A lock-less shared memory barrier without atomic operations

Ronny Brendel
Tutors: Sascha Klüppelholz, Marcus Völp

March 21, 2013

Abstract

Today's concurrent algorithms waste time, because they often work in a needlessly deterministic fashion. One of these is the barrier. In this report we present an extension of Mc Guire's lock-less, chaotic barrier protocol. We prove its correctness by means of checking appropriate properties on a markov decision process model. Furthermore we analyse quantitative properties on a continuous-time markov chain model of the protocol. Last we compare the performance of a straight-forward implementation to two common barrier implementations. The new protocol improves barrier performance by using the CPU's inherent probabilistic behaviour rather than enforcing a deterministic protocol to follow. The barrier example shows that implementing this idea in concurrent algorithms has the potential to greatly benefit performance and should definitely be pursued.

1 Introduction

Inspired by the idea of pW/CS [7] we intend to evaluate which other parallel synchronisation operations can be improved through stochastic algorithms. One of these operations is the barrier.

A barrier is a synchronisation primitive where each thread in a team waits until all of them reach this point in the execution before continuing. Torsten Höfler's diploma thesis [6] gives a good overview and analysis of current barrier protocols.

The barrier is a common synchronisation operation. In OpenMP [9], by default, many operations such as for-loops, sections, and single directives invoke an implicit barrier at the end of the operation. Many distributed computing algorithms use a *lock-step* approach to interleave computation with communication phases. For example a program simulating weather divides the region in question into a grid. In each of the squares on this grid the weather progression of the next 20 seconds is simulated concurrently. As a next step the squares need to exchange information such as the temperature and humidity at the borders to the neighbouring squares in order to be able to simulate the next 20 seconds. To facilitate this behaviour, barriers are used to make sure all squares finished simulating and are ready to exchange data, before repeating this process.

According to a survey conducted at the HLRS [11] the barrier is the 5th most time-consuming Message Passing Interface (MPI) [8] operation. On average 6%

of the whole CPU time spent in MPI calls and 0.81% of the whole program time is spent in barriers. A very small fraction of this is the actual time used synchronizing. The majority is due to waiting for other threads to arrive, because the computation between threads is unbalanced (some take longer, some are quicker). It is still useful to pursue improvements in the barrier protocol for benchmarks and programs that require lots of synchronisation.

The report is structured as follows: First, we present the new protocol. In Section 4 we prove its correctness. The next step is to analyse quantitative properties of the protocol. Furthermore an implementation will be evaluated and compared to the barrier implementations of GNU OpenMP [5] and PThreads [4]. Concluding thoughts, and ideas for future work wrap up the report.

2 Analysis of the proposed barrier protocol

The protocol depicted in Figure 1 is part of a personal communication with Nicholas Mc Guire. Therefore no reference can be cited.

```
shared: barrier:=1
local: me:=primes[index], other:=primes[len-1-index]

do {
    if barrier≢0 mod me {
        barrier := barrier * me
    }
} while barrier≢0 mod productOfAllPrimes

do {
    if barrier≡0 mod other {
        barrier := barrier / other
    }
} while barrier≠1
```

Figure 1: Pseudo code of the proposed barrier protocol

The idea is that each thread in a team is identified by two different primes. The primes are chosen from a given set of primes which has as many elements as there are threads. One run of the barrier is a run through both loops. In the first loop a thread multiplies itself to the barrier variable. All threads arrived when barrier equals the product of all primes in the set. Analogously, in the second loop, each thread divides itself from the barrier variable until it equals one. After this, the barrier operation is completed.

Intuitively each thread runs busily in its loop trying to commit itself to the barrier. Since modification of the barrier variable (read until write) is not atomic, the threads rapidly overwrite each other's commitment attempts. But eventually all threads will be multiplied to the barrier. The idea is that, even though the threads read and write wildly, this approach is potentially faster than currently used protocols, which are lock-based or lock-less using atomic operations.

Having two different primes for one thread (Why not just one?) is an attempt to avoid a possible deadlock: Assume `me=other`. A thread finishes the first loop and enters the second, and then divides itself from the barrier again. Since the thread responsible for multiplying itself to the barrier is already in the second loop, the barrier cannot equal the product of all primes anymore. Thus some threads are stuck in the upper loop.

A similar scenario can still be created in the proposed protocol: Suppose two threads sharing one prime (as `me` and `other`) exit together. They both divide their primes from the barrier in the second loop. Since the thread having the shared prime as `me` cannot multiply itself to the barrier again, the other threads are, again, stuck in the first loop. Because the memory used by the barrier variable is little, we can introduce a second one to solve this issue. The second and first loop would then both multiply until they reach the product of all primes, but on two distinct variables, one for each loop.

Another problem is that ”`if barrier % me > 0` { `barrier *= me` }” is not atomic – We read barrier twice. A thread can multiply itself multiple times to the barrier and eventually overflow the bound of it. Figure 2 depicts an example where a thread multiplies its prime to the barrier variable twice. This problem can be solved by introducing a thread-local copy of barrier. On this copy a thread then executes the operations above and writes the copy back.

time	prime			comment
	2	3	5	
1	read 1	read 1		
2	write 2			
3			read 2	
4		write 3		
5	read 3			barrier $\not\equiv 0 \mod 2$
6			write 10	
7	read 10			
8	write 20			$20 = 2 * 2 * 5$

Figure 2: Example execution of multiplying a thread twice to the barrier

Having solved these two issues, there is still another way to create a deadlock. Suppose one thread completes the first loop and leaves. Then another thread rewrites barrier to a value where the thread which has already left is not committed. This thread now has no means of multiplying itself to the barrier again. Thus all threads except one are locked inside the first loop. A solution to this problem is to introduce a new shared variable that signals all threads whether one thread has completed the barrier. A thread is then allowed to leave the loop, if either barrier equals the product of all primes or another thread has already left.

Having all of the above problems and solutions in mind, we now present an extended version of the protocol.

3 The extended protocol

In addition to solving the problems mentioned before, we switch from multiplying primes to the barrier variable to a bitset representation (one thread per bit). This way we can have up to 64 threads in the barrier whereas the product of the first 64 primes would exceed the bounds of a 64-bit integer variable.

The resulting protocol is depicted in Figure 3.

```
shared: entry:=0, exit:=0, left:=false
local: copy, me:=$2^{threadIndex}$, full:=$\sum_{i=0}^{numThreads-1} 2^i$

if left = false {

    do {
        copy := entry
        if copy & me = 0 {
            copy  := copy | me
            entry := copy
        }
    } while copy $\neq$ full and left = false

    left := true
    exit := 0

} else if left = true {

    do {
        copy := exit
        if copy & me = 0 {
            copy := copy | me
            exit := copy
        }
    } while copy $\neq$ full and left = true

    left  := false
    entry := 0

}
```

Figure 3: Pseudo code of the extended barrier protocol

Allow us to explain. & is the bitwise and operator. | is the bitwise or operator. They are used to check for bits and add bits to a bitset. *entry* is a bitset where each bit stands for one thread having arrived in the first loop. *exit* has the same role for the second loop. *copy* is a thread-local copy of entry or exit depending on the loop the thread is currently in. *left* is true if one thread has left the first loop, or in other words the first barrier is completed and everyone can now leave it. Conversely if left is false, the second barrier has been completed. *me* is the bit of the current thread; it is always a power of two. *full* is a constant meaning all threads have successfully committed themselves to entry/exit.

Initially entry and exit are zero and left is false. A thread arriving in the first loop takes a copy of entry and checks whether it is in it (`copy&me = 0`). If not, it adds itself and writes the result back to entry. Next it checks whether everyone has committed its bit to the entry variable (`entry = full`) or at least one thread has left the loop (`left = true`). If so, it can leave and if not, it has to repeat from the point where it takes a copy of entry.

Suppose one thread has passed the first loop. It then sets left to true signalling the other threads that the barrier has been completed. All other threads will exit the loop soon, too.

Suppose one thread encounters the barrier again. Because left has been set to true after having left the previous barrier, the outermost `if` now directs the thread to the second loop and repeats the procedure described above. Instead of entry it will now use the variable exit. When this barrier is completed, left will be set to false, thus the third encounter will use the first loop again. Repeat. The first barrier is used on every even number of visits, the second on every odd number of visits.

This is another deviation from the original protocol, where it was necessary to finish both loops to complete the barrier operation. During the development and analysis of the new protocol it became apparent that switching between loops, using the variable left, still makes for a correct protocol, which we will prove in the next section. With this optimization, we speed up the barrier by a factor of two.

If we omit the distinction between the entry and exit loop, a thread would not be able to safely reenter the barrier while other threads are still in it. It could immediately exit it again, which is not a desired behaviour. A means of stopping this thread, until all other threads have left, would be necessary. Since 1 bit per thread is not much memory and we can nicely distinguish between the first and second loop, using the variable left, this approach has been chosen.

An important assumption is that the read/write of entry and exit happen in one step. If writing to and reading from entry would take multiple steps, a deadlock scenario could be created. Thus on current computer systems this approach is limited to 32 threads. Interestingly, even on pure 64-bit systems the limit is 32. Circumventing this restriction is part of future work.

4 Correctness of the protocol

In order to be correct this protocol must always terminate and a thread can only leave the barrier if all other threads are present. To verify these two properties we use a markov decision process model of the protocol (Figure 4). To prove termination we check whether location 17 can be reached infinitely often. This implies that the protocol is deadlock-free. Two models, one in Spin [12], one PRISM [10], have been implemented. Both show that the protocol terminates under the mild assumption of weak fairness for scheduling threads. To prove the second part we query the model checkers whether it is possible that two threads are in different loops at the same time. Both, PRISM and Spin, show that this is not the case.

Note that in a real implementation, for example in the C programming language, it would be up to the compiler to decide in which order to evaluate the two subexpressions in the while condition. This freedom of choice is reflected in the model using non-determinism.

Intuitively the protocol terminates, because every time all threads have looped at least once, a minimum of one thread has been successfully committed to the entry variable. Thus, after n repetitions, where n is the number of threads, all the threads have been registered and the barrier is completed.

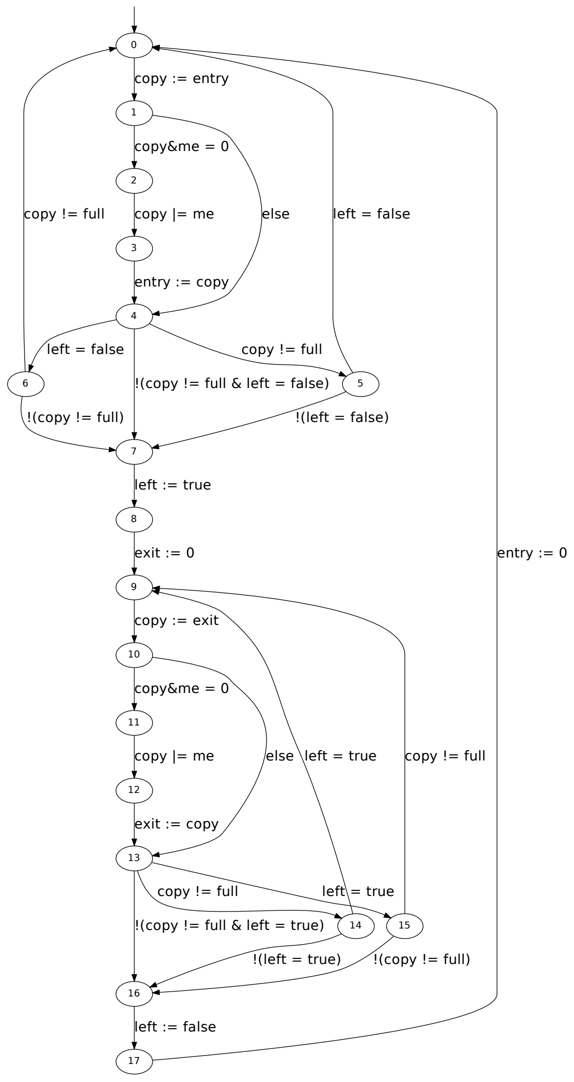

Figure 4: Markov decision process model of the protocol

5 Quantitative analysis of the protocol

In order to analyse quantitative properties of the protocol, we need to determine the expensive operations. Since the protocol uses shared variables to communicate, reads and writes from and to those variables are potentially costly. Each time a shared variable is modified or read, depending on the state of the cache line it is on, the operation takes a different amount of CPU time. Therefore we consider it worthwhile to model the cache behaviour aside from the protocol itself. The cache model will be discussed in the next subsection. Details that are not important for the timing of the protocol, like the non-determinism in the while condition, have been omitted. The resulting Continuous-time markov chain (CTMC) is shown in Figure 5. The rate of a transition is given in parenthesis. There are three different rates: shared read, shared write and work. A *tick* represents an instantaneous operation and therefore receives a transition rate of 1.0. *work* is used to account for computation periods between barrier visits. We explain the rates for shared reads and writes in the following subsection.

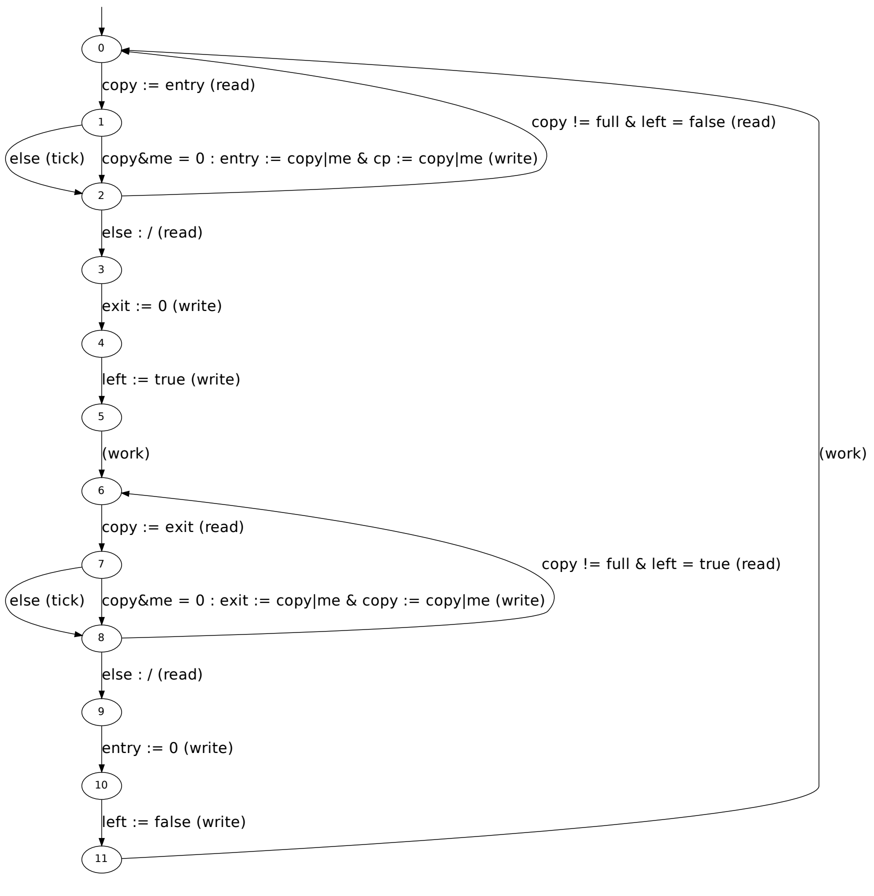

Figure 5: Continuous-time Markov chain model of the protocol

5.1 Modeling cache behaviour

The following description shows a very simple view on the CPU cache and omits many details of current cache protocols. This is necessary to keep the model at a manageable size.

Each CPU core has its own copy of a cache line. A cache line copy has three distinct states. It can be *modified*, which means the core has the only up-to-date copy of the cache line and all other copies are marked *invalid*. It might be *shared*, which means the core has an up-to-date copy, but one or more other cores have a correct copy, too. Or the cache line copy is *invalid*, meaning the local copy is not up-to-date. Figure 6 shows how a cache line copy changes its state depending on events occurring. The dotted transitions are triggered by events (read/write) occurring on other cores, whereas the solid transitions are due to events issued by the own core.

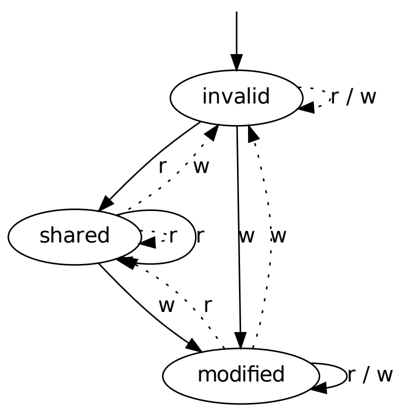

Figure 6: Model of the cache behaviour

For example if a core reads a variable, it first needs to make sure that all other cores take notice and change its cache line copy state to shared in case it was modified before. After this the core fetches the cache line, marks it shared and continues reading the variable.

The timings of shared read and shared write are as follows. A read on a modified or shared variable does not imply any extra work. It can immediately use the cached data. Therefore we consider this operation instantaneous. If a read is done on an invalid copy, it has to first fetch an up-to-date copy of the cache line and make sure all other cores are notified before it can proceed. This usually takes around 50 CPU cycles. A write on a modified variable is instantaneous, since all other cores do not have an up-to-date copy and therefore do not have to be notified. If the cache line in question is in a shared or invalid state, we first have to wait until all other cores follow the request to invalidate their copies of the cache line. After this we can safely write to our local copy and mark it modified. This operation usually takes around 100 cycles.

The number of cycles those operations take is strongly dependent on the CPU itself. For the CTMC model we assume a cache read to use 50 cycles and a cache write to use 100 cycles. The rates of the transitions corresponding to the described events is then the reciprocal of the cycle time: 1, $\frac{1}{50}$ or $\frac{1}{100}$.

5.2 Properties

We want to predict how long the barrier operation takes to complete. Therefore the following three property have been chosen.

1. How long does it take for one thread to pass the barrier?
2. How long does it take for all threads to pass the barrier?
3. If one thread has left the barrier, how long does it take until all of them complete the barrier?

To formalise the above properties we use a continuous stochastic logic [1] [3] representation. For the third query we use the *conditional long-run probability*, which is explained in detail in "Waiting for Locks: How Long Does It Usually Take?" [2]. n represents the number of threads and t is a constant for the time waited in the bounded eventually ($\Diamond_{\leq t}\phi$)

1. $P_{=?}\big[\Diamond_{\leq t} \bigvee_{1\leq i\leq n} loc_i \geq 5\big]$
2. $P_{=?}\big[\Diamond_{\leq t} \bigwedge_{1\leq i\leq n} loc_i \geq 5\big]$
3. $ClrP_{=?}\big[\Diamond_{\leq t} \bigwedge_{1\leq i\leq n} loc_i \geq 5, \bigvee_{1\leq i\leq n}(loc_i \geq 5 \wedge \bigwedge_{1\leq j\leq n, i\neq j} loc_j < 5)\big]$

5.3 Evaluation

The results of the queries for a barrier operation between three threads are shown in Figure 7 and 8. We can see that 90% of the time the barrier is completed within 1000 CPU cycles. If one thread is past the barrier, all other threads will get past it in less than 380 cycles with a probability of 90%.

Note that for these queries the duration of the work period has little influence and is set to 1 cycle.

We were unable to scale this model beyond three threads due to the huge size of the resulting state space. Future work includes tackling this limitation.

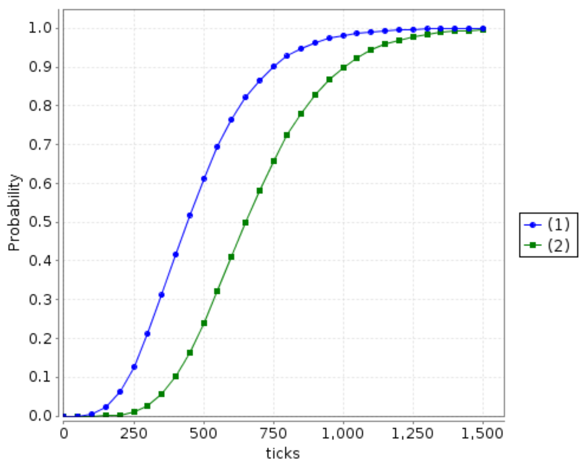

Figure 7: Model checking results of query 1 and 2

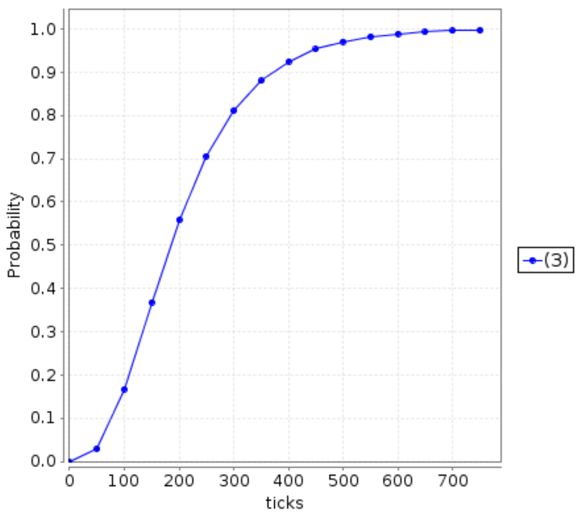

Figure 8: Results of query 3

6 Implementation and measurement

In this section we compare a straightforward implementation of the protocol to two common barrier implementations.

6.1 Considerations

In order to evaluate the performance of the barrier, we spawn a certain amount of threads, repeatedly visit the barrier with a fixed number of repetitions, and afterwards end all threads. We measure the whole running time of the program. To get more reliable results we make sure the operating system's scheduler cannot reposition threads between cores, i.e., we create a one-to-one mapping between threads and cores and pin this configuration for the duration of the measurement. The implementation of the algorithm is an exact translation of the pseudocode in Figure 3 into the C programming language.

The measurements were conducted on a 64 thread, 32 core and 4 socket AMD Opteron$^{\mathrm{TM}}$6274 system.

6.2 Evaluation

The results are shown in Figure 9 and 10. Due to time constraints, the measurements reliable. But they should give an intuition in which ballpark the performance is.

thread count	OpenMP	PThreads	new
4	3.3s	3.5s	0.2s
8	3.3s	3.5s	0.3s
16	3.2s	3.7s	0.6s
32	5.0s	5.5s	1.4s

Figure 9: Measurement results for consecutive barrier visits without delay

When not having any delay, but rather repeating the barrier as quickly as possible, the new protocol outperforms GNU OpenMP's and libc's PThreads implementation by far. This is likely due to the fact that both of them are lock-based and spend considerable time waiting for callbacks.

thread count	OpenMP	PThreads	new
4	5.1s	5.1s	4.6s
8	5.5s	5.5s	4.8s
16	5.9s	5.9s	5.1s
32	7.0s	7.0s	6.1s

Figure 10: Measurement results for barrier visits with 1ms delay

When adding 0.1 millisecond (about 200,000 cycles) of waiting time, the runtime of the new barrier still finishes roughly 10% earlier.

A minor drawback of the new barrier protocol is that the intended false-sharing of the cache line, with the variables on it, adds strain to the cache bandwidth. Measurements show that about 1% of the total cache bandwidth is lost, regardless of the thread count. The percentage is this low, because the speed of false-sharing is limited by the waiting time for transferring a cache line between two caches and the time it takes to invalidate the line on all other caches.

Another drawback is that, due to the busy waiting, the total number CPU cycles used by the protocol is larger than for callback-based approaches. In a setting were performance is important this would not be a problem, because the CPU does not spend the cycles saved on any real work. But it would indeed save energy to remove strain from the CPU.

There exist approaches to reduce the negative effects of both these problems. This will be part of future work.

7 Conclusion

In this report we presented a new barrier protocol whose main advantage over existing protocols is that it neither relies on locking mechanisms nor atomic operations. We proved the correctness of the protocol by checking an MDP model in two different model checkers for potential deadlocks and inconsistencies. Furthermore we presented a brief analysis of a CTMC model showing some basic properties of the protocol. Moreover we saw that a straightforward implementation of the new barrier is significantly faster than two currently used barrier implementations.

8 Future Work

Due to time-constraints, much of what needs to be done is enumerated in this section.

The CTMC model of the protocol is already too large to be analysed when being used with four threads. One should explore ways to reduce its size. Common approaches include symmetry reduction, partial order reduction and reducing the model manually depending on the properties to be checked. Once this has been achieved, further quantitative analysis on the model should be done.

There are various possibilities to make an implementation faster and/or more resource-friendly than the straight forward one shown in this report. For example one could experiment with using the *mwait* instruction in order to wait for changes in a shared variable, thus not having to actively poll it. Another way to speed up the protocol is that a thread keeps track of which other threads it has already seen inside the barrier and does not only commit itself to the it but the other threads it has seen, too.

One should also explore ways to circumvent the current 32 thread limit. Using an array of entry/exit variables and only operating on the items we need should resolve the problem. Since we only want to add one bit at a time, writes would then be atomic again.

Another interesting pursuit is to evaluate ways to modify the protocol in order to make it work in a distributed setting, thus making it an alternative to current distributed memory barrier protocols.

References

[1] A. Aziz, K. Sanwal, V. Singhal, and R. Brayton. Verifying continuous time markov chains. In *Computer Aided Verification*, pages 269–276. Springer, 1996.

[2] C. Baier, M. Daum, B. Engel, H. Härtig, J. Klein, S. Klüppelholz, S. Märcker, H. Tews, and M. Völp. Waiting for locks: How long does it usually take? In *FMICS*, pages 47–62, 2012.

[3] C. Baier, J.-P. Katoen, and H. Hermanns. Approximative symbolic model checking of continuous-time markov chains. *CONCUR99 Concurrency Theory*, pages 781–781, 1999.

[4] GNU C Library. `http://www.gnu.org/software/libc`.

[5] GOMP. `http://gcc.gnu.org/projects/gomp`.

[6] T. Höfler. Evaluation of publicly available barrier-algorithms and improvement of the barrier-operation for large-scale cluster-systems with special attention on infinibandtm networks. *`http://archiv.tuchemnitz.de/pub/2005/0073/data/diploma.pdf`*, 4, 2005.

[7] N. Mc Guire. Probabilistic write copy select. In *13th Real-Time Linux Workshop*, pages 195–206, Oct. 2011.

[8] Message Passing Interface. `http://mpi-forum.org`.

[9] OpenMP. `http://openmp.org`.

[10] PRISM. `http://www.prismmodelchecker.org`.

[11] R. Rabenseifner. Automatic mpi counter profiling. In *42nd CUG Conference*, 2000.

[12] Spin. `http://spinroot.com`.