreover, the

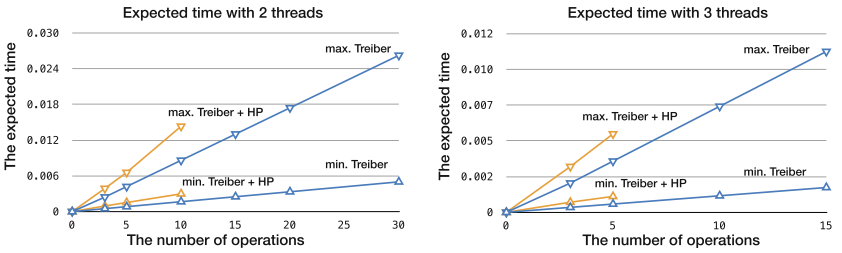

Fig. 5. The expected execution time versus # operations for Treiber stacks (Color figure online)

difference of the maximal time-bounded reachability between both stacks is quite small compared with the minimal values (Fig. 6).

Evaluation. We obverse that increasing the number of threads in the system has a significant impact on the minimal probability of finishing three operations for both stacks. Adding the HPs to Treiber stack is expensive in the worst case scenario, since the curves of minimal reachability probabilities of Treiber stacks with HPs (blue lines) are quite away from the corresponding curves (yellow lines) representing the minimal reachability probabilities of Treiber stacks without HPs. However, increasing the number of threads may alleviate this problem. Note that we did not have any result of the minimal probability of Treiber stack + HP with 4 threads due to memory out.

Concurrent lists. We consider four lock-based lists: a coarse-grained synchronization list (cgs) [11], a fine-grained synchronization list (fgs) [11], the Heller *et al.* lazy list (lazy) [10] and the optimistic list (opt.) [11]. All these concurrent lists are list-based implementations of a concurrent set object, where we can add (or remove) a value[5] to (or from) the list (set). Adding an element which is already in the list is unsuccessful. The lists are *data-dependent*, and their performance strongly depends on the data to be added or removed. Thus, we model an additional process to generate pseudo random numbers to be added or removed to/from the list. To test extreme situations, we set the generated random number to be only 1 or 2 in our experiment. This will cause a large number of unsuc-

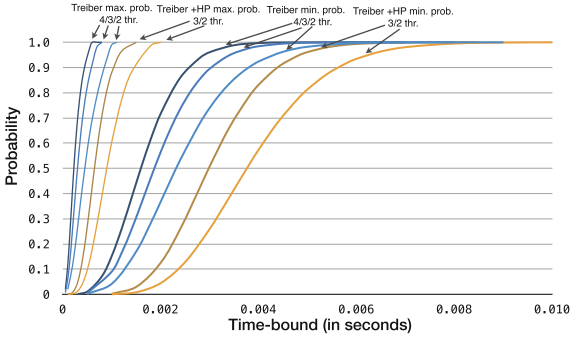

Fig. 6. The prob. of finishing 3 oper. within a given time-bound of Treiber stacks (Color figure online)

[5] In our experiments, we consider lists of natural numbers.

cessful operations. Since these data structures employ different granularities of locks, we discuss the effect of such granularities of locks on the performance under this setting. For space reasons, we focus only on the expected time of finishing different number of operations with several threads.

Table 3. The state spaces and comp. time of max/min expected time of lists

	Coarser grained sync. list (cgs)				Fine grained sync. list (fgs)				Lazy sync. list (lazy)				Optimistic sync. list (opt.)			
#Thr.-#Oper.	state space	red. st. space	red. rate	gen.+red./ comp. time (in s)	state space	red. st. space	red. rate	gen.+red./ comp. time (in s)	state space	red. st. space	red. rate	gen.+red./ comp. time (in s)	state space	red. st. space	red. rate	gen.+red./ comp. time (in s)
2-3	5871	541	0.908	26/0.09	6205	906	0.854	22/0.2	41147	3171	0.923	31/1.6	46859	3587	0.923	29/1.8
2-5	17343	1546	0.911	26/0.5	17713	2883	0.837	23/1.5	168k	11031	0.934	36/21	193k	13202	0.932	34/24
2-10	52305	4135	0.921	26/4	52063	8231	0.842	24/14	633k	42863	0.932	51/565	736k	43457	0.941	54/576
2-15	87393	6916	0.921	28/12	86983	13773	0.842	26/50	1.11M	78279	0.929	53/3485	1.29M	111k	0.914	62/11643
2-20	122k	9505	0.922	30/21	121k	19121	0.842	29/108	1.58M	112k	0.929	64/9158	1.84M	155k	0.916	90/15251
2-30	192k	14875	0.923	33/71	191k	30011	0.843	32/338	2.53M	181k	0.928	103/24475	2.95M	249k	0.916	128/58909
3-3	156k	2269	0.985	30/1.2	148k	3831	0.974	28/3.3	5.93M	57926	0.990	244/1375	6.82M	42497	0.994	294/422
3-5	445k	7624	0.983	33/14	409k	16033	0.961	32/89	33.7M	276k	0.992	49604/98989	40.2M	442k	0.989	T.O.
3-10	1.35M	23729	0.982	62/465	1.21M	57722	0.952	3524/266	M.O.	-	-	-	M.O.	-	-	-
3-15	2.27M	39212	0.983	94/1160	2.0M	98207	0.951	1980/7243	M.O.	-	-	-	M.O.	-	-	-

State space size and analysis times. Table 3 shows the generated state spaces and analysis times of the lists. State spaces in some scenarios can not be generated due to either memory out at state space generation or time out of computing the expected time. The generated state spaces correctly reflect the granularities of locks[6]: cgs > (is coarser than) fgs > lazy > opt. The bisimulation reduction times for the large state spaces are about 60 % of the generation times.

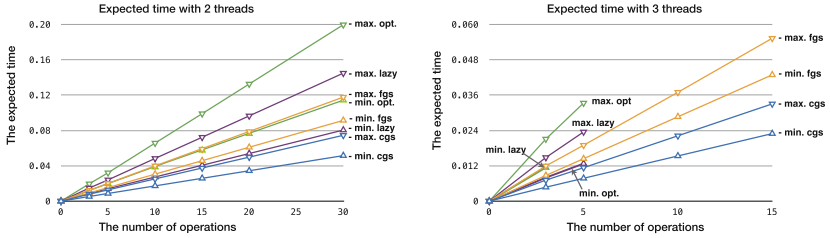

Fig. 7. The expected execution time versus # read/write oper. of concurrent lists (Color figure online)

Expected time results. One would expect that the fine-grained concurrency (=finer granularity of lock) allows more operations to be performed per unit time than coarse-grained concurrency. Hence one expects for the throughput: opt > (the expected time is smaller) > lazy > fgs > cgs. However, in our experiment this is not the case. Reversely, we can easily observe that cgs finishes 3 operations sooner than the others in both best and worst scenarios (Fig. 7).

[6] If for a given scenario, the number of states of a data structure is higher than for another data structure, it allows for more concurrency and has finer lock granularity.

Evaluation. The results of expected times above indicate that the finer-grained concurrency will not always achieve a higher throughput when the data race is (extremely) intensive. The more unsuccessful operations will lower the overall throughput and in such scenario the fine-grained implementation could be worse than the coarse-grained implementation.

5 Conclusion

This paper presented the modelling and performance analysis of various concurrent data structures using a combination of the CADP and MAMA tools. We emphasise that probabilistic model checkers such as PRISM/MRMC are not appropriate as they do not support non-deterministic continuous-time stochastic models. Future work consists of validating our model-based results against concurrent data structure implementations (e.g., in Java).

Acknowledgments. We thank Wendelin Serwe for his support in CADP.

References

1. Baier, C., Daum, M., Engel, B., Härtig, H., et al.: Locks: picking key methods for a scalable quantitative analysis. J. Comput. Syst. Sci. **81**(1), 258–287 (2015)
2. Cederman, D., Chatterjee, B., Tsigas, P.: Understanding the performance of concurrent data structures on graphics processors. In: Kaklamanis, C., Papatheodorou, T., Spirakis, P.G. (eds.) Euro-Par 2012. LNCS, vol. 7484, pp. 883–894. Springer, Heidelberg (2012). doi:10.1007/978-3-642-32820-6_87
3. Deng, Y., Hennessy, M.: On the semantics of Markov automata. Inf. Comput. **222**, 139–168 (2013)
4. Dodds, M., Haas, A., Kirsch, C.M.: A scalable, correct time-stamped stack. In: POPL, pp. 233–246. ACM (2015)
5. Doherty, S., Groves, L., Luchangco, V., Moir, M.: Formal verification of a practical lock-free queue algorithm. In: Frutos-Escrig, D., Núñez, M. (eds.) FORTE 2004. LNCS, vol. 3235, pp. 97–114. Springer, Heidelberg (2004). doi:10.1007/978-3-540-30232-2_7
6. Eisentraut, C., Hermanns, H., Zhang, L.: On probabilistic automata in continuous time. In: LICS, pp. 342–351 (2010)
7. Garavel, H., Lang, F., Mateescu, R., Serwe, W.: CADP 2011: a toolbox for the construction and analysis of distributed processes. STTT **15**(2), 89–107 (2013)
8. Guck, D., Hatefi, H., Hermanns, H., Katoen, J., Timmer, M.: Analysis of timed and long-run objectives for Markov automata. Log. Methods Comput. Sci. **10**(3) (2014)
9. Guck, D., Timmer, M., Hatefi, H., Ruijters, E., Stoelinga, M.: Modelling and analysis of Markov reward automata. In: Cassez, F., Raskin, J.-F. (eds.) ATVA 2014. LNCS, vol. 8837, pp. 168–184. Springer, Heidelberg (2014). doi:10.1007/978-3-319-11936-6_13
10. Heller, S., Herlihy, M., Luchangco, V., Moir, M., Scherer III, W.N., Shavit, N.: A lazy concurrent list-based set algorithm. Parallel Process. Lett. **17**(4), 411–424 (2007)

11. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann (2008)
12. Mateescu, R., Serwe, W.: Model checking and performance evaluation with CADP illustrated on shared-memory mutual exclusion protocols. Sci. Comput. Program. **78**(7), 843–861 (2013)
13. Michael, M.M.: Hazard pointers: safe memory reclamation for lock-free objects. IEEE Trans. Parallel Distrib. Syst. **15**(6), 491–504 (2004)
14. Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: PODC, pp. 267–275. ACM (1996)
15. Neuhäußer, M.R., Katoen, J.-P.: Bisimulation and logical preservation for continuous-time Markov decision processes. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 412–427. Springer, Heidelberg (2007). doi:10.1007/978-3-540-74407-8_28
16. Tofan, B., Schellhorn, G., Reif, W.: Formal verification of a lock-free stack with hazard pointers. In: Cerone, A., Pihlajasaari, P. (eds.) ICTAC 2011. LNCS, vol. 6916, pp. 239–255. Springer, Heidelberg (2011). doi:10.1007/978-3-642-23283-1_16