Brushing the Locks out of the Fur: A Lock-Free Work Stealing Library Based on Wool

Håkan Sundell
School of Business and Informatics
University of Borås, 501 90 Borås
E-mail: Hakan.Sundell@hb.se

Philippas Tsigas
Department of Computer Science and Engineering
Chalmers University of Technology, 412 96 Göteborg
E-mail: tsigas@chalmers.se

Abstract

We present a lock-free version of the light-weight user-level task management library called Wool, in an aim to show that even extremely well tuned, in terms of synchronization, applications can benefit from lock-free programming. Explicit multi-threading is an efficient way to exploit the offered parallelism of multi-core and multi-processor based systems. However, it can sometimes be hard to express the inherited parallelism in programs using a limited number of long lived threads. Often it can be more straightforward to dynamically create a large number of small tasks that in turn automatically execute on the available threads. Wool is a promising and efficient library and framework that allows the programmer to create user tasks in C with a very low overhead. The library automatically executes tasks and balances the load evenly on a given number of threads by utilizing work stealing techniques. However, the synchronization for stealing tasks is based on mutual exclusion which is known to limit parallelism and efficiency. We have designed and implemented a new lock-free algorithm for synchronization of stealing tasks in Wool. Experiments show similar or significantly improved performance on a set of benchmarks executed on a multi-core platform.

1. Introduction

Explicit multi-threading is an efficient way to exploit the offered parallelism of multi-core and multi-processor based systems. However, it can sometimes be hard to express the inherited parallelism in programs using a limited number of long lived threads. Often it can be more straightforward to dynamically create a large number of small tasks that in turn automatically execute on the available threads. See Figure 1 for an example of how this kind of task parallelism can be expressed for computing Fibonacci numbers using the Wool [3] library.

```
#include <stdio.h>
#include <stdlib.h>
#include "wool.h"

TASK_1( int, fib, int, n )
{
    if( n<2 ) return n;
    else {
        int a,b;
        SPAWN( fib, n-2 );
        a = CALL( fib, n-1 );
        b = SYNC( fib );
        return a+b;
    }
}

TASK_2( int, main, int, argc,
char **, argv )
{
    printf( "%d\n", CALL( fib, atoi( argv[1] ) )
        );
}
```

Figure 1. A simple Fibonacci function defined using the Wool library.

Fundamental to multi-threading is the ability to share data among the threads. To avoid inconsistency of the shared data due to concurrent modifications, accesses to the shared data must be synchronized and the common solution using mutual exclusion is known for carrying several serious problems. The alternative of using non-blocking synchronization can avoid these problems and have been shown to permit substantial performance improvement of parallel applications[7]. Two basic non-blocking methods have been proposed in the literature; *lock-free* and *wait-free* [5].

One of the first implementations of a light-weight task management system was the Cilk-5 framework introduced by Frigo et al. [4]. This framework keeps a task queue (ac-

tually stack) for each thread, from where other threads can *steal* tasks by concurrently removing tasks from the other end of the data structure. This *work-stealing deque* data structure has then been improved by using lock-free synchronization by Arora et al. [1], Chase and Lev [2] and several others later. By limiting to only allow tasks that can be executed several times (i.e., having no side effects) this approach has been further improved by Michael et al. [6]. Wool by Faxén [3] is a promising task management library that achieves a substantial improvement compared to the deque approach, while still allowing tasks with side effects, by collapsing the data structure layers and instead synchronize the stealing directly on each individual task. The synchronization used in Wool is lock-based, although highly optimized to allow for maximal concurrency.

We present a new lock-free algorithm for the synchronization in Wool, in an aim to show that even extremely well tuned lock-based applications can benefit from lock-free programming.

2. The New Lock-Free Algorithm

In Wool [3] the data structures for describing and organizing the work load are consisting of `Workers` and `Tasks`. Each `Worker` is representing a specific thread. The queue of work to do, added by this thread, is represented by an array of `Tasks`. Normally, this queue is only used by its worker, but when a worker is idle it will try to take work also from other worker's queues. This concurrent *stealing* needs synchronization and the common solution (e.g., in [1] [2] [6]) is to synchronize through operations on the queue.

In Wool, thieves and victims instead synchronize through the task descriptors in the task queue of the victim. A `Task` contains the following information:

- A field f describing the function that executes the task, where values are either a pointer to the function or `INLINED`.
- A field balarm indicating whether the task has been stolen, where values are either `READY`, `STOLEN` or `DONE`.
- The arguments the task was spawned with.
- The return value of the task (shares space with the arguments).

For keeping track of the currently steal-able tasks, each `Worker` descriptor keeps a field bot that points to the first task in the array that could possibly be stolen. In Wool, synchronization between thieves as well as between thieves and a victim are handled using mutual exclusion, see Figure 2 for pseudo-code description of the synchronization part of

```
1  bool steal( Worker *victim )
2  {
3      lock( victim->lck );
4      Task *t = victim->bot;
5      t->balarm = STOLEN;
6      memory_barrier();
7      if( t->f == INLINED ) {
8          unlock( victim->lck );
9          t->balarm = READY;
10         return false;
11     } else {
12         victim->bot++;
13         unlock( victim->lck )
14         ... // Run the task
15         memory_barrier();
16         t->balarm = DONE;
17         return true;
18     }
19 }
20
21 void sync( Task *t )
22 {
23     t->f = INLINED;
24     memory_barrier();
25     if( t->balarm != READY ) {
26         // Wait for thief to fully decide
27         lock( self->lck );
28         if( t->balarm == READY ) {
29             unlock( self->lck );
30             ... // Run the task
31         } else {
32             unlock( self->lck );
33             ... // Wait for thief to finish
34             self->bot--;
35         }
36     }
37 }
```

Figure 2. The old lock-based algorithm for synchronization between thief and victim.

```
void FAA( int volatile *address, int number ) atomically do {
    *address = *address + number;
}
// also CAS( void * volatile *address, void * oldvalue, void *newvalue )
bool CAS( int volatile *address, int oldvalue, int newvalue ) atomically do {
    if( *address == oldvalue ) {
        *address = newvalue;
        return true;
    }
    else return false;
}
// also DWCAS( void * volatile *address, void * oldvalue1, void *oldvalue2, void *newvalue1, void *newvalue2 )
bool DWCAS( int volatile *address, int oldvalue1, int oldvalue2, int newvalue1, int newvalue2 ) atomically do {
    if( address[0] == oldvalue1 && address[1] == oldvalue2 ) {
        address[0] = newvalue1;
        address[1] = newvalue2;
        return true;
    }
    else return false;
}
```

Figure 3. The Fetch-And-Add (FAA) and Compare-And-Swap (CAS) atomic primitives.

```
bool steal( Worker *victim )
{
    Task *t = victim->bot;
    f = t->f;
    if( f != INLINED && DWCAS( &t->f, f, READY, f, STOLEN ) ) {
        FAA( &victim->bot, 1 );
        ... // Run the task
        memory_barrier();
        t->balarm = DONE;
        return true;
    }
    else return false;
}

void sync( Task *t )
{
    t->f = INLINED;
    memory_barrier();
    if( t->balarm == READY ) {
        ... // Run the task
    }
    else {
        ... // Wait for thief to finish
        FAA( &self->bot, -1 );
    }
}
```

Figure 4. The new lock-free algorithm for synchronization between thief and victim.

Wool. In order to optimize for the common case (*sync* is executed much more often than *steal*), the victim does not use locking:

- at all when modifying steal-able status (i.e., f).
- most of the times when checking for stealing.

However, due to the weak synchronization from the side of the victim (the thieves synchronize always using locks) it can happen that the thief, after it announces that a task is `STOLEN`, it recognizes that the task is no longer steal-able. This can happen in the case where f was set to `INLINED`, by the victim, concurrently with the announcement action by the thief. In order to take care of this scenario the victim needs to make sure that the thief has finished with its stealing decision before trusting the balarm stealing status. One way that this can be achieved is by making the victim to wait for the thief to release the lock, the way it is done in Wool.

In order to create a lock-free algorithm for the synchronization in Wool, we need to utilize atomic primitives available in the hardware for shared memory systems. Figure 3 describes the semantics of some operations for atomic updates that are commonly available on contemporary systems. Recently, several architectures (e.g., Intel x86 and x64) support atomic updates of also two adjacent memory words (e.g., 128 consecutive bits on a 64-bit word architecture).

By utilizing the atomic operations and a careful composition of the involved fields of the `Task` data structure, we have managed to design a lock-free scheme of the synchronization part needed in stealing. As in Wool, for performance reasons the victim uses weaker and faster atomic primitives than the thieves, and in addition the stealing status can now be fully trusted directly by the victim without any extra synchronization procedure.

By utilizing the *DWCAS* operation we can design a lock-free algorithm of the stealing synchronization, see Figure 4 for a description of the new algorithm. In the task descriptor, the field balarm is placed directly after the field f and can thus be updated together by the *steal* function. If the victim has decided to execute the task itself, the field f has been set to `INLINED` which is noted by the *DWCAS* operation that fails. At the same time, the *DWCAS* operation will fail if another thief already has managed to update the balarm field to `STOLEN`, thus guaranteeing that each task can only be stolen once.

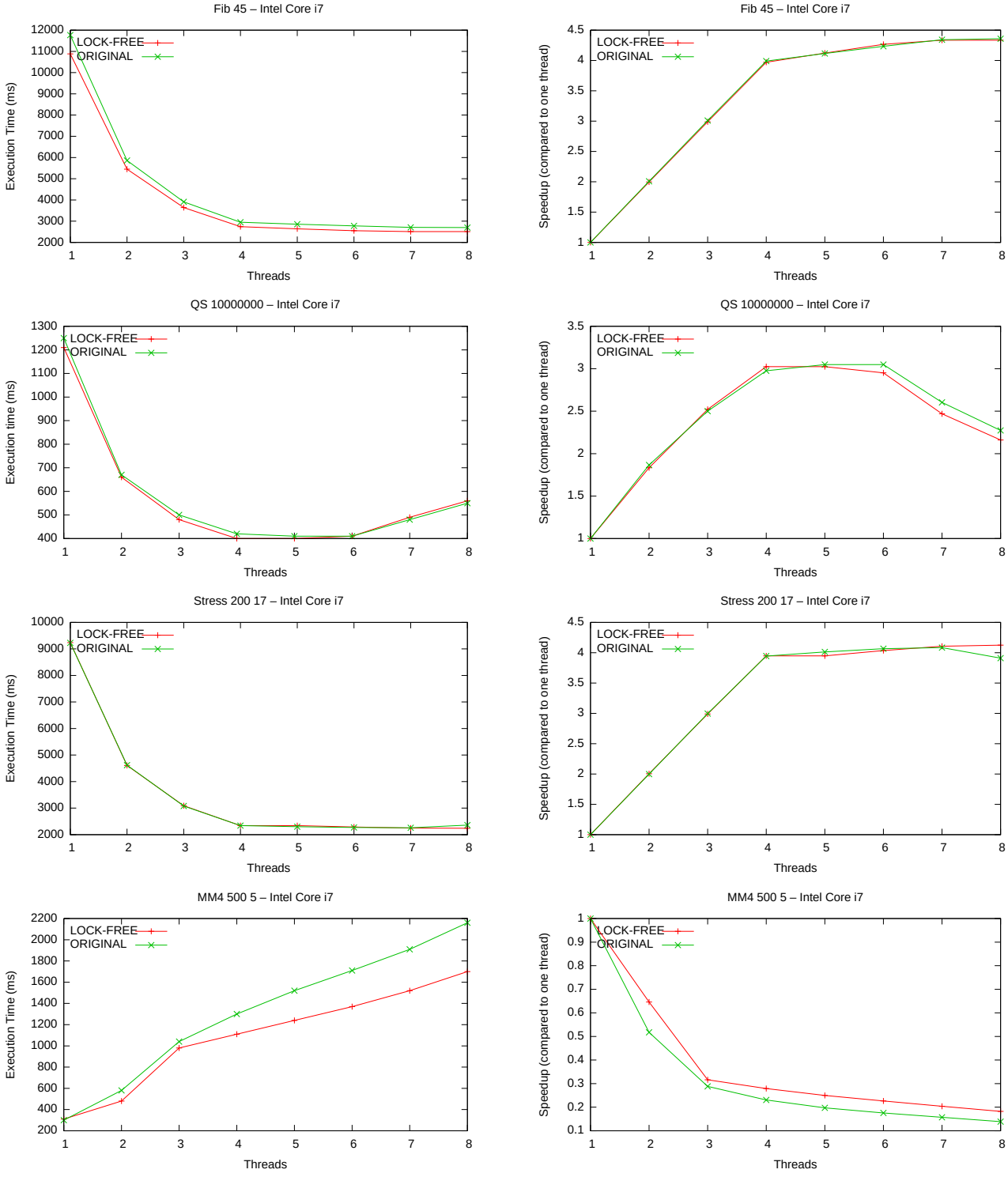

Figure 5. Experiments on a 8-way Intel Core i7 processor system.

3. Experiments

We have performed experiments on a contemporary multi-core platform in order to estimate the possible performance benefits of replacing the synchronization in Wool with the new lock-free one. For comparison, the same benchmarks as in [3] have been executed on an Intel Core i7950 3 GHz with 6 GB DDR3 1333 MHz system running Linux 2.6. This processor has 4 cores, capable of executing 2 threads each. The benchmarks used were Fibonacci (fib), Quicksort (qs) and Stress. In addition we also used a Matrix multiplication (mm4) benchmark. The results of the benchmarks are shown in Figure 5, with actual execution time to the left and speedups (relative to executing one thread of the corresponding implementation) to the right.

Apparently, benchmarks with low memory utilization (i.e., fib and stress) scale almost ideally along the cores, and benchmarks with higher memory utilization (i.e., qs) scales fairly well with exception from the mm4 benchmark. One explanation for the bad scaling is that the mm4 benchmark has a considerably higher fraction of steals than inline executions. Notably is that the lock-free implementation performs significantly better on the fib and mm4 benchmarks and the same performance as the original lock-based implementation on the other benchmarks.

4. Conclusions

We have designed a lock-free synchronization for the Wool library that has been introduced for light-weight task execution. By replacing the highly optimized lock-based synchronization in Wool with the new lock-free scheme, we get a faster and composable, thanks to not using blocking, work stealing scheme. Our experiments were conducted using a set of benchmarks executed on an Intel Core i7 platform. This work was introduced in order to verify our conjecture that even highly optimized lock-based synchronization schemes, like the one that the original Wool uses, can be further optimized using lock-free programming.

Interesting future work is to compare with other task management libraries and implementation on other architectures.

References

[1] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 119–129, 1998.

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

[3] K.-F. Faxén. Wool - a work stealing library. In *Proceedings of the First Swedish Workshop on Multi-Core Computing (MCC08)*, pages 117–124, 2008.

[4] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation (PLDI '98)*, pages 212–223, 1998.

[5] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 11(1):124–149, Jan. 1991.

[6] M. M. Michael, M. T. Vechev, and V. 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.

[7] P. Tsigas and Y. Zhang. Integrating non-blocking synchronisation in parallel applications: Performance advantages and methodologies. In *Proceedings of the 3rd ACM Workshop on Software and Performance*, pages 55–67. ACM Press, 2002.