Benchmarking Idempotent Work Stealing

Günther Eder
geder@cosy.sbg.ac.at

Prof. Christoph Kirsch
Supervisor

University of Salzburg
Department of Computerscience
Jakob-Haringer-Strasse 2
5020 Salzburg
Austria

16. 04. 2012

Abstract

We study the trade-off between semantical relaxation, performance, and scalability of work-stealing stack algorithms. In particular, we analyze four different versions of which two implement regular work-stealing stacks [1] and the other two implement semantically relaxed variants of work-stealing stacks [3]. In the benchmarks we descovered that increasing semantical relaxation generally leads to increasing performance and scalability.

1 Introduction

1.1 Introduction

We show the performance and scalability of four different work stealing stack algorithms. Two implement regular semantics [1] and the other two implement semantically relaxed variants [3] of work stealing stacks. We discuss the trade-off of this LIFO implementations while dealing with balanced and unbalanced workloads. To improve performance of a data structure which works in a highly parallel environment we weaken the semantics of a regular work stealing stack algorithm. This means we let the implementation deviate from the standard work stealing stack implementation [3]. There are different variations of semantical deviation [5]. Our approach is to allow elements on the stack to be returned more then once [3]. This allows us to use less synchronization.

1.2 Approach

In the following section the different algorithms, their drawbacks and benefits will be described closer. We discuss the implementation of the chosen algorithms, the benchmark and different problems that occoured during this process. At last we take a look at the benchmark results and compare the different algorithms.

2 Algorithms

Four algorithms are tested in this work, two are state of the art and already well known. The other two deviate from the regular work-stealing semantics.

2.1 Standard Work Stealing

Each thread has its own local stack (Figure 1) on which it can apply two operations: *put()* and *take()*. A third operation *steal()* can be applied to the stacks of the other threads.

The Work Stealing idea: if a thread is finished with its local stack and has capacity to work, it should invoke *steal()* on another threads stack to balance the workload.

The semantics of the Standard Work Stealing implementation are strict which means it is a common LIFO stack. Each element pushed with *put()* onto the stack, taken from it with *take()* or *steal()* uses a Compare and Swap (Listing 2) operation. We will refer to Compare and Swap 2 as CAS in this work. CAS is an atomic operation that guarantees the correct state of the stack.

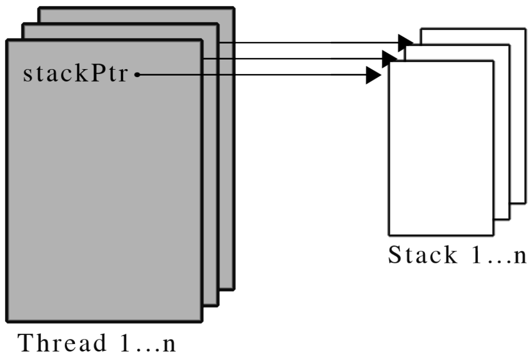

Figure 1: Work Stealing Setup

2.2 Chase-Lev Workstealing

The Chase-Lev [1] work stealing algorithm is state of the art and one of the fastest implementations [3] of this data structure type. Chase-Lev still is a common LIFO data structure and does not deviate from the regular Work Stealing semantics, but it is designed to use less synchronization.

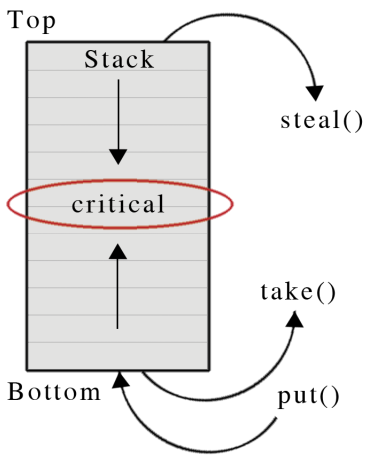

Figure 2: Chase-Lev Stack

In Figure 2 we see the *put()* and *take()* operating on bottom of the threads local stack.

This seems uncommon since a regular stack operates with push and pull on its top. In this case it is actually only a formal deviation and results in the same semantics as a regular stack. The *steal()* operation on the other hand is accessing the stack on top. Processing both sides is possible because the Chase-Lev algorithm is designed to use an array as stack.

Since the owner thread is the only one which can push data onto the stack, *put()* requires no synchronization. Furthermore *take()* requires synchronization only if one element is left on the stack (Figure 2, red circle). In that case, it could occur that a *steal()* and *take()* operation happen at the same time, which would result in returning one element twice. The *steal()* operation uses synchronization in every case, except the stack is empty.

This approach performes well, since the workload on the local stack is the more important one. The higher effort for *steal()* is a small drawback.

2.3 Idempotent Work Stealing

The Idempotent Work Stealing [3] algorithm uses semantical weakening [5]. In this case, it is possible to get the same object more than once from the stack. This deviation from the regular semantics allow Idempotent Work Stealing [3] to use synchronization only in the *steal()* operation. The *put()* and *take()* operations are free from atomics. All three operations access the stack from top as we would suspect from a regular stack.

This lack of synchronization promises improvement in scalability and performance.

2.3.1 Semantical Deviation

We define semantical deviation [5] in the amount of times an object can be returned from the stack. By this definition the Idempotent Work Stealing algorithm has a worst-case semantical deviation of n-1 (n dependent on the stack length). This means that in the worst case the same element is returned as often as the amount of elements on the stack, minus one.

2.4 WCSD-Low

This new variation of the Idempotent Work Stealing [3] algorithm has the possibility of a lower worst case semantical deviation or *WCSD*. This bound is lower or equal to the semantical deviation of Idempotent Work Stealing [3], so we call it WCSD-Low.

```
Structures:
  Task: task information, removed flag
  LifoIwsq:
  anchor: <integer,integer>; // <tail,tag>
  capacity: integer
  tasks: array of Task

constructor LifoIwsq(integer size) {
  anchor := <0,0>;
  capacity := size;
  tasks := new Task[size];
}
```

```
void put(Task task) {
  Order write in 3 before write in 4

1 <t,g> := anchor;
2 if (t = capacity) {expand(); goto 1;}
3 tasks[t] := task;
4 anchor := <t+1,g+1>;

}


TaskInfo take() {

1 <t,g> := anchor;
2 while (t > 0 && tasks[t].removed == true) t−−;
3 if (t=0) return EMPTY;
4 task := tasks[t−1];
5 anchor := <t−1,g>;
6 return task;

}

TaskInfo steal() {
  Order read in 1 before read in 4
  Order read in 5 before CAS in 6

1 <t,g> := anchor;
2 while (t > 0 && tasks[t].removed == true) t−−;
3 if (t=0) return EMPTY;
4 a := tasks;
5 task := a[t−1];
6 if !CAS(anchor,<t,g>,<t−1,g>) goto 1;
7 task.removed = true;
8 return task;

}
```

Listing 1: Pseudo Code WCSD-Low

The only change is an extra attribute (*removed*) to keep track, which elements were already removed from the stack. Checking this attribute is done in the *take()* and *steal()* operation on line 2. Setting it, is only conducted on line 7 in the *steal()* function. In the *steal()* operation we use an atomic CAS to synchronize with any other stack.

A semantical deviation can only occur if a *take()* and *steal()* operation is in progress at the same time. Since *take()* only accesses the own stack, it can happen only once at a time and of course an arbitrary amount of *steal()* operations. To this point the algorithm is identical to Idempotent Work Stealing [3].

Our new variation WCSD-Low is built under following assumption:

- In a situation described above, a *steal()* operation will execute line 7 after the CAS on line 6 (Listing 1).
- On the next execution of *take()* it will skip this element which leads to a lower semantical deviation as the Idempotent Work Stealing [3] algorithm has.

3 Implementation

In this section we will see details and some problems concerning the implementation of this benchmark. The benchmark itself is written in *C*. Libraries needed in order to be able to compile this project: the math library to calculate the results and the pthread library to create threads and barriers.

3.1 Synchronisation

To have a better understanding of the costs of synchronization mechanisms like CAS (e.g. *cmgxchg* on Intel based systems), we implemented a benchmark which shows this overhead. The CAS semantics is shown in Listing 2. Our test setup is a set of threads, where all share one memory item. In this case we use a simple integer variable. Each thread tries to access this shared variable using read and write operations. To guarantee a correct state, this can only be accomplished safely by using atomic operations.

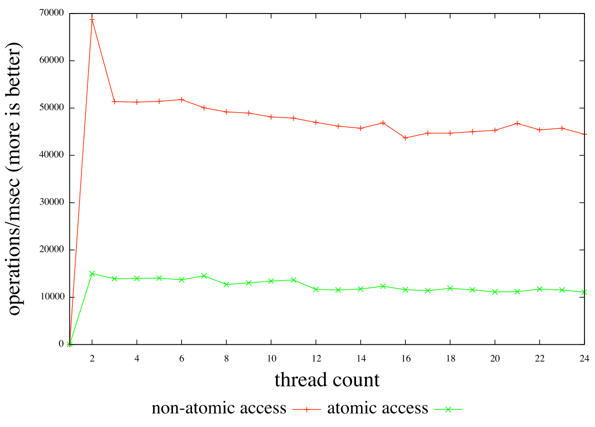

Figure 3: Shared Memory Access High Contention

The first test is accessing the shared memory non-atomically and the second test uses atomics, but both with the same semantics (Listing 2). Since we are only interested in the overhead that atomics cause, the correct state of the shared variable is less important.

In Figure 3 we can see a red graph which shows the non-atomic access on the shared memory and a green graph which pictures the atomic access. We see that the overhead for synchronization in this basic case is about a factor of three, not depending on how many threads we use.

```
function CAS(pointer,old,new){
    if ∗pointer == old then
        ∗pointer := new
        return true
    else
        return false
}
```

Listing 2: Compare and Swap [2]

3.2 Barriers

In order to start all threads at once we use a pthread barrier. This barrier implementation is provided by the POSIX pthread library.

Earlier versions of this benchmark where using busy waiting loops to synchronize the threads after creation. This seemed to be slower in some test-cases then the pthread barrier. It is possible that this is scheduler-dependent, but was not testet and is left for future work.

3.3 Time Measurement

The *rdtsc()* instruction returns number of cycles since the start of the system. In order to calculate the time consumption we first have to determine the CPU frequence. We use the *sleep(1)* command and substract the *rdtsc()* values before and after from each other. Since *sleep(1)* will intercept the program for one second, the subtraction will give us the number of cycles completed in one second (Hz).

4 Experiments

In this section we show the results of our benchmark. We start with the balanced benchmarks where each thread has the same amount of elements on the stack, then we show the results for the unbalanced variants.

This results are depending on the work performed between the pulls, so we show different benchmark results each using a different amount of work.

Since the allocation for 24 arrays with the size of 200 million took too long, we reduced to the size of 30 million elements per array. But we increased the runs per test up to 15 for better over all test results.

4.1 Workload

The work a thread performs until it pulls the next element from a stack we refer to as workload. Simulating this effort is not trivial, so we used different amounts of work to get insight how this effects the scalability.

One degree of workload in our benchmark equals three integer multiplications performed in a loop. In order to get a higher contention on the data structure we use less workload, as shown in Table 1.

contention	workload
maximum	no workload
medium	degree one (three integer multiplications)
low	degree three (nine integer multiplications)

Table 1: Contention & Workload

4.2 Memory Management

We test the stack implementation up to 200 million elements per stack, which gives us a total of 16 billion elements on 24 stacks, which means 38.4 GByte on a 64 bit architecture only to allocate the arrays. To be able to handle this amount of data elements only one was element allocated and reused for each stack position. The allocation of the huge arrays takes up to 40 seconds.

An approach for future work would be to extend the benchmark so it can run without exiting after every configuration, this would lower the allocation overhead.

4.3 Test Environment

For the test we use an Intel Xeon E7 4850 with 4 CPUs 2.0 GHz each, 10 cores/cpu and 2 threads/core which gives possible 80 threads by 128 GB Ram running Ubuntu 10.04.3 with Linux kernel 2.6.32.

4.4 Balanced Stack

In Figure 4 we see that the Idempotent Work Stealing [3] algorithm still performs best under maximum contention but the new variation WCSD-Low is close behind. Maximum (Table 1) means that there is no work effort taking place between each *take()* operation.

When the contention on the stack decreases, the performance of all algorithms becomes more similar, as expected. If the workload between each *take()* or *steal()* increases, the synchronization is taking less effect on the result, as we can see in Figure 5 and Figure 6.

4.5 Unbalanced Stack

Figure 7 shows the four algorithms under maximum (Table 1) contention.

For this benchmark we used a different stack length for each thread, which could vary upt to 10 percent.

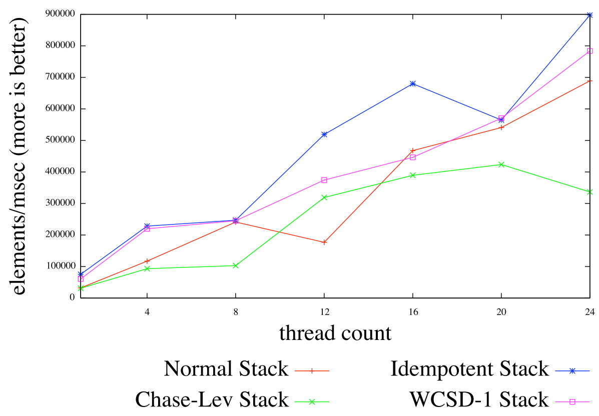

Figure 4: Balanced Stack with maximum contention

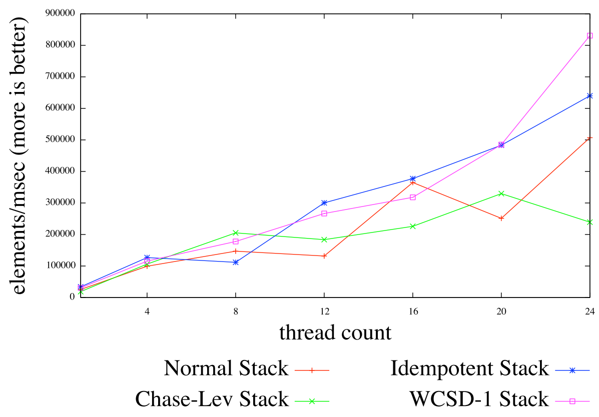

Figure 5: Balanced Stack with medium contention

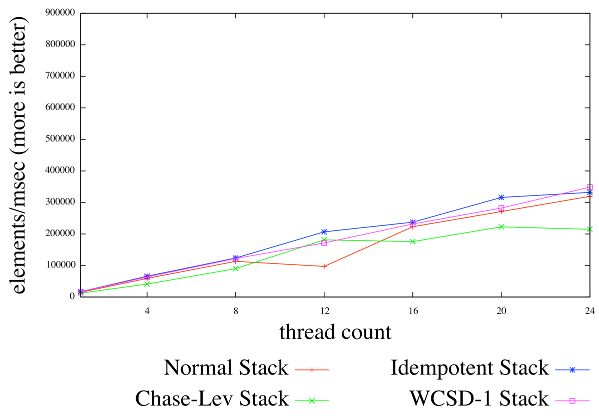

Figure 6: Balanced Stack with low contention

Again the Idempotent Workstealing algorithm [3] performs best. Our results show positive scalability only until 16 to 20 threads under maximum contention.

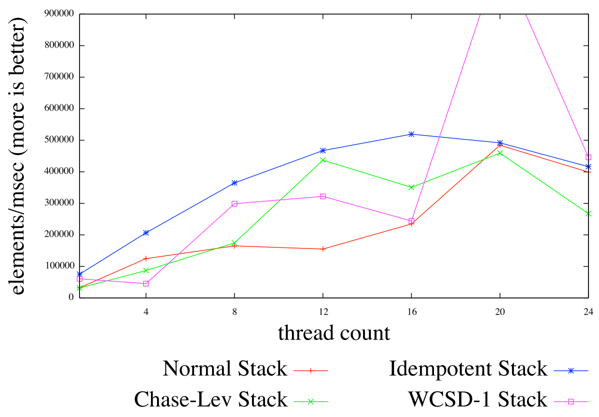

Figure 7: Unbalanced Stack with maximum contention

In Figure 8 we see that under less contention the scalability is better. Also the performance for a higher tread count is better then with maximum contention.

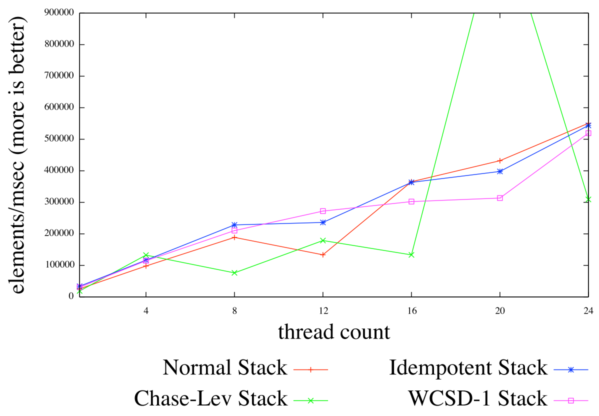

Figure 8: Unbalanced Stack with medium contention

Like in the balanced tests (Figure 6) we can see the same effect in Figure 9 taking place. As the contention declines, the results get more similar to each other.

5 Conclusion

Our benchmark shows the performance is dependent on the workload. If the workload is increasing the performance of the data structure decreases and the type of data structure used is less important.

We see the scalability for all variants, balanced and unbalanced workload is positive. Only when using high

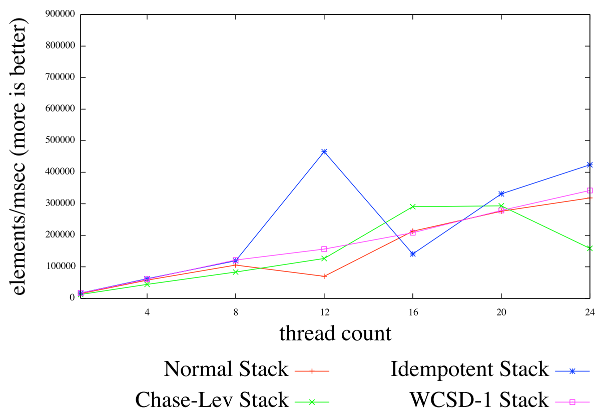

Figure 9: Unbalanced Stack with low contention

or maximum contention the Idempotent Work Stealing [3] can outperform the other implementations.

If a data structure is needed which provides high performance and positive scalability while used under high contention, our tests show that the Idempotent Work Stealing [3] algorithm will be the best choice if the semantical deviation is no concern for the task.

6 Acknowledgement

Thanks to Hannes Payer for supplying me with this new variation WCSD-Low and supporting me in this project.

List of Figures

References

[1] David Chase and Yossi Lev. Dynamic circular work-stealing deque. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, SPAA '05, pages 21–28, New York, NY, USA, 2005. ACM.

[2] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13:124–149, January 1991.

[3] Maged M. Michael, Martin T. Vechev, and Vijay A. Saraswat. Idempotent work stealing. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 45–54, New York, NY, USA, 2009. ACM.

[4] Hannes Payer, Harald Roeck, Christoph M. Kirsch, and Ana Sokolova. Performance, scalability, and semantics of concurrent fifo queues. Technical report, Department of Computer Sciences, Jakob-Haringer-Strasse 2, 5020 Salzburg, Austria, 2011.

[5] Hannes Payer, Harald Roeck, Christoph M. Kirsch, and Ana Sokolova. Scalability versus semantics of concurrent fifo queues. In *Proceedings of the 30th annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, PODC '11, pages 331–332, New York, NY, USA, 2011. ACM.