

Lock-free Stack and Queue: Java vs .NET

Matko Botinčan[‡], Davor Runje[§]

[‡] *Department of Mathematics, University of Zagreb
Bijenička cesta 30, 10000 Zagreb, Croatia*

E-mail: mabotinc@math.hr

[§] *Faculty of Mechanical Engineering and Naval Architecture, University of Zagreb
Ivana Lučića 5, 10002 Zagreb, Croatia*

E-mail: davor.runje@fsb.hr

Abstract. *In this paper, we report and analyze the behavior of Java and .NET implementations of lock-free stack and queue in different settings.*

Keywords. Concurrent programming, lock-free collections, Java, .NET, performance.

1. Introduction

In recent years, a considerable amount of research in area of concurrency is devoted to design, analysis and verification of non-blocking algorithms since they can deliver significant performance benefits to computer systems based on multi-core processors. Such algorithms typically use alternative synchronization techniques eschewing use of locks, and achieve a more robust and reliable performance than corresponding implementations having a lock-based synchronization.

A synchronization technique is *lock-free* if it ensures that whenever a thread tries to perform an action on the synchronized object, some thread (not necessarily the same one) will complete its action within a finite number of steps. In other words, the lock-free technique ensures that some thread always make progress. However, there is no progress guarantee for any individual thread, only for the set of active threads as a whole.

A concurrent data structure (i.e. a data structure that is meant to be used in a multithreaded environment) is called lock-free if it is synchronized by using a lock-free technique. Most modern hardware architectures provide synchronization primitives such as *compare-and-swap* (CAS) or *load-*

linked/store-conditional (LL/SC) which are strong enough for implementing basic concurrent data structures.

However, even a lock-free linked list requires an equivalent of an atomically markable reference, i.e. a pointer that can be atomically marked [5, 7]. Unlike Java 1.5+ [6], whose API provides the class `AtomicMarkableReference`, this feature is not supported by .NET 2.0 and 3.0 [2, 1]. Furthermore, the .NET API does not provide a support for any kind of (lock-free) synchronized data structure.

The main goal of this paper is comparison of the behavior of lock-free stack [10] and queue implementations [9] in Java 1.5 and .NET 2.0. We report and analyze the measured running times as well as the number of performed CAS operations with respect to the number of threads accessing each data structure concurrently.

2. Compare-and-swap

Lock-free data structures we deal with in this paper use compare-and-swap (CAS) synchronization primitive. CAS takes three parameters: an address of a memory location, an expected value and a new value. The new value is atomically written to the given memory location only if the content of the location equals to the expected value (Figure 1).

Both Java (since version 1.5) and .NET (since version 1.0) have an API support for CAS. In Java, it is provided via the `compareAndSet()` instance method of the class `AtomicReference` [3, 6], while .NET [2] provides it through the `CompareExchange()` static method of the class `Interlocked`. Since

```

bool CAS(addr, expval, newval) {
    atomic {
        if (*addr == expval) {
            *addr = newval;
            return true;
        }
        else
            return false;
    }
}

```

Figure 1. Compare-and-swap (CAS)

CAS in Java is implemented as an instance method (the only call mechanism supported by Java is call-by-value [4]), objects that CAS will be called on have to be wrapped in AtomicReference instances.

3. Lock-free Stack

The lock-free stack we consider in this paper is based on the straightforward approach first proposed by Treiber in [10]. Here the stack of elements of type T is represented as a singly-linked list of nodes of type $\text{Node}\langle T \rangle$:

```

class Node<T> {
    T item;
    Node<T>* next;

    Node(T t) {
        item = t;
        next = null;
    }
}

```

The top of the stack is denoted by a pointer Head to the first element in the list:

```
Node<T>* Head;
```

The presented code is written by using the C++ syntax (as it is more concise than Java or C#), however, we omit unnecessary technical details such as access specifiers or **template** keywords.

Figures 2 and 3 show the implementation of push and pop operations of the lock-free stack, respectively. Note that as pushing on and popping off the stack correspond to swinging the Head pointer back and forth, in an optimistic scenario (that is, under negligible thread contention) both operations succeed after a single invocation of CAS.

```

Push(T item) {
    Node<T>* oldHead;
    Node<T>* newHead =
        new Node<T>(item);
    do {
        oldHead = Head;
        newHead->next = oldHead;
    } while
        (!CAS(&Head, oldHead, newHead));
}

```

Figure 2. Push operation of the lock-free stack

```

T Pop() {
    Node<T>* oldHead;
    Node<T>* newHead;
    do {
        oldHead = Head;
        if (oldHead == null)
            return null;
        newHead = oldHead->next;
    } while
        (!CAS(&Head, oldHead, newHead));
    return oldHead->item;
}

```

Figure 3. Pop operation of the lock-free stack

4. Lock-free Queue

The lock-free queue employs the cooperative approach introduced by Michael and Scott in [9]. Here the queue is also implemented as a singly-linked list of nodes of type $\text{Node}\langle T \rangle$. It is accessed through two pointers Head and Tail :

```
Node<T>* Head;
Node<T>* Tail;
```

The Head pointer points to the first node in the list, which is a dummy node (not only to simplify the special cases of empty and single-item queues, but also to allow more concurrency). In states in which no thread is accessing the queue, the Tail pointer points to the last node in the list. Initially, we have:

```
Head = Tail = new Node<T>();
```

Figure 4 shows the implementation of the enqueue operation of the lock-free queue. In order to insert a node at the end of the list, it has to link the next pointer of the last

```

Enqueue(T item) {
    Node<T>* oldTail;
    Node<T>* oldTailNext;
    Node<T>* newTail =
        new Node<T>(item);
    while (true) {
        oldTail = Tail;
        oldTailNext = Tail->next;
        if (oldTail == Tail) {
            if (oldTailNext == null) {
                if (CAS(&Tail->next, null,
                    newTail))
                    break;
            }
            else
                CAS(&Tail, oldTail,
                    oldTailNext);
        }
    }
    CAS(&Tail, oldTail, newTail);
}

```

Figure 4. Enqueue operation of the lock-free queue

```

T Dequeue() {
    Node<T>* oldHead;
    Node<T>* oldHeadNext;
    Node<T>* oldTail;
    while (true) {
        oldHead = Head;
        oldHeadNext = Head->next;
        oldTail = Tail;
        if (oldHead == Head) {
            if (oldHead == oldTail) {
                if (oldHeadNext == null)
                    return null;
                CAS(&Tail, oldTail,
                    oldHeadNext);
            }
            else
                if (CAS(&Head, oldHead,
                    oldHeadNext))
                    break;
        }
    }
    return oldHeadNext->item;
}

```

Figure 5. Dequeue operation of the lock-free queue

node to the new node and swing the Tail pointer to it. In an optimistic scenario, this process requires two independent CAS operations. In addition, the presented code helps

other threads by swinging the Tail to the next node in case when Tail is found out not to point to the last node.

The implementation of the dequeue operation of the lock-free queue is shown in Figure 5. It removes a node from the beginning of the list by swinging the Head pointer to the next node which requires a single CAS operation in an optimistic scenario. In addition, it advances the Tail pointer if the Tail is falling behind, assisting in that way other threads.

5. Performance Evaluation

The main goal of this paper is to compare the behavior of Java and .NET implementations of the lock-free stack and queue with respect to the number of threads accessing each data structure concurrently. The employed versions of virtual machines were 1.5.0_10 in the case of Java and 2.0 in the case of .NET. Both virtual machines were run under default set of parameter settings.

From the user perspective, the far most important factor is the speed, i.e. the additional overhead in running time arising from the access by multiple threads. However, this is only a shallow measure as it is influenced by many parameters that are hard to estimate and control such as the behavior of the thread scheduler or the cost of context switching between threads.

Therefore, we also count the number of performed CAS operations since it gives us an independent measure that can be compared to the theoretical lower bounds from previous sections. It can also be used to predict behavior of the implementation on other platforms (when knowing the cost of performing a single CAS operation).

5.1. Description of the Benchmark

The evaluation benchmark that we have employed spawns a number of threads each repeatedly performing an addition immediately followed by a removal of an element from a shared instance of the lock-free data structure. That is, each thread executes a designated number of invocations of Push() followed by Pop() in case of the lock-free

stack, and Enqueue() followed by Dequeue() in case of the lock-free queue. The amount of work assigned to threads is divided evenly so that the total number of performed operations remains the same regardless of the number of spawned threads. The objective of such benchmark is to capture the behavior of the lock-free stack and queue under heavy load that combines usage of both types of operations.

5.2. Evaluation Results

The benchmark has been experimentally evaluated on two different machines — one with a single-core Intel Pentium M 1.5GHz CPU, and the other one with a dual-core Intel Pentium D 3GHz CPU. The obtained results for the single-core CPU are presented in Tables 1 and 2, while Tables 3 and 4 show the results for the dual-core CPU. Each table contains average values resulted from 100 executions of the benchmark.

The number of performed CAS operations on the single-core CPU equals to or is very close to the theoretical minimum. Table 2 indicates that in rare cases a thread performing an operation on a lock-free data structure might be switched inside a loop body such that after it is switched back, the invocation of CAS fails. Both Tables 1 and 2 show that .NET implementation outperforms Java implementation in the actual running time. Since the cost of CAS on a single-core platform is negligible, the observed discrepancy is most likely due to differences in thread scheduling. Namely, smaller time slices are assigned to threads in Java than in .NET which results in higher thread contention and more time being spent on context switching.

Tables 3 and 4 show that on the dual-core CPU the discrepancy in running times is somewhat smaller due to significant cost of CAS on this hardware platform. The influence of the CAS cost becomes even more evident if we compare the number of performed CAS operations with the corresponding running times. Further, Table 4 evidences that differences in thread scheduling cause running times for Java not to increase regularly with respect to the number of threads.

6. Conclusions and Future Work

Lock-free techniques often offer better and more scalable concurrent performance than lock-based techniques. Experimental results from the literature strongly support this statement for various concurrent data structures implemented in Java or C/C++ (e.g. [5, 7]). This paper presented experimental results for Java 1.5 and .NET 2.0 implementations of lock-free stack and queue. To the best of the authors knowledge this seems to be the first such report in the literature. We hope that it will provide a helpful insight into which platform offers a potentially better concurrent performance.

We plan to investigate more complicated lock-free data structures in the future (although .NET does not have a support for atomically markable reference, it is possible to implement it outside the managed code). Also, the performance evaluation deserves a more careful approach. Namely, most implementations of lock-free data structures (in particular, the ones in this paper) rely on correct garbage collecting mechanism in order to avoid the ABA problem [7]. Our performance evaluation tried to neglect the influence of the memory allocator as much as possible. However, more plausible results should be expected from the implementation that uses lock-free memory allocation [8].

References

- [1] .NET Framework web page:
<http://msdn.microsoft.com/netframework/>.
- [2] Common Language Infrastructure (CLI), Standard ECMA-335, 4th edition. Ecma International, 2006. Available on the web at: <http://www.ecma-international.org/publications/standards/Ecma-335.htm>.
- [3] B. Goetz, J. Bloch, J. Bowbeer, D. Lea, D. Holmes, T. Peierls. Java Concurrency in Practice. Addison–Wesley, 2006.
- [4] J. Gosling, B. Joy, G. L. Steele, G. Bracha. The Java Language Specification, Third Edition. Addison–Wesley, 2005.
- [5] S. Heller, M. Herlihy, V. Luchangco, M. Moir, W. N. Scherer, N. Shavit. A Lazy Concurrent List-Based Set Algorithm. In Pro-

Table 1. Averaged results obtained for the lock-free stack on a single-core 1.5GHz CPU

# Threads	# Performed CAS operations		Running times [s]	
	Java	.NET	Java	.NET
1	2000000	2000000	0.084	0.116
2	2000000	2000000	0.213	0.116
3	2000000	2000000	0.277	0.115
4	2000000	2000000	0.395	0.115
5	2000000	2000000	0.456	0.115
6	2000000	2000000	0.507	0.116
7	2000000	2000000	0.554	0.115
8	2000000	2000000	0.630	0.114
9	2000000	2000000	0.684	0.113
10	2000000	2000000	0.764	0.114
11	2000000	2000000	0.816	0.114
12	2000000	2000000	0.913	0.113
13	2000000	2000000	0.910	0.112
14	2000000	2000000	0.945	0.113
15	2000000	2000000	1.043	0.112
16	2000000	2000000	1.057	0.111

Table 2. Averaged results obtained for the lock-free queue on a single-core 1.5GHz CPU

# Threads	# Performed CAS operations		Running times [s]	
	Java	.NET	Java	.NET
1	3000000	3000000	1.205	0.194
2	3000003	3000000	2.565	0.189
3	3000006	3000000	3.755	0.188
4	3000007	3000000	4.994	0.187
5	3000005	3000000	5.924	0.186
6	3000005	3000000	6.929	0.186
7	3000004	3000000	7.752	0.185
8	3000004	3000000	8.507	0.186
9	3000003	3000000	9.589	0.184
10	3000005	3000000	10.252	0.183
11	3000004	3000000	10.853	0.183
12	3000004	3000000	11.652	0.183
13	3000004	3000000	12.275	0.182
14	3000004	3000000	12.444	0.181
15	3000003	3000000	13.526	0.181
16	3000004	3000000	13.936	0.181

ceedings of the 9th International Conference on Principles of Distributed Systems, Lecture Notes in Computer Science 3974. Springer, 2006, pp. 3–16.

[6] Java Platform, Standard Edition web page: <http://java.sun.com/javase/>.

[7] M. M. Michael. High performance dynamic

lock-free hash tables and list-based sets. In Proceedings of the 14th ACM Symposium on Parallel algorithms and architectures. ACM Press, 2002, p. 73-82.

[8] M. M. Michael. Scalable Lock-free Dynamic Memory Allocation. In Proceedings of the ACM SIGPLAN 2004 Conference on Pro-

Table 3. Averaged results obtained for the lock-free stack on a dual-core 3GHz CPU

Number of threads	Number of performed CAS operations			Running times [s]		
	Java	.NET	.NET with thread affinity	Java	.NET	.NET with thread affinity
1	2000000	2000000	2000000	0.185	0.156	0.156
2	3605347	2311542	2672216	1.264	0.441	0.484
3	3428966	2846468	2762855	1.477	0.900	0.901
4	3513919	2800791	2879338	1.748	1.001	0.964
5	3575533	2643090	2683091	1.966	0.887	0.848
6	3638797	2747360	2813381	2.227	0.905	0.723
7	3616644	2782823	2875511	2.108	0.865	0.932
8	3630769	2919585	2903983	2.109	0.988	0.957
9	3599000	2885644	2906914	2.275	0.907	0.885
10	3648616	2777645	2822346	2.361	0.885	0.837
11	3645474	2930689	3028974	2.423	1.042	0.945
12	3610170	3010584	3008117	2.533	1.015	0.936
13	3599974	2941086	2985901	2.637	0.974	0.906
14	3613443	2861877	2933356	2.607	0.957	0.847
15	3569441	2976860	2983662	2.712	0.993	0.939
16	3648974	3034455	3034355	2.816	1.001	0.921

Table 4. Averaged results obtained for the lock-free queue on a dual-core 3GHz CPU

Number of threads	Number of performed CAS operations			Running times [s]		
	Java	.NET	.NET with thread affinity	Java	.NET	.NET with thread affinity
1	3000000	3000000	3000000	1.677	0.307	0.29
2	3293252	3411673	3288229	4.791	0.622	0.621
3	3313894	3419309	3360516	3.225	1.264	1.278
4	3350876	3367342	3373534	7.226	1.124	1.287
5	3358748	3449534	3372311	6.494	0.971	0.959
6	3377674	3465975	3382097	7.747	1.015	0.996
7	3373808	3419107	3395008	8.926	1.198	1.208
8	3380638	3443109	3427798	13.198	1.213	1.251
9	3387511	3456496	3436536	8.224	1.113	1.095
10	3384314	3445379	3430880	5.112	1.095	1.208
11	3391326	3451168	3468368	4.790	1.215	1.233
12	3376152	3483615	3426337	4.882	1.283	1.269
13	3361296	3488519	3461091	4.998	1.153	1.143
14	3349865	3447999	3464166	4.973	1.171	1.203
15	3342741	3494409	3476753	4.910	1.265	1.238
16	3341760	3451453	3447125	5.045	1.299	1.282

gramming Language Design and Implementation. ACM Press, 2004, p. 35–46.

the 15th ACM Symp. on Principles of Distributed Computing. ACM, 1996, p. 267-275.

[9] M. M. Michael, M. L. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In Proc. of

[10] R. K. Treiber. Systems programming: Coping with parallelism. Research report RJ 5118, IBM Almaden Research Center, 1986.