Futex based locks for C11's generic atomics

Jens Gustedt

▶ To cite this version:

Jens Gustedt. Futex based locks for C11's generic atomics. [Research Report] RR-8818, INRIA Nancy. 2015. hal-01236734

HAL Id: hal-01236734
https://inria.hal.science/hal-01236734

Submitted on 2 Dec 2015

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Futex based locks for C11's generic atomics

Jens Gustedt

ISSN 0249-6399   ISRN INRIA/RR--8818--FR+ENG

RESEARCH REPORT
N° 8818
December 2015
Project-Team Camus

Futex based locks for C11's generic atomics*

Jens Gustedt[†‡]

Project-Team Camus

Research Report n° 8818 — December 2015 — 13 pages

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%.

Key-words: lock primitives, atomics, C11, futex, Linux

* A short version of this paper is accepted for publication in SAC'16, see [Gustedt 2016]

† Inria, France

‡ Université de Strasbourg, ICube, France

RESEARCH CENTRE
NANCY – GRAND EST

615 rue du Jardin Botanique
CS20101
54603 Villers-lès-Nancy Cedex

Verrous basés sur futex pour les opérations atomiques génériques de C11

Résumé : Nous présentons un nouveau algorithme pour une primitive de verrouillage et son implantation qui se basent sur l'interface natif de Linux, l'appel système `futex`. Ceci nous permet d'assembler le support à la compilation de structures de données atomiques tant qu'il ne peut pas être réalisé par une instruction matérielle. Une tel outil est nécessaire pour l'interface atomique de C11, car ici la qualification **_Atomic** peut être attribuée à presque tous les types de données. Notre structure de verrouillage vérifie des propriétés spécifiques pour son champ opérationnel et ses besoins de performance. Ainsi nous sommes en mesure surpasser les performances de la bibliothèque `libatomic` de gcc par 60%.

Mots-clés : primitives de verrouillage, opérations atomiques, 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 by the C compiler and the implementation of 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`.

By consensus in the community an API interface has emerged that both, `gcc` and `clang`, implement.

These library stubs can rely on all their knowledge of the architecture specific atomics for the non-contended part of the implementation, finally that is the context in which they were created. But since they are generic, in case of contention these library stubs rely on generic OS interfaces to put threads to sleep. 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 for which there are reliable and efficient implementations.

This situation is less than ideal, a language interface should map to platform specific properties and draw the best performance from any given hardware. 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 benchmarks that are presented in this paper come with the reference implementation of Modular C, see http://cmod.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.

2. TOOLS FOR DATA CONSISTENCY AND RACES

Data races are the most difficult challenge for parallel programming. They often lead to hard to trace erratic errors. The main problems are:

atomicity:. Writes to memory may or may be not split by the hardware into several chunks that are written separately. A reader may occasionally see inconsistent values, for example a high and a low word of a wide integer that originate from different writes.

divisibility:. A read-modify-write operations such as a simple `i++` may be split into several CPU instructions, e.g a read, an increment and a write. If a first write of another threads falls sometime between the read and the write, the new write will erase the value with one that is outdated.

memory ordering:. The results of read and writes to different objects in memory might be perceived by different threads in different order. Reordering of instructions may originate from optimization (compiler) or occur at run time (processor).

A data type that is guaranteed to avoid the first problem is called *atomic*. Before the C11 revision C only had one data type, **sig_atomic_t**, that had this guaranty to be read and written in one chunk. It can be used to communicate state between a user program and a signal handler.

For the second problem, divisibility, C had no standard tool. No other operation than read or write of **sig_atomic_t** was guaranteed to be *indivisible*. The *indivisible* operations that most computing hardware offer could not be accessed through language features. Usually they had to be programmed through extensions such as inline assembler or special compiler builtins.

Before C11, C also had no thread concept, so the memory ordering problem could not even be formalized within the vocabulary of the C standard. Obviously, it also could not provide tools to deal with it.

2.1. C atomics and its library interfaces

With modern multi-processor and multi-core hardware, parallel programming is an imperative for many if not most user applications that are used on a larger scale. Therefore it was crucial for C to provide the concepts and tools that are necessary to deal with it. So C11 introduces a lot of vocabulary and two optional features: threads through the `<threads.h>` interface, and atomics trough the `<stdatomic.h>` interface. Evidently here we are more interested in the latter, but it is important to note that both features need each other to unfold all of their potential.

C11 introduced a new qualifier, **_Atomic**. A such qualified object guarantees that any read or write access to it is *atomic* in the sense we have defined above. This qualification also guarantees that between different threads all standard operations (defined through operators such as `+=` or functional such as **atomic_exchange**) are perceived as *indivisible*. Note well that this guarantee is only given *between threads* and *in perception*: in reality an operation can well be divided into several processor instructions and the perception guarantee does not extend to visibility between the main program and signal handlers. An operation that extends perception of indivisibility to signal handlers is called *lock-free* in the jargon of the C standard. Below we will see where this choice of words originates.

C11 also introduces different concepts of **memory_order** for atomic operations. The whole of that specification is much too complex to unfold, here. In the following we will assume *sequential consistency* (**memory_order_seq_cst**) for all atomic operations. This forces all atomic operations to appear totally ordered between all threads.

We will use the following atomic operations:

atomic_store: store a new value
atomic_exchange: store a new value and return previous
atomic_fetch_add: add value and return previous
atomic_fetch_sub: subtract value and return previous
atomic_compare_exchange_weak: compare to desired, then exchange, may fail

2.2. Atomic instructions on modern hardware

Almost since the beginning of modern computing, parallelism was implemented in hardware and the consistency problems that we introduced above became apparent. An early overview of the problem had been given by [Netzer and Miller 1992]. Modern hardware (which almost always is inherently parallel) deals with this by providing special instructions, usually referred to as *atomic instructions*. It is important to note that these are not the same as the atomic operations on the level of the C language.

word size:. Usually atomic instructions are limited to word sized data types. Available on most platforms are instructions for 8, 16, 32 and 64 bit data types. Some also extend to 128 bit.

primitives:. The instructions that are implemented may or may not directly correspond to atomic operations. E.g some CPU may have a proper instruction for the increment operator `++`, e.g `x86_64`, on others, e.g `arm`, such an operation will have to be composed from primitives.

boundedness:. Atomic instructions may give a guarantee to succeed within a bounded time (usually some memory cycles) or just return success or failure. For the latter, this may result in C level operations that have an *unbounded* response time.

state:. Atomic instructions may operate on an internal state of the platform. E.g `arm` CPU work with a feature called *monitors* that memorize state of previous atomic access to memory.

Due to all these differences, programming with atomic instructions directly on assembly level is a mess, and in general it is very tedious to provide portability between different architectures. Such code has to ensure

— the correct composition of atomic primitives to obtain sensible semantics of the operations,
— the correct alignment of all atomic object such that they do not cross cache line boundaries,
— the correct ordering of instructions, e.g it has to ensure that neighboring store instructions are not reordered by the CPU,
— that the unboundedness of some operation may not result in application deadlocks,
— and that the OS correctly restores processor state when the execution context switches from one thread to another or to a signal handler.

Luckily, C11 now ensures that only C compiler and C library Implementers have to consider all the glorious details of a specific architecture. One problem remains though, and this is what this paper is all about. Because of the limited word size for atomic instructions, the implemented compiler operations cannot just resort to a composition of atomic primitives on the atomic object itself. If an object is large, say 128 bit wide, or has a size that is not a power of 2, they must rely on external or internal *locks* that protect a *critical section*, CS. That is they need an auxiliary object that protects the data object by means of some *lock primitives* and by memorizing a *state* of the application.

Typically such locks can be made invisible between different threads, but remain visible between a thread and its signal handler. So the access to an object that is qualified with **_Atomic** but that needs a lock for operation may be divisible with respect

to a signal handler. This property is what coined C's terminology of *lock-free* that we already mentioned above.

2.3. Fast user space mutexes

In a singular toolbox called *Fast User space muTEXes*, `futex` for short, the Linux kernel combines two levels of operations for the implementation of lock primitives, see [Hutton et al. 2002; Hart 2009]:

(1) User space atomic integers with lock-free operations are used to regulate access to the lock as long as it is not congested.
(2) Wait and wake-up system calls resolve conflicts when the lock is under congestion by multiple threads or processes. They relate to such integers by address (user space or kernel space addresses) and are guaranteed to be perceived as indivisible by the caller.

In the beginning, when `futex` were first introduced they needed non-standard features: assembly extensions for the atomic instructions, and a system call interface into the Linux kernel. Fortunately with the atomics interface of C11 we now have a standardized tool for the first. For the second, in the following we will assume that we have two library calls **futex_wait** and **futex_wake** at our disposal. This is only a very restricted set of the available futex operations that is simple to master and avoids most of the complexity of the interface as it was described by [Drepper 2011]. The only difficulty that remains is that a call to **futex_wait** is only successful if the kernel can guarantee that the value of the atomic object still is as expected. 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);
}
```

Here the second parameter to **futex_wait** guarantees that the thread will only be set to sleep if the value of the atomic object `*lck` still is `prev`. As a consequence the **smpl_lock** function will iterate until the atomic exchange succeeds in modifying the value from a previous value of `0` to the value of `1`.

The second parameter of **futex_wake** corresponds to the maximal number of threads that are to be woken up. So here, the thread that holds the lock restores the object `*lck` to the value `0` and wakes up one possible waiter.

Both functions as described above 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 any knowledge if there even is such a thread that is waiting for it.

To avoid these two shortcomings, system libraries that implement locks (such as e.g `glibc` and `musl`) usually combine two strategies:

— A first spinning phase attempts the atomic operation several times. Thereby an application with a very short CS can mostly avoid sending threads into sleep.
— They use at least two **_Atomic** objects, one for the lock itself and a second one that counts the waiters. By checking if the counter is 0, this allows to avoid useless calls to **futex_wake**.

Even though these additions enlarge the lock data structure and add one atomic operation to the unlock function these strategies have proven to be much more efficient then our simplistic versions, above.

3. A NEW GENERIC LOCK ALGORITHM USING FUTEX SYSTEM CALLS

To construct and adapted lock data structure for our situation, we want to have the following properties:

— The size of the data structure should be minimal: it should use just one 32 bit word as it is needed by the `futex` calls.
— When there is no contention, the number of atomic operations should be minimal. That is one such operation for each, `lock` and `unlock`, should suffice in that case.
— The procedure should be efficient, that is it should not unnecessarily waste resources. In particular, threads that have no chance to acquire the lock should be put into an OS sleep state.
— The procedure should be optimized for the use case, namely a CS that is very short, just the time to a copy operation of a small object.
— If the number of threads is bounded, the procedure should be deadlock free.

Typical OS lock structures also have the problem of *fairness*, that is that they should ensure that no single thread has an waiting time for the lock that is much longer than for others. For a lock that is used for atomics, this is only a secondary concern if we can guarantee liveness. As we will see below in the benchmark section, congestion here only is noticeable for an extremely high load, for which we may not expect any reasonable application responsiveness.

3.1. The algorithm

For our strategy, see Listing 1, we use a single **unsigned** value that at the same time holds the lock bit (HO bit) and a 31 bit counter.[1] That counter is not viewed as a counter of the threads that are in a kernel wait, but counts the number of threads inside the critical section. An update of the counter part is done once when a thread enters the CS. Compared to the number of times the counter is accessed under congestion such events are relatively rare. Thereby we save memory bandwidth for the update, and we also avoid too much interaction between the different threads that compete for the lock.

(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, if successful.
(2) Otherwise, we increment the lock value atomically and enter an acquisition loop.
    (a) First, we spin `E` times (`E` determined below) to set the HO bit as well, and thus acquire the lock.
    (b) If that fails, we suppose that the lock is under congestion and we go into a **futex_wait**.

[1]On Linux, **unsigned** is always 32 bit wide.

Listing 1. A lock algorithm using a single **_Atomic unsigned**.

```
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 (!atomic_compare_exchange_weak(lck, &cur, ftx_set(1))) {
  cur = ftx_fetch_add(lck, 1) + 1;
  for (;;) {
   while (!ftx_lkd(cur)) {
    if (atomic_compare_exchange_weak(lck, &cur, ftx_set(cur)))
       return;
    for (unsigned i = 0; i < E && ftx_lkd(cur); i++)
     cur = atomic_load(lck);
   }
   while (ftx_lkd(cur)) {
    futex_wait(lck, cur);
    cur = atomic_load(lck);
   }
  }
 }
}
```

Going into the **futex_wait** may fail if the value changes, but since additional threads only change the counter when they arrive, this will not happen too often and the thread goes to sleep, eventually. 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.

```
void ftx_unlock(ftx* lck) {
 unsigned prev = atomic_fetch_sub(lck, ftx_set(1u));
 if (prev != ftx_set(1u)) futex_wake(lck, 1);
}
```

3.2. Analysis

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. The worst case scenario for our use of our lock primitive is that the thread that holds the lock, say T_0, is unscheduled while inside the CS. Suppose further that there are N other threads that are ready to be scheduled, and that once they are scheduled they start to compete for the lock.

Different quantities are interesting for an analysis of the runtime behavior of the algorithm. We can control one of them, namely the time t_{mono} that a scheduled thread spends spinning before trying to switch to **futex_wait**. Three others are platform dependent:

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

A value t_{para} close to t_{mono} indicates a perfect parallelism, a value of $P \cdot t_{\text{mono}}$ means that there is none at all. Usually it will be greater than t_{mono}, e.g because of memory contention or contention on other shared resources (execution pipelines, caches). We derive some other quantities from the above:

$\widehat{P}$. given as $\frac{P \cdot t_{\text{mono}}}{t_{\text{para}}}$ is the *parallelism* of the platform.

E. given as $\frac{t_{\text{mono}}}{t_{\text{para}}} = \frac{\widehat{P}}{P}$ is the *efficiency* of the platform.

For example, on a modern hyperthreaded machine with 4 cores in total, $\widehat{P}$ is typically between 2.5 and 3, E is between 0.625 and 0.75.

Remark 3.1. On a platform where $\widehat{P}$ is close to one, the spinning phase of the algorithm should entirely be skipped.

This is simply because there no other thread can make progress while a thread is spinning. Thus spinning would just waste resources and the state of the application would not progress. So from now on we may assume that $\widehat{P} \geq 1 + \epsilon$ for some reasonable value of $\epsilon > 0$.

LEMMA 3.2. *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. □

This already shows that, provided no other descheduling takes place, our algorithm is deadlock-free.

Now, once a thread successfully goes into **futex_wait** a new thread T_P can be scheduled, compete for the lock and change the `futex` value. It may disturb all other threads that are trying to go into **futex_wait**, forcing them to restart their attempt.

Remark 3.3. Provided that no threads are unscheduled otherwise, that there are always P threads inside the CS and that at least one of them has finished spinning, after a time of t_{fail} another threads succeeds his call to **futex_wait**.

That is, under these circumstances we have a stable regime where each t_{fail} seconds a thread enters **futex_wait**.

To be able to ensure that there is always at least one thread that has finished spinning, we observe that if $t_{\text{para}} \leq t_{\text{fail}}$ (or equivalently $t_{\text{mono}} \leq E \cdot t_{\text{fail}}$) a newly scheduled thread has finished spinning when the next thread successfully goes into **futex_wait**.

LEMMA 3.4. *Provided that no threads are unscheduled otherwise, that there are always P threads inside the CS and that $t_{para} \leq t_{fail}$, threads succeed calls to* **futex_wait** *at a rate of $1/t_{fail}$ per second.*

Or, roughly if $P \ll N$ the time for all threads to calm down and successfully call **futex_wait** is $N \cdot t_{\text{fail}}$.

THEOREM 3.5. *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.*

PROOF. This progress can be of two forms. Either there is another thread than T_0 that does not enter the the CS and thus progresses the application, or T_0 will be rescheduled and finishes its CS. □

The time t_{mono} has not only an influence for this worst case, but is also responsible for the response time in the non-congested situation. The longer we spin, the higher the probability to acquire the lock without going into **futex_wait**. So the best compromise would be to choose $t_{\text{mono}} = E \cdot t_{\text{fail}}$. Practically a factor of 0.9 always guarantees liveness of the application and shows good performance on average.

4. BENCHMARKS

4.1. The framework

We have run a long series of benchmarks to validate the approach. The code for the benchmark is integrated in *p11* with comes with *Modular C*, see Cmod. For compilation of that benchmark we also need a C11 compliant library, that has C11 threads, and a C11 compiler that also has gcc extension. We used `musl` for the first and gcc and clang for the latter.

The implementation of our algorithm is a bit more sophisticated than what may appear above. In particular it takes care of reducing the number of atomic operations to a minimum and to use memory ordering for the locks that is adapted to the case.

4.2. The test program

The test in p11 is called p11#test#lifo. It is based on a stack implementation (Last In First Out) that uses an atomic pair of pointers for the head to avoid the ABA problem, [IBM 1983; Michael 2004]. For the benchmarks, the size of the atomic data structure has been chosen in such a way that the generic atomic functions based on locks are chosen.

The idea of this benchmark is to have a application that runs under full load, stresses the platform with a lot of allocations and deallocations and in the middle of that does a lot of locking and unlocking. It works as follows:

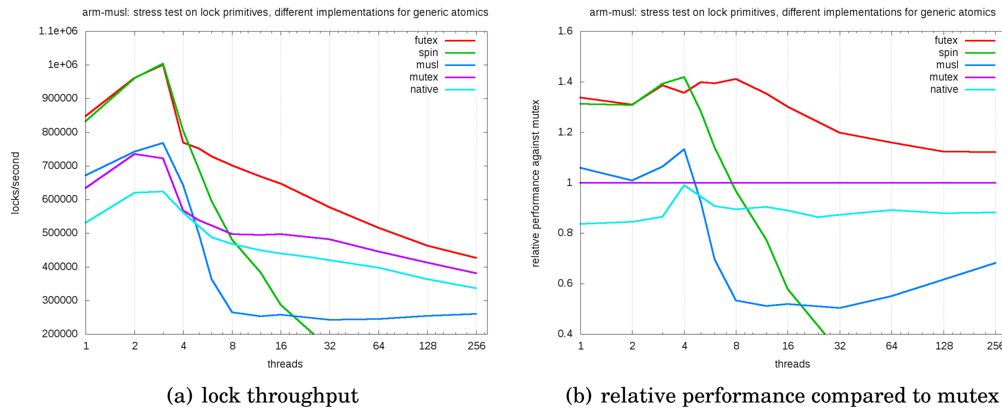
It creates or deletes a random number of list elements for the lifo inside a loop. All test runs last `10s` and were repeated at least 10 times. The measure that is reported is the number of list elements that have been handled per second on average.

The parameters of the runs are the number of threads that run in parallel, values ranging from `1` up to `256`. Different lock primitives can be chosen at compile time to protect the head of the LIFO:

futex. the futex based algorithm described here
mutex. based on a standard mutex
musl. musl's low-level `lock/unlock` functions
spin. a spin lock using atomic exchange
native. the compilers "native" generic lock, also a mutex

4.3. The test platforms

4.3.1. An `arm7` machine with 4 cores. This machine has 4 symmetric `arm7` cores at a `1.3 GHz` with `2 GiB` of RAM. This system is equipped with Alpine Linux, so it has `musl` as



(a) lock throughput (b) relative performance compared to mutex

Fig. 1. benchmarks on `arm`

a native C library. The processor has atomic instructions for word sizes up to 64 bit. The compiler is `gcc` version `5.2`. Here, the native C library is musl and so the atomic library of gcc are compatible. Therefore the benchmarks include "native".

4.3.2. A x86_64 machine with 2x2 hyperthreaded cores. This is a i7-4600U CPU at `2.10GHz` and with `8 GiB` of RAM. The OS is Debian Linux, with `glibc` as native library. The processor has atomic instructions for word sizes up to 128 bit. The compiler is `gcc` version `4.9`. Because musl is not the native C library, the atomic library of gcc is not compatible. Therefore the benchmark "native" is missing.

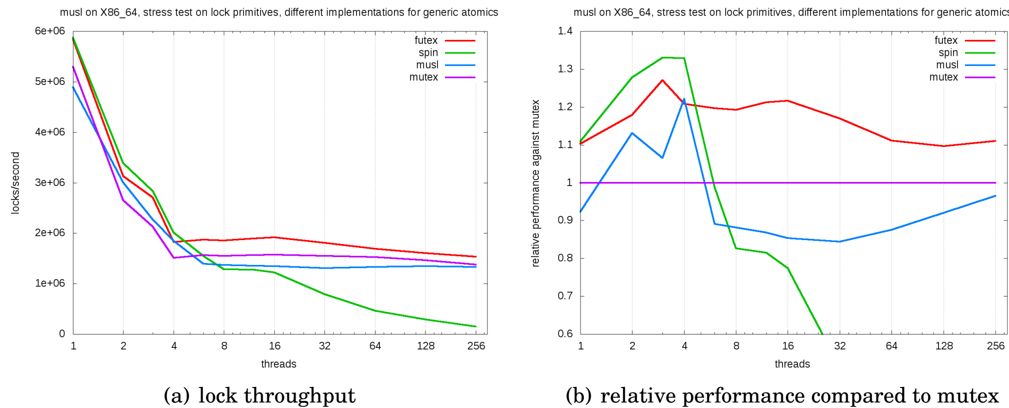
4.4. Performance comparison

Fig. 1(a) shows the results on the `arm` 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 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 a base. 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.

To finish let us consider the `x86_64` platform, Fig. 2. 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



(a) lock throughput

(b) relative performance compared to mutex

Fig. 2. benchmarks on x86_64

even decelerates. Nevertheless the relative performance difference between the different lock implementations look very similar.

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. is deadlock free, and that it shows better performance than other lock implementations.

This is not surprising, an implementation that is tuned for the purpose (very short CS) and that may avoid stacked calls into the C library should always perform better than a generic one. Surprising to us was the wide performance gap between the 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. http://alpinelinux.org/
Clang. http://clang.llvm.org/
Ulrich Drepper. 2011. Futexes are tricky. Red Hat Inc., rev. 1.6. (2011). http://www.akkadia.org/drepper/futex.pdf
gcc. GNU Compiler Collection. https://gcc.gnu.org/
glibc. GNU C library. https://www.gnu.org/software/libc/
Jens Gustedt. 2016. Futex based locks for C11's generic atomics, extended abstract. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. ACM. accepted for publication.
Darren Hart. 2009. A futex overview and update. *LWN.net* (2009). https://lwn.net/Articles/360699/

Andrew J. Hutton, Stephanie Donovan, C. Craig Ross, Hubertus Franke, Rusty Russell, and Matthew Kirkwood. 2002. Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux. In *Proceedings of the Ottawa Linux Symposium*. 479–495. https://www.kernel.org/doc/ols/2002/ols2002-pages-479-495.pdf

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. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf

Maged M. Michael. 2004. *ABA Prevention Using Single-Word Instructions*. Technical Report 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. DOI:http://dx.doi.org/10.1145/130616.130623

POSIX. 2009. *ISO/IEC/IEEE Information technology – Portable Operating Systems Interface (POSIX®) Base Specifications*. Vol. 9945:2009. ISO, Geneva, Switzerland. Issue 7.

RESEARCH CENTRE
NANCY – GRAND EST

615 rue du Jardin Botanique
CS20101
54603 Villers-lès-Nancy Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399