pW/CS - Probabilistic Write / Copy-Select (Locks)

Nicholas Mc Guire
Disttributed & Embedded Systems Lab, Lanzhou University
Tianshui South Road 222,Lanzhou,P.R.China
mcguire@lzu.edu.cn,safety@osadl.org

Abstract

The initial problem of protecting data for concurrent access was relatively simple, the goal was exclusive access to a shared resource. Elaborate semantical variations of the atomicity theme have been developed over the past decades, followed by a ever increasing focus on scalability. At the same time a continuously increasing complexity of operating systems and applications has resulted in a steady growth of the complexity of the locking subsystem - one could speculate that the locking complexity has been growing faster than the overall complexity of operating systems, but it would be hard to put numeric evidence on this claim - suffice it to state that the development of Linux in the transition from 2.2 to 2.4 and to the now current 3.X series of kernels has been very much dominated by locking issues related to scalability [8], [7].

At the same time we have seen that locking semantics has become more complex, priority inheritance/priority ceiling, fine grain locking, lock types dependent on global state [5] and large lock dependencies (or lock chains [13]) becoming common. This growth in complexity has dramatically impacted the development of real-time OS like the Preempt-RT real-time extension to the Linux kernel - not too surprising considerable efforts related to real-time are lock related [4],[6]. The paradigm has roughly remained the same - explicit mutual exclusion to critical regions and atomicity of access in a functionally deterministic manner along with hardware support for more elaborate atomic instructions (i.e. cmove,cmpx16).

This approach has a serious draw back:

- it is hard to make locking scalable
- detecting and fixing locking problems is becoming more difficult
- the performance impact of locking - notably on real-time - is problematic
- The worst case behavior is only a miniscule sub-state-space hard to actually reach during testing
- Timing wise the worst case is always the loaded system, thus reliable prediction of load impact is limited.

The question is - is there an alternative ? Notably one that scales with growing complexity ? Its not yet time to give a simple yes or no answer, but we believe that we can state that for some locking problems there are solution that can actually inherently scale with growing complexity. The problem simply has to be approached from a different perspective.

Operating systems have been traditionally modeled as deterministic constructs - code is deterministic - but in system scope this simply does not hold. Non-determinism at the temporal level paired with preemptible operating systems inherently leads to the inability to predict the global state of an operating system even in the near future (lets say a few seconds into the future). Thus modeling a task as running on a "random" global state - the operating system - allows a new perspective for access to shared resources. Taking one step back, locking was not introduce to provide exclusive access, locking was introduced to ensure consistency of access to a shared resources - locking being one way this can be done in a straightforward manner. At times where memory was a scarce resource this approach made a lot of sense - with RAM readily available, though with significant access performance differences depending on physical location, alternative solutions for consistent access to shared resources may make more sense - one of these methods is probabilistic locking.

In this paper we present the motivation for a simply probabilistic lock - arguably the term lock is inappropriate - but we retain it as it serves the same purpose as the traditional locks - guarantee

consistency of shared data. This lock is not a one-fits-all solution to the problem of shared data in concurrent systems - but rather it should be seen as an attempt to change the perspective and view contemporary systems as what they are - inherently random systems - and capitalize on this notion to resolve the scalability problem.

1 Introduction

Computer science has been much focused on deterministic methods - notably when it comes to synchronization methods we rely on correctness proofs to assure that the methods are sound. While this does guarantee that these methods will not fail as long as we actually are able to model them (that is we know all involved locks) allowing us to exclude inconsistency of data - they are causing serious problems in the transition to multi-core systems - they don't scale well. A further issue with the deterministic approach is that the worst case is expected under heavy load and thus testing only has a limited significance in certifying correctness as the state space that would need to be covered by testing is simply too large.

The question we ask is simple if the methods in use are actually solving problems that exist or if they are not heavily involved in solving non-existing problems with considerable overhead to do so. Even worse - on very large systems deterministic approaches might well be the problem. Is it reasonable to assume arbitrary defined task sequences or arbitrary preemption sequences at the temporal level? For any real life system this makes little sense - for full fledged general purpose OS it makes absolutely no sense. Even more, the inherent randomness of modern CPUs [9] makes it close to impossible to actually achieve synchronous sequences of concurrent access to unprotected global objects, even if one were to maliciously attempt to do so. Methods like WCET estimation are becoming (actually are at this point) prohibitively pessimistic and thus practically not usable for multicore systems.

The following algorithm contains a race - and it can be quite trivially shown under what conditions the race exists (i.e. a SPIN model would reveal this). We will introduce a lock-less/0-wait algorithm to read a register (of in principle arbitrary size) while concurrently writing it. We will show that this algorithm is reliable with the reader and the writer being non-atomic and then argue that while theoretically unsafe with a non-atomic reader and non-atomic writer it is practically - that is statistically - safe with an arbitrary reliability target. Thus the trade-off is spatial replication vs interprocess synchronization time.

The main contribution of this article is to demonstrate that the growing complexity of modern systems (complex hardware and software) needs answers to core questions, that rather than fighting complexity, capitalize on it and result in robust systems under real-world conditions.

1.1 Race Condition

A race is a access pattern on a shared object that can't be judged from the context of the involved tasks only. It is important to note that unprotected shared access in it self does not suffice to create inconsistency - essentially occurrence of inconsistency is bound to access patterns. thus there are two options:

- unify the context - i.e. add a shared lock to join the context
- de-couple context - i.e. randomize access to minimize joined context

The first is the "traditional" deterministic locking scheme, the second is not actually that new, but maybe just not yet presented in the context of locking. The goal is to design synchronization that scales with complexity rather than trying to enforce simplicity at the local level by increasing the global complexity.

The properties of masking locks build on the notion of inherent non-determinism of concurrency in modern CPUs. Basically this non-determinism at task level is precisely the cause for race conditions in the first place, if modern systems were strictly synchronous at the global level then we could predetermine any access patterns and consequently protect. Sources of non-determinism a plentiful in modern systems, not only asynchronous interrupts, but also non-deterministic cache replacement strategies, ECC RAM and flash (the later with correction rates in the order of 1 out of 100 accesses projected [11]), complex dependencies of instruction execution times, etc. All of this leads to a non-deterministic timing

- that is execution time jitter - and paired with preemtibility - to a non-deterministic global state from the perspective of the individual thread of execution.

In safety related or HA systems traditionally random faults have been mask by replication and redundancy - we take a similar approach here but at a much smaller scale - the critical object is a single data object and the "fault" is the writing process. We start with a well studied and simple class - a single writer multiple reader construct - similar to the one introduced by Peterson in his influencial paper "Concurrent reading while writing" [15], whereby the assumptions about the read and write operations are very much relaxed to reflect the nature of modern super-scalar multicores, that is no memory barriers or volatile data types are assumed. The design goals for race masking are:

- lock-less / 0-wait
- hand-shake-free
- non-atomic reader/writer
- constant number of steps for read and write (O(1))
- concurrent multiple reader, single writer
- never later than a locking version
- reader and writer crash safe
  - none can be blocked
  - no reader will receive an inconsistent or old value if the writer crashes
  - No bounds on the number of crashing processes
- an arbitrary probability of success can be provided (level of replication)
- failure probability decreases with increasing system complexity.
- failure probability decreases with increasing system load.
- failure probability decreases with the number of participating processes.

Scalability is not mentioned here simply because we don't yet have a good model to actually describe and analyze scalability but clearly scalability is a prime target. Wile we list non-atomic read-write it should be noted that we are assuming that single 32 bit entities are written atomically - that is a write of a word to a memory/register location will never permit an inconsistent concurrent read - either the old value is read in its entirety or the new value is read in its entirety but no "mix" of the two - any sane architecture will guarantee that (at least at present).

1.2 Concept

The concept is embarrassingly trivial, the writer simply writes to the shared object indescriminent of the state of any reader. Obviously this would not be safe for a single object - as with safety related systems where random faults must be covered - we simply view the concurrent threads as "randomly" accessing the data object and the writer is viewed as the "fault-injecter".

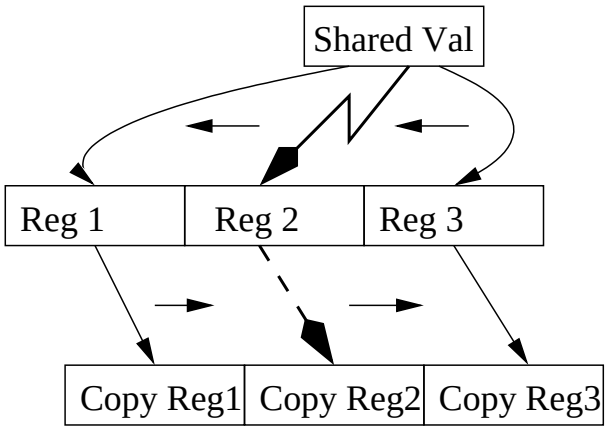

FIGURE 1: *fault-injection model*

Thus the therapy is simply replication. Rather than writing one protected region, we simply write N unprotected regions and with the inherent randomness of complex hardware software systems we can provide an arbitrary high probability that at least one of the regions is consistent at any time and thus can be retrieve by the readers. Summarized this simply means:

- probabilistic guarantee of success that can be set to arbitrary value
- write operation: write replicated registers unprotected
- read operations: copies replicated registers and selects

hence the name probabilistic Write/Copy-Select pW/CS lock.

textbfUnderlying principle:

The principles are roughly modeled along the lines of loosely coupled replicated systems to mitigate random faults in safety related systems:

- temporal serialization is replaced by spatial "concurrency"
- atomicity is replaced by a probability of success.
- atomicity of single object updates must be guaranteed (that is the write of a single 32bit word must be guaranteed to be consistent (that is the single load or store is consistent - which should hold on any CPU I hope).

replication is done to guarantee that ate last one register is consistent and complete at any time (with an arbitrary selectable probability) for a given assumed maximum synchronous preemptions of reader and writer thread. The value available is always the last complete value written (though an in-progress write may be incomplete) - in any case a reader always gets access to the last consistent register copy, thus never later than a locking solution would provide.

So pW/CS addresses consistency of concurrently accessed data - it does not address completeness nor ordering issues - it is the lowest level primitive for sharing non-atomic resources without introducing a joint context constraint.

1.3 Categorization

In Lamport's taxonomy this is a regular 1-writer algorithm

> A regular variable is a safe variable in which a Read that overlaps one or more Writes returns either the value of the most recent Write preceding the Read or of one of the overlapping Writes.

though Lamport's taxonomy [3] might not be applicable to a probabilistic locking scheme but it fulfills the criteria quite nicely. As this is a low-level primitive only, the motivation to build on such definitions is to allow deducing high-level constructs (i.e. monitors seem a quite natural option) to build on this primitive.

2 Register layout and protocol

In this section we describe the reader and writer protocols as well as the replicated register layout.

Replicated register set layout

The principle layout is simply a set of N registers with 2N markers guarding it, so a N-register for pW/CS protection would be:

```
[marker,reg1,marker]...[marker,regN,marker]
```

Note that the markers are to be unique if unbounded reader delays are permitted, if reader delays are bounded then the markers type space must be sufficient to cover uniqueness within the reader delay for un-delayed writers (or register aliasing could occur - i.e. a role-over of a marker if the marker were only a char). In the proof of concept implementation the largest inherently atomically writable object, a 32bit value, is used as marker.

read and write protocol

The readers and writers have a simple protocol to follow, basically the direction of access is inverted. The selection of direction is of course arbitrary - the essence only that readers and writers access in opposite direction. Now on weakly ordered architectures this might not hold (or require larger number of replicas) - but as the approach in it self is non-deterministic this does not matter as no consistency assumptions are actually made.

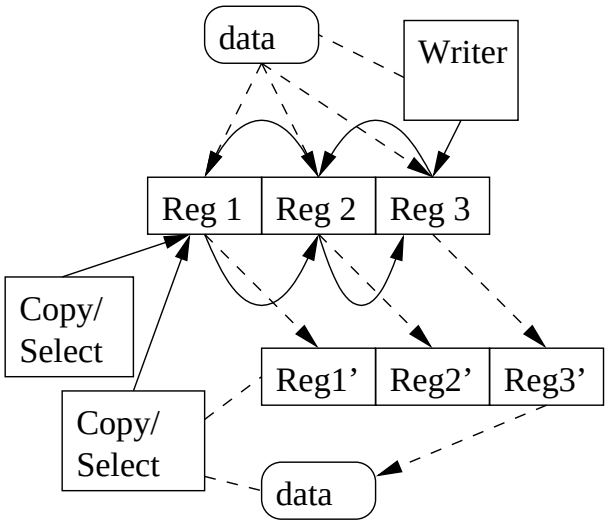

FIGURE 2: *access model*

- writer protocol:

  - update replicas (left to right)
  - update protocol:
    * update leading (left) marker
    * update register
    * update the trailing (right) marker

- reader protocol:
  - copy register set (right to left)
  - select consistent register
  - selection protocol:

```
for(reg=right,reg<left,reg--){
  if(markers identical ){
     select register
  }
}
return register
```

protection by write and read being in opposite directions. It is not possible to get inconsistent data even with a single pass - it is though possible to get no data (all data is found to be inconsistent). The probability of the occurrence of all data is inconsistent can though be brought down to an arbitrary low value with sufficient replication.

Note that there are possibilities for "smarter" protocols than the above brute-force one. For the proof-of-concept implementation this simple minded approach was shown to work just fine - and as it allows simple modeling it is what we are currently using.

3 race-masking with implicit reader locking

The initial motivation for looking into race masking was to allow lock-less coordination of real-time and non-real-time tasks on a real-time enhanced GNU/Linux system. Essentially this section is to show that the introduction of real-time priorities will also only improve the situation but never aggravate it in the sense that the probability of success is never reduced.

A non-probabilistic variant is by implicit priority locking of the reader - if the reader has higher priority than the writer then it is not possible for the writer to preempt the reader and thus it is guaranteed that the reader will be able to copy the entire buffer uninterrupted - in this case it can also be guaranteed that the reader gets at least one consistent copy if $N \geq 3$ replicas of the register are used.

This is nothing really new, Lamport suggested this in 1985 [3] suggesting the idea actually stems from 1977 but uses it in a deterministic algorithm to implemnt a multivalent regular register. Dijkstra proposes a non-deterministic slection in his paper titled "Guarded Commands, Nondeterminacy and Formal Derivation of Programs" 1975 [1] from which we use the idea of gards to protect a set of in principle non-deterministically selected actions (copying of the register). Interestingly enough Hoare in "Communicating Sequencial Processes" [2] describes a number of situations based on Dijkstras da nguarded commands that resemble the pW/CS locks proposed here, though the context is quite different. Ultimately non-determinism has been proposed in many publications though we are not aware of examples where this non-determinism is actually utilized - this alone is the novelty of the proposed design here and we believe it is potentially useful in resolving scalability problems in at least some situations.

4 General race-masking with probabilistic locking

If the requirement of well set priorities and thus one-sided non-preemptibility is dropped then there is a possibility that the read will return with none of the registers in a detectable consistent state. That is actually we don't know the state of the register - we infer positively that the register is consistent if the markers are identical. At the same time we can not positively infer an inconsistency of the register in case of markers being unequal though. But taking the inconsistency of the markers as indication of inconsistency of the registers is a pessimistic assumption in all cases and thus safe.

To fail the access to the registers must be strictly in lock-step order for readers and writers - for M replicas M*2+1 lock-step access would be needed to result in all registers being inconsistent.

A collision (all registers in an intermediate state) would require 2N+1 synchronous preemption. With synchronous preemption we mean that the preemption of the reader must occur after a complete register with markers was read every time and the preemption of the writer must happen in the middle of the register region every time plus that last read must also be preempted to ensure that no register is read in a consistent state.

Such an aliasing for $N = 3$ requires synchronous

preemption of reader and writer in at least 6 consecutive cases - this means a sixfold synchronous race conditions is needed to result in inconsistent data - what is the probability of such a scenario if single race conditions are hard to reproduce ?

In fact the race condition could be extended to arbitrary number of race hits to be needed to result in inconsistent data and thus one can provide arbitrary probability of success (at the expense of larger number of replications).

This solution could be described as, a somewhat paradox, "safe race" - safe to an arbitrary probability of successful reading of at least one consistent register.

The current proof-of-concept is for a register consisting of 3 integer values, but is extensible to any data structure - we note though that the prime interest is in resolving synchronization of small data objects, where race occurrence is very unlikely and thus traditional locking excessively wasteful.

The writer is simply an unconditional write to the register set.

```
do{
  /* unconditional write */
  ui[i].w_enter++;
  ui[i].period = period;
  ui[i].duty = duty;
  ui[i].bit = 1;
  ui[i].w_exit++;
  i++;
}while(i < NUM_REPLICA);
```

The reader copies the register set in reverse order and then runs a selection loop on it:

```
while(!exit_cond){
   i = NUM_REPLICA-1;
   do{
     pwm[i].w_enter = ui[i].w_enter;
     pwm[i].period = ui[i].period;
     pwm[i].duty = ui[i].duty;
     pwm[i].bit = ui[i].bit;
     pwm[i].w_exit = ui[i].w_exit;
     i--;
   }while( i >= 0);

   for(i=0;i<NUM_REPLICA;i++){
     if(pwm[i].w_enter - pwm[i].w_exit == 0){
       /* consistent register set found */
     }
   }
```

The selection can then simply set the pointer to the first valid entry found, note that the first found is the last written thus the most current of the N replicated registers so the selection can stop once a consistent register set was found. To ensure this the individual replicas though must be on cache line boundaries - if they were fit in a single cache line then the ordering implemented in the software would not necessarily be honored by the hardware.

5 Properties

5.1 Assessment of the randomness hypothesis

The most interesting issue in the experiments was to determine if the data can bolster the claim of a random fault scenario. If this assumption is false then obviously the underlying model would not be valid and consequently the conclusions also not - at least not at this point in time.

The random fault model is basically claiming that the writer actually has the same properties as a random fault injection - even though it is obviously systematic in nature, its timing is suspected to be truly random. If this holds then the mitigation of the fault also holds - with some constraints of course that will be developed a bit later.

To assess that the failures are actually random we take two main data samples into account.

- timing distribution
- distribution of single buffer inconsistencies

From this data, presented below, we can conclude that the writer process actually exhibits properties of a random fault (SEU).

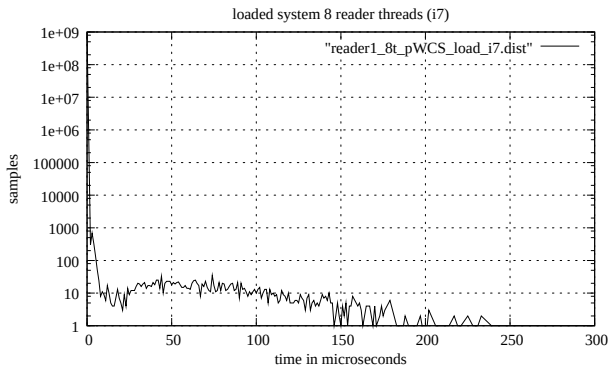

FIGURE 3: *timing distribution of one reader*

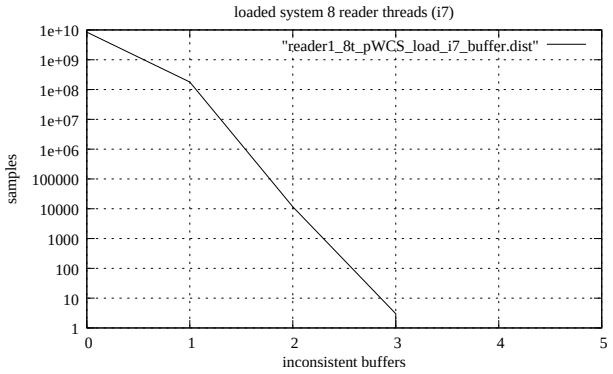

FIGURE 4: *buffer inconsistency distribution of one reader*

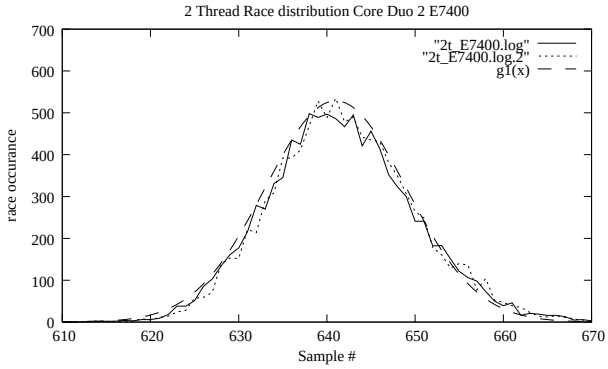

FIGURE 6: *distribution of race probability for given loop length (E7400 QuadCore)*

5.2 probability of failure

In the above example a 6-tuple replica was in use, for N registers 2*N+1 synchronous preemption are required - what is the probability of this happening ?

To see this we look at the distribution of the race occurrence on a single unprotected global integer over the loop length. 10000 runs are done and then the occurrence of races is plotted, showing the approximation of the race probability.

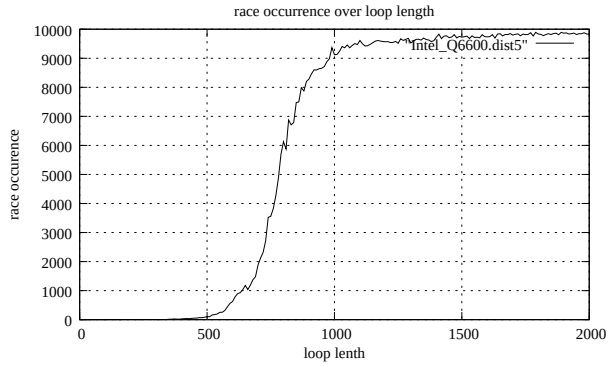

FIGURE 5: *race on single unprotected global variable with two threads over the loop length*

The actual occurrence of a race is almost perfectly normal distributed, if one creates 1000 instances of two racing threads for a given fixed loop length and records the number of races that occurred the distribution is close to a perfect gauss curve.

To calculate the probability of success we need a probability of a race condition in the first place. Thus the probability of one register actually being read inconsistent.

5.3 Failure rates of pW/CS

To estimate the failure rate we implemented a pW/CS protected data object and ran tests where we record the distribution of inconsistent registers (actually inconsistent markers which is a conservative indication of inconsistent registers) and plot the distribution for different scenarios. From this data we derive a model of the distribution and estimate the probabilities involved.

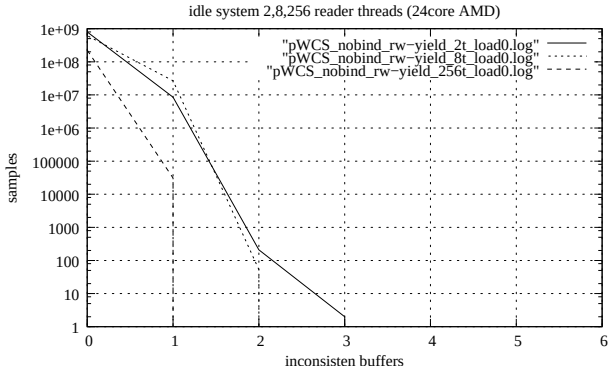

FIGURE 7: *buffer inconsistency distribution idle system (24 core AMD)*

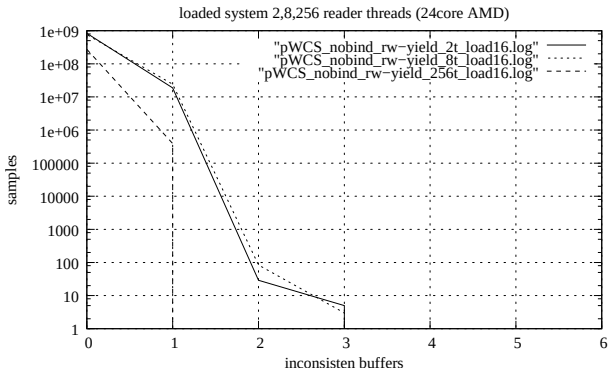

FIGURE 8: *buffer inconsistency distribution load 16 (24 core AMD)*

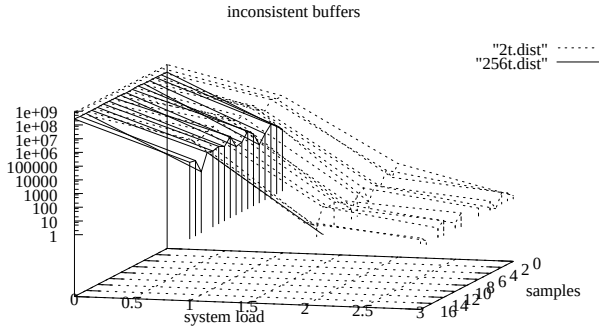

FIGURE 9: *buffer inconsistency distribution load sweep from 0 to 16, 2 threads vs 256 threads (24 core AMD)*

Note that the idle is the worst case (as expected). Further this code has a close to minimal loop body thus the probability of a preemption occurring in the critical section is very high and can be expected to be smaller in almost ever other case. Again this is quite the opposite of what you have in traditional locking where "keep it simple" is considered best-practice - with probabilistic locks increased complexity of access to the shared data is actually an advantage. The more erratic the system is the lower the probability of N lock-step preemptions leading to all buffers being inconsistent.

Notably current tests have shown that larger and thus more complex systems perform better than simpler systems - though we must note that we only had very limited access to large systems so the tests were generally only short runs and sometimes under not well specified load conditions.

5.4 Performance of pW/CS

The performance evaluation is quite preliminary, partially because we don't have a sufficiently complete model yet, partially because the test-case is quite artificial and it needs to be demonstrated that this actually holds for a real problem. The performance assessment is done by looking at the time it takes to access the shared data object and plotting this time as a histogram - it shows that the time distribution is very favorable for the probabilistic lock even on a loaded system (note that readers and writer are SCHED_OTHER not RR or FIFO).

The comparison is done between code using pW/CS and code using a normal pthread_nutex to protect the shared object.

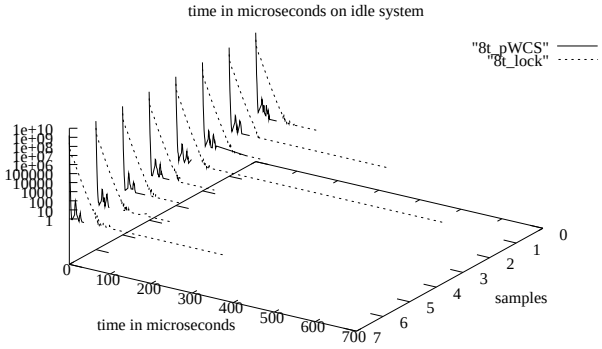

FIGURE 10: *timing idle system (4 core Intel)*

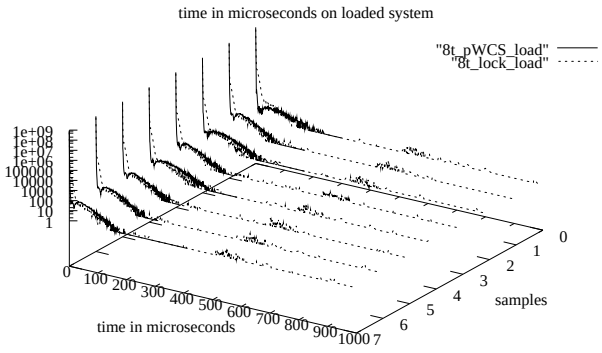

FIGURE 11: *timing loaded system (4 core Intel)*

Notably running the same test on a larger system - a 24 core AMD (2 CPUs) one can clearly see that the difference between the probabilistic approach and the deterministic approach widens.

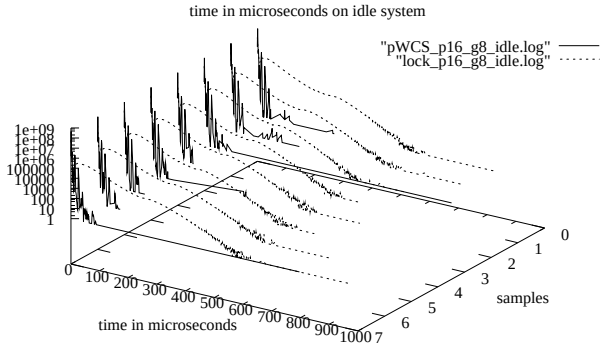

FIGURE 12: *timing idle system (24 core AMD)*

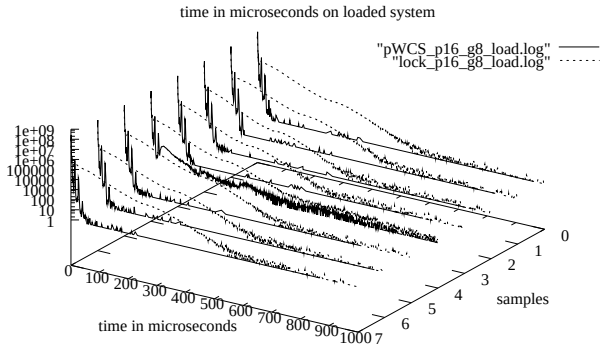

FIGURE 13: *timing loaded system (24 core AMD)*

If one can generalize these results is though still open due to the very limited test base we are able to utilize for this work.

5.5 Fairness

Fairness of any synchronization object is a critical issue. While locks don't exhibit much unfairness on small systems (single to 4 core systems) and mild load scenarios, the lock-fairness can become a major issue on 16 or 32++ way systems. Any new locking proposal thus must exhibit fairness and scalability (note that fairness can also be achieved "brute-force" at the price of performance and/or scalability - i.e. server type approaches).

pW/CS lock can't lead to reader nor writer starvation - if it does then the CPU was overloaded to begin with - but the locking regime it self does not contribute to un-fairness or starvation. with respect to scalability we are not yet sure if we can give it a thumbs-up, tests have been limited to a few systems up to now and only one was a 16 core (nehalem) and one 24 core AMD system, thus it is to early to say - from the conceptual side we believe that this is a scalable solution though.

5.6 Failure behavior

The maybe most interesting behavior of pW/CS is that a failure of one of the participating threads is simply irrelevant the writer can at best leave one data replica inconsistent , a reader would go entirely unnoticed as it does not alter the state of the shared object at any time.

pW/CS obviously does not have any double locking issues as there is no actual lock involved.

The prototype implementation used counters to check consistency, this is simple to implement but strictly not sufficient to guarantee correctness of the shared object, a better solution, though somewhat more involved computationally is to use a simple CRC to ensure consistency - first tests are running but were not ready on time for this paper (we need reasons to publish more papers on this any way...)

5.7 Testability

Traditional locking needs to be tested under load conditions, which are not only hard to generalize but never can cover all possible situations. Probabilistic approaches on the other hand can be designed to have their worst case probability of collision in the idle system - that is a high load improves the probability of a un-synchronized access and thus also decreases the probability of a collision which requires a complex synchronized access pattern. Thus in principle probabilistic locks are fully testable by testing on the idle system. We would like to emphasis that we don't yet see this proof-of-concept as verified to carry this property though we do think that it is possible to build synchronization that exhibits the property of "idle is the worst case".

6 Possible advantages

While "deterministic" code can't be exhaustively tested, and it is common that while testing a number of profiles, maybe for extensive periods of time, that the problem surfaces in either an untested profile or simply as a matter of time.

The root problem is that we can't reliably produce the "worst-case" situation on a system, not even on a fairly simple hardware/software system, thus leaving the occurrence of the worst-case to the field. With other words the problem becomes more likely with high-load situations and we can't test all possible combinations of high-load situations.

What is now behind the problem is that the race condition becomes a rare but possible problem - a specific global state of the system - if it occurs we fail. In this sense the failure is deterministic (functional view) but its dependency is relatively complex so it is hard to test. On the other hand the state of the good case (looking at the successful execution of the synchronization object) is well defined "deterministic" in most states - but the state space is very large so it is hard to achieve coverage. All we need to do is turn it around - make the race depend on a complex deterministic global state and make the good case independent of a particular state - that is - the good case should have a very large state space coverage, and the bad case a small and testable state-space.

The mitigation proposed here is to use synchronization that exhibits its worst-case behavior on the idle system. A probabilistic lock will have it highest probability of aliasing, and thus failing, in an idle system, and the higher or more erratic the load situation is the better it gets - because the probability of synchronous preemption that cause the possible collision decreases. Thus we regain the ability to test synchronization potentially as we can now reliably provide the worst-case with one single profile - the idle system. At the same time the good case does not have a single deterministic global state constellation but rather happens in a large number of independent states (that is only one register must be consistent from N).

A second aspect of raciness is the temporal dimension - in traditional systems one could observe that something that worked well for a long time suddenly fails because of optimization or faster hardware - we had the "implicit ordering" simply by the execution flow that protected the unprotected critical region. Now the probabilistic lock has exactly the opposite qualities, the fast the system gets the lower the probability of the reader not achieving a consistent read before being preempted, and this also holds for optimization of compilers - so again we can test the worst case - slow system, unoptimized code - it can only get better for the probability of the race condition not occurring in all N replicas of the register set.

Finally the issues of Amdahl's law, the longest serialized portion of code can quickly dominate the overall performance. As pW/CS has no serialization of readers and the writer there is no impact on concurrent threads by individual threads being preempted/blocked - the reader will always have access to the latest consistent buffer the writer was able to provide.

6.1 Next steps

The current proof-of-concept implementation is suboptimal in that it creates a N-replica copy for each reader. This is an unnecessary overhead in that it would be at most suitable to create such a "scratch-pad" per NUMA-node on a NUMA system, for non-NUMA a direct selection from the writers replicas is also an option. Further on the implementation side, the currently used counters should be replaced by a stronger consistency check - with the increasing overhead of accessing remote CPUs the overall local computation that can be expended if communication can be reduced to handshake-free (write-and-forget) semantics are conciderable, equally the expendable spatial overhead is considerable before approaching a lock based implementation (on a 4x4 system that we had access to temporarily a implementatoin using 100 replicas was still faster than a locking version for a shared object of 16 bytes !). More work needs to be done to understand where the break-even point would lie and consequently where this approach would be suitable.

The second large area of future work is in the modeling of this concept. The current approach of a quite brute-force implementation to get a better understanding of the approach and its potential is hardly suitable for actual deployment in a real system if no formal model is available for assessment. Unfortunately the available models don't fit the approach well. Maybe with the exception of Dijkstras guarded commands and non-determinstc if construct. The only real diffference being that while Dijkstras guarded commands evaluate the guards to determin if execution should take place, pW/CS unconditionally coplies the replicated register instance and then uses the "guards" to determin if the selection should take place or if the replica is abandoned - currently we intend to utilizing Dijkstras constructs to model pW/CS.

7 Conclusion

With ever growing complexity, designing deterministic while optimal systems is becoming increasingly hard (or actually impossible). In this paper we propose to look at potentially capitalizing on the growing complexity rather than fighting it - by utilizing the inherent randomness of complex systems in combination with probabilistic locking methods.

We demonstrate the feasibility of this approach with an admittedly naive implementation of a critical section shared between a concurrently executing readers and writer of arbitrary priority. The results indicate that with growing complexity of the system, with higher

system load, and with increased number of readers the probability of failing is reduced. Further faster systems have a higher probability of success than slower systems, and equally optimization of compiler plays to our advantage.

We are aware that this is too early to call this a sound and reliable result but the preliminary investigation does indicate that the proposed path - stop fighting complexity, use it ! - is worth investigation in more detail.

The most notable obstacle to utilize such approaches in our opinion is the lack of appropriate models for probabilistic approaches. This clearly will be our next steps in this effort to capitalize on the inadvertable trend of growing hardware and software complexity. Further a systematic tradeoff study, comparing traditional locking options in relation to system complexity will be on our TODO list.

The main conclusion from this work though is simply that locking may not be the best solution for concurrent access to shared objects - rethinking the problem in the context of modern super-scalar multicore systems might well be worth the effort.

sfsources used for this project are availabe on request under the GPL V2 from dslab [14]

References

[1] *Guarded Commands, Nondeterminancy and Formal Derivation of Programs*, Edsger W. Dijkstra, August 1975 COMMUNICATIONS OF THE ACM VOLME 18 NUMBER 8

[2] *Communicating Sequencial Processes*, C.A.R Hoare, August 1978 COMMUNICATIONS OF THE ACM,VOLUME 21, NUMBER 8

[3] *On Interprocess Communication*, Leslie Lamport, December 1985 – Part I: Basic formalism, Part II: Algorithms. SRI INTERNATIONAL

[4] *ELC: A PREEMPT_RT roadmap*,Jake Edge on a talk by Thomas Gleixner, April 2011 HTTP://LWN.NET/ARTICLES/440064/

[5] *Linux Kernel Development (3ed Edition)*, Robert Love, July 2010 ADDISON-WESLEY

[6] *migrate disable infrastructure*, Peter Zijlstra, July 2011 LINUX 3.0-RC7-RT0

[7] *fasync() BKL pushdown*, Jonathan Corbet, June 2008, HTTP://LWN.NET/ARTICLES/287083/

[8] *hrtimers and beyond transformation of the Linux time(r) system*, Thomas Gleixner, Douglas Niehaus, 2006 HTTP://WWW.KERNEL.ORG/PUB/LINUX/KERNEL/PEOPLE/TGLX/HRTIMERS/OLS2006-HRTIMERS.PDF

[9] *Analysis of inherent randomness of the Linux kernel*, Nicholas Mc Guire, Peter Odhiambo Okech, September 2009 DSLAB LANZHOU UNIVERSITY

[10] Completely Fair Scheduler,Ingo Molnr, October 2007, LINUX 2.6.23

[11] *LEC(TM) for Flash Memory*,Lyric Semiconductor,Undated,WWW.LYRICSEMICONDUCTOR.COM http://gigaom.com/2010/08/16/lyric-semiconducto/

[12] *sched_fair.c*, Ingo Molnar,Peter Zijlstra, et. al, LINUX-2.6/KERNEL/SCHED_FAIR.C

[13] *Lockdependency Validator*,Ingo Molnar,Arjan van de Ven, May 2006 HTTP://LWN.NET/ARTICLES/185605/

[14] *pWCS, gauss,c,dist.c*,Nicholas Mc Guire, 2011,HTTP://DSLAB.LZU.EDU.CN:8080/MEMBERS/HOFRAT/PWCS/

[15] *Concurrent reading while writing*, Gary L. Peterson, James E. Burns, 1983. ACM Transactions on Programming Languages and Systems