Futex based locks for C11's generic atomics (extended abstract)

Jens Gustedt
INRIA and ICube, Université de Strasbourg, France

ABSTRACT

We present a new algorithm and implementation of a lock primitive that is based on Linux' native lock interface, the `futex` system call. It allows us to assemble compiler support for atomic data structures that can not be handled through specific hardware instructions. Such a tool is needed for C11's atomics interface because here an `_Atomic` qualification can be attached to almost any data type. Our lock data structure for that purpose meets very specific criteria concerning its field of operation and its performance. By that we are able to outperform gcc's `libatomic` library by around 60%.

CCS Concepts

•**Software and its engineering** → **Runtime environments;** *Concurrent programming structures;*

Keywords

lock primitives, atomics, C11, futex, Linux

1. INTRODUCTION

Only very recently (with C11, see JTC1/SC22/WG14) the C language has integrated threads and atomic operations into the core of the language. Support for these features is still partial: where the main open source compilers gcc and Clang now offer atomics, most Linux platforms still use glibc as their C library which does not implement C11 threads. Only platforms that are based on MUSL as C library, e.g Alpine, are feature complete.

The implementation of the C11 atomic interface typically sits in the middle between the implementation of the core language and the C library. It needs compiler support for the individual atomic operations and a *generic atomic lock* interface in the C library when no low-level atomic instruction is available.

Since Linux' open source C libraries do not implement the generic interface, the compilers currently provide a library stub that implements the necessary lock by means of a combination of atomics and POSIX `pthread_mutex_t`.

From a point of view of the C language the natural interface for that would be C11 threads, but since these are still missing in `glibc`, they fall back to POSIX threads, instead of mapping to platform specific properties and drawing the best performance from a given platform. In this work we present a specific algorithm for the generic lock that relies on a specific Linux utility, the `futex` system call.

A `futex` combines one atomic integer and OS scheduling. In our approach, we use one **unsigned** to implement the lock and a waiter count at the same time. The resulting data type of minimal size (32 bit on all architectures) and the algorithm can take advantage by minimizing the number of CPU to memory transfers. In most cases one such transfer is sufficient, where other algorithms have to update a lock and a waiter counter separately.

To our knowledge pursuing this approach to a complete solution is new. Previously urban myth had it that such approaches would risk deadlocks if confronted to heavy load, because repeated failures of calls to **futex_wait** could lead to unbounded loops. We are able to prove that such unbounded loops will not happen for our algorithm. Also, our measurements have shown that such an approach can be very effective: failed system calls to **futex_wait** are much less costly than commonly thought.

Our algorithm and its implementation is part of a larger open source project to provide the necessary interfaces (header files) and library support for C11's `<stdatomic.h>`. It is available at http://stdatomic.gforge.inria.fr/. The code is functional to be used with `gcc` and `clang`, even for older version without full support for atomic operations. In a later stage, we intent to integrate the whole project into the `musl` C library. A full version of this paper, Gustedt (2015), is available online as https://hal.inria.fr/hal-01236734.

2. TOOLS FOR DATA CONSISTENCY AND RACES

Data races are the most difficult challenge for parallel pro-

SAC 2016,April 04 - 08, 2016, Pisa, Italy

ACM 978-1-4503-3739-7/16/04…$15.00
DOI: http://dx.doi.org/10.1145/2851613.2851956

gramming. They often lead to hard to trace erratic errors and these had become apparent almost since the beginning of modern computing. An early overview of the problem had been given by see *e.g.* Netzer and Miller (1992). Modern hardware deals with races by providing so-called *atomic instructions*. C, as of version C11, abstracts and encapsulates these instructions in atomic types (`_Atomic` keyword) and *operations*.

Dealing with races efficiently also requires support from the execution platform, namely the OS. In a singular toolbox called *Fast User space muTEXes*, `futex` for short, the Linux kernel combines two levels of operations (atomics and a system call) for the implementation of lock primitives, see Hutton et al. (2002); Hart (2009). With these interfaces a simple but inefficient lock structure `smpl` could look as follows:

```
typedef _Atomic(int) smpl;
void smpl_lock(smpl* lck) {
 for (;;) {
  int prev = atomic_exchange(lck, 1);
  if (!prev) break;
  futex_wait(lck, prev);
 }
}
void smpl_unlock(smpl* lck) {
 atomic_store(lck, 0);
 futex_wake(lck, 1);
}
```

Both functions are simplistic and not very efficient. The first, `smpl_lock`, is inefficient because each failed attempt to acquire the lock will result in a call into the OS kernel, even if the lock would be available almost instantly. The second, `smpl_unlock`, tries to wake up another thread without knowing if any thread is waiting.

3. A NEW GENERIC LOCK ALGORITHM USING FUTEX SYSTEM CALLS

For our strategy we use a single `unsigned` value that at the same time holds the lock bit (HO bit) and a 31 bit counter.

```
typedef _Atomic(unsigned) ftx;
#define ftx_set(VAL)  (0x80000000u | (VAL))
#define ftx_lkd(VAL)  (0x80000000u & (VAL))

void ftx_lock(ftx* lck) {
 unsigned cur = 0;
 if (!ftx_cmpxch(lck, &cur, ftx_set(1))) {
  cur = ftx_fetch_add(lck, 1) + 1;
  for (;;) {
   while (!ftx_lkd(cur)) {
    if (ftx_cmpxch(lck, &cur, ftx_set(cur))) return;
    for (unsigned i = 0; i < E && ftx_lkd(cur); i++)
     cur = ftx_load(lck);
   }
   while (ftx_lkd(cur)) {
    futex_wait(lck, cur);
    cur = ftx_load(lck);
   }
  }
 }
}
```

That counter counts the number of threads inside the critical section. An update of the counter part is done once when a thread enters the CS.

1. A thread is on the fast path for the lock when the overall value is `0`. The lock can be acquired with one atomic operation which sets the counter and the lock bit simultaneously.
2. Otherwise, we increment the lock value atomically and enter an acquisition loop.
   (a) First, we spin `E` times to acquire the lock.
   (b) If that fails, we go into a `futex_wait`.

Unlocking is a very simple operation. The locker has contributed `ftx_set(1u)` to the value, and just has to decrement the value atomically by that amount. The return value of the operation reveals if other threads still are in the CS, and a `futex_wake` call can be placed accordingly.

It is relatively easy to see that this new strategy provides a functional lock primitive using just a 32 bit data structure and one atomic operation for fast `ftx_lock` and `ftx_unlock`. It remains to show that it cannot deadlock.

$t_{\mathbf{fail}}$ is the maximum of two system specific times: the time a thread T_1 may either spend in a failed attempt to `futex_wait` or that the system needs to put T_1 to sleep and start another thread T_2.

P is the *number of processor cores*, which is viewed to be equal to the maximum number of threads that are scheduled simultaneously.

$t_{\mathbf{para}}$ is the time that P threads need for a spinning phase that they perform in parallel.

LEMMA 3.1. *Provided that no other threads are unscheduled, after at most* $t_{para} + (P-1) \cdot t_{fail}$ *seconds a first thread successfully calls* `futex_wait`.

PROOF. For the first term, observe that after t_{para} time, at least one thread has finished the spinning phase, and attempts `futex_wait`.

While no thread is unscheduled at most P scheduled threads can enter the CS. There are at most $P-1$ atomic increments that change the futex value. Thus the first thread that enters the CS will need at most t_{para} time for spinning and then `futex_wait` may fail at most $P-1$ times in a row. □

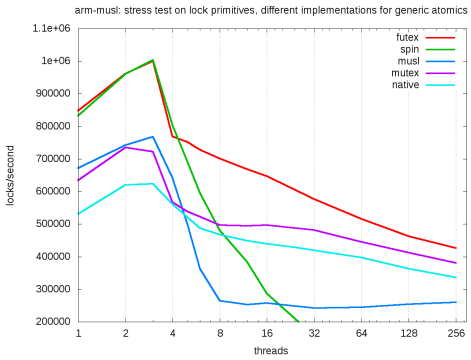
With this lemma, in the full version of this paper, Gustedt (2015), we are able to prove the following theorem:

THEOREM 3.2. *Let be* T_0 *a thread out of* $N \gg P$ *that is unscheduled when holding the lock. Provided that none of the threads is unscheduled by other means and that* $t_{para} \leq t_{fail}$, *after a time of* $N \cdot t_{fail}$ *the application makes progress.*

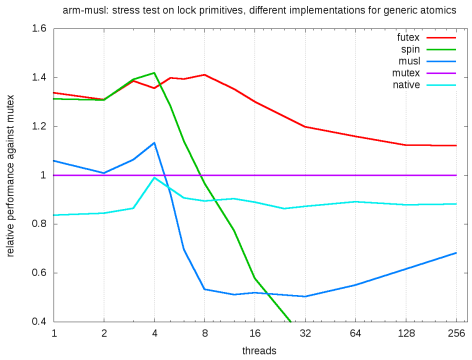
4. BENCHMARKS

Our benchmark comes with the reference implementation of Modular C, see http://cmod.gforge.inria.fr/. It is a list-based stack implementation that uses an atomic pair of two values for the head to avoid the ABA problem, IBM (1983); Michael (2004). The size of the atomic data structure has been chosen to force the use of the lock-based generic atomic functions.

Fig. 1(a) shows the results on a 4 core `arm7` platform. We see that all lock implementations allow for an acceleration of the application when a small number of threads is used. But what is also clear that the "native" lock performs worst for the case that is the most interesting: the range where



(a) lock throughput



(b) relative performance compared to mutex

Figure 1: benchmarks on `arm`

each thread has its own CPU core at its disposal. Even the "mutex" lock performs better.

We also see that musl's internal lock structure shows a drastic performance loss when it comes to congestion. This is due to a switch of the spinning strategy: as soon as congestion is detected, spinning is abandoned and threads directly attempt **futex_wait**. This is meant to ensure fairness of lock acquisition, but as we can see for our use case it has a dramatic impact on the application throughput.

Fig. 1(b) shows the relative performance of the same experiments, where the "mutex" implementation is taken as base for comparison. We see that our new implementation is about 60% better than the "native" version, or 40% than a direct implementation with mutex. It combines the good performance of a spinlock for the less congested range with a good policy for strong congestion.

Other test have been run on a `x86_64` platform with 2x2 hyperthreaded cores. Although it has more compute power than the other, the atomics of the hardware are much less performing. This is due to the fact that here an atomic instruction usually enforces a complete synchronization at a cost of about 50 CPU cycles. Basically, the CPU is blocked for this number of cycles. Compared to that, in a monitor based approach as on the arm architecture part of these cycles can be used for other computations. So on the `x86_64` platform any atomic operation incurs a strong latency penalty. Thereby, our application is not even able to accelerate for 2, 3 or 4 threads as it was the case on arm. In the contrary it even decelerates. Nevertheless the relative performance difference between the different lock implementations look very similar. For figures and more information we refer to the full version of this paper, Gustedt (2015).

5. CONCLUSION

We have presented a new locking algorithm that combines consequent use of C11 atomics with Linux' futex system calls. We have proven that it is deadlock free, and that it shows better performance than other lock implementations.

By pursuing this research we learned to mistrust some of the urban legends that turn around atomics, futexes and lock structures in general. At least when we stick to the basics (**futex_wait** and **futex_wake**) and if we have a decent interface for atomics, programming them is not as difficult as the legends suggest. Also using a system call is not so much worse that spinning around an atomic access. The performance factor between the two is only about 10, and so spinlocks in the order of 10 should be sufficient in many cases.

This support library is now available as open source at http://stdatomic.gforge.inria.fr. We hope to integrate it into the C library that we used for most of our experiments, `musl`.

References

Alpine Linux. http://alpinelinux.org/

Clang. http://clang.llvm.org/

gcc. GNU Compiler Collection. https://gcc.gnu.org/

glibc. GNU C library. https://www.gnu.org/software/libc/

Jens Gustedt. 2015. *Futex based locks for C11's generic atomics*. INRIA RR-8818 https://hal.inria.fr/hal-01236734

Darren Hart. 2009. A futex overview and update. *LWN.net* (2009). https://lwn.net/Articles/360699/

Andrew J. Hutton et al. 2002. Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux. In *Proceedings of the Ottawa Linux Symposium*. 479–495.

IBM 1983. *IBM System/370 Extended Architecture, Principles of Operation*. IBM. SA22-7085.

JTC1/SC22/WG14 (Ed.). 2011. *Programming languages — C* (cor. 1:2012 ed.). Number ISO/IEC 9899. ISO.

Maged M. Michael. 2004. *ABA Prevention Using Single-Word Instructions*. Tech. Rep. RC23089. IBM Research.

MUSL libc. http://musl-libc.org

Robert H. B. Netzer and Barton P. Miller. 1992. What Are Race Conditions? Some Issues and Formalizations. *ACM Lett. Program. Lang. Syst.* 1, 1 (March 1992), 74–88.

POSIX. 2009. *ISO/IEC/IEEE Information technology – Portable Operating Systems Interface (POSIX®) Base Specifications*. Issue 7. ISO, Geneva, Switzerland.